MPI: A Message-Passing Interface Standard
Version 2.1

Message Passing Interface Forum

June 23, 2008

MPI: A Message-Passing Interface Standard
Version 2.1

Message Passing Interface Forum

June 23, 2008

This document describes the Message-Passing Interface (MPI) standard, version 2.1. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined.

Technically, this version of the standard is based on "MPI: A Message-Passing Interface Standard, June 12, 1995" (MPI-1.1) from the MPI-1 Forum, and "MPI-2: Extensions to the Message-Passing Interface, July, 1997" (MPI-1.2 and MPI-2.0) from the MPI-2 Forum, and errata documents from the MPI Forum.

Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1 (June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2 and some errata documents to one combined document, and this document, MPI-2.1, combining the previous documents. Additional clarifications and errata corrections to MPI-2.0 are also included.

This document describes the Message-Passing Interface (MPI) standard, version 2.1. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined.

Technically, this version of the standard is based on "MPI: A Message-Passing Interface Standard, June 12, 1995" (MPI-1.1) from the MPI-1 Forum, and "MPI-2: Extensions to the Message-Passing Interface, July, 1997" (MPI-1.2 and MPI-2.0) from the MPI-2 Forum, and errata documents from the MPI Forum.

Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1 (June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2 and some errata documents to one combined document, and this document, MPI-2.1, combining the previous documents. Additional clarifications and errata corrections to MPI-2.0 are also included.

©1993, 1994, 1995, 1996, 1997, 2008 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

ii

Version 2.1: June 23, 2008, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations have been merged into the Chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document.

Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 Chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document.

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that don't fit elsewhere, in particular language interoperability.

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only new function in MPI-1.2 is one for identifying to which version of the MPI Standard the implementation conforms. There are small differences between MPI-1 and MPI-1.1. There are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2 and MPI-2.

Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum reconvened to correct errors and make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1, which is this document. The changes from Version 1.0 are minor. A version of this document with all changes marked is available. This paragraph is an example of a change.

Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message-passing.

This is the final report, Version 1.0, of the Message-Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LaTeX on May 5, 1994.

Please send comments on MPI to `mpi-comments@mpi-forum.org`. Your comment will be forwarded to MPI Forum committee members who will attempt to respond.

Version 2.1: June 23, 2008, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations have been merged into the Chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document.

Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 Chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document.

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that don't fit elsewhere, in particular language interoperability.

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only new function in MPI-1.2 is one for identifying to which version of the MPI Standard the implementation conforms. There are small differences between MPI-1 and MPI-1.1. There are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2 and MPI-2.

Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum reconvened to correct errors and make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1, which is this document. The changes from Version 1.0 are minor. A version of this document with all changes marked is available. This paragraph is an example of a change.

Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message-passing.

This is the final report, Version 1.0, of the Message-Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LaTeX on May 5, 1994.

Please send comments on MPI to `mpi-comments@mpi-forum.org`. Your comment will be forwarded to MPI Forum committee members who will attempt to respond.

iii

Contents

Contents

Acknowledgments xvii

1 Introduction to MPI 1
1.1 Overview and Goals . . . 1
1.2 Background of MPI-1.0 . . . 2
1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0 . . . 3
1.4 Background of MPI-1.3 and MPI-2.1 . . . 3
1.5 Who Should Use This Standard? . . . 4
1.6 What Platforms Are Targets For Implementation? . . . 4
1.7 What Is Included In The Standard? . . . 5
1.8 What Is Not Included In The Standard? . . . 5
1.9 Organization of this Document . . . 6

2 MPI Terms and Conventions 9
2.1 Document Notation . . . 9
2.2 Naming Conventions . . . 9
2.3 Procedure Specification . . . 10
2.4 Semantic Terms . . . 11
2.5 Data Types . . . 12
2.5.1 Opaque Objects . . . 12
2.5.2 Array Arguments . . . 14
2.5.3 State . . . 14
2.5.4 Named Constants . . . 14
2.5.5 Choice . . . 15
2.5.6 Addresses . . . 15
2.5.7 File Offsets . . . 15
2.6 Language Binding . . . 15
2.6.1 Deprecated Names and Functions . . . 16
2.6.2 Fortran Binding Issues . . . 16
2.6.3 C Binding Issues . . . 18
2.6.4 C++ Binding Issues . . . 18
2.6.5 Functions and Macros . . . 21
2.7 Processes . . . 22
2.8 Error Handling . . . 22
2.9 Implementation Issues . . . 23
2.9.1 Independence of Basic Runtime Routines . . . 23
2.9.2 Interaction with Signals . . . 24

v

2.10 Examples 24

3 **Point-to-Point Communication** **25**
3.1 Introduction 25
3.2 Blocking Send and Receive Operations 26
3.2.1 Blocking Send 26
3.2.2 Message Data 27
3.2.3 Message Envelope 28
3.2.4 Blocking Receive 29
3.2.5 Return Status 31
3.2.6 Passing MPI_STATUS_IGNORE for Status 33
3.3 Data Type Matching and Data Conversion 34
3.3.1 Type Matching Rules 34
Type MPI_CHARACTER 36
3.3.2 Data Conversion 36
3.4 Communication Modes 38
3.5 Semantics of Point-to-Point Communication 41
3.6 Buffer Allocation and Usage 45
3.6.1 Model Implementation of Buffered Mode 47
3.7 Nonblocking Communication 47
3.7.1 Communication Request Objects 49
3.7.2 Communication Initiation 49
3.7.3 Communication Completion 52
3.7.4 Semantics of Nonblocking Communications 56
3.7.5 Multiple Completions 57
3.7.6 Non-destructive Test of status 63
3.8 Probe and Cancel 64
3.9 Persistent Communication Requests 68
3.10 Send-Receive 73
3.11 Null Processes 75

4 **Datatypes** **77**
4.1 Derived Datatypes 77
4.1.1 Type Constructors with Explicit Addresses 79
4.1.2 Datatype Constructors 79
4.1.3 Subarray Datatype Constructor 87
4.1.4 Distributed Array Datatype Constructor 89
4.1.5 Address and Size Functions 94
4.1.6 Lower-Bound and Upper-Bound Markers 96
4.1.7 Extent and Bounds of Datatypes 97
4.1.8 True Extent of Datatypes 98
4.1.9 Commit and Free 99
4.1.10 Duplicating a Datatype 100
4.1.11 Use of General Datatypes in Communication 101
4.1.12 Correct Use of Addresses 103
4.1.13 Decoding a Datatype 104
4.1.14 Examples 111
4.2 Pack and Unpack 120

vi

4.3 Canonical MPI_PACK and MPI_UNPACK 127

5 Collective Communication 129
5.1 Introduction and Overview 129
5.2 Communicator Argument 132
5.2.1 Specifics for Intracommunicator Collective Operations 132
5.2.2 Applying Collective Operations to Intercommunicators 133
5.2.3 Specifics for Intercommunicator Collective Operations 134
5.3 Barrier Synchronization 135
5.4 Broadcast 136
5.4.1 Example using MPI_BCAST 136
5.5 Gather 137
5.5.1 Examples using MPI_GATHER, MPI_GATHERV 140
5.6 Scatter 147
5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV 149
5.7 Gather-to-all 152
5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV 154
5.8 All-to-All Scatter/Gather 155
5.9 Global Reduction Operations 159
5.9.1 Reduce 160
5.9.2 Predefined Reduction Operations 161
5.9.3 Signed Characters and Reductions 163
5.9.4 MINLOC and MAXLOC 164
5.9.5 User-Defined Reduction Operations 168
Example of User-defined Reduce 170
5.9.6 All-Reduce 171
5.10 Reduce-Scatter 173
5.11 Scan 174
5.11.1 Inclusive Scan 174
5.11.2 Exclusive Scan 175
5.11.3 Example using MPI_SCAN 176
5.12 Correctness 177

6 Groups, Contexts, Communicators, and Caching 181
6.1 Introduction 181
6.1.1 Features Needed to Support Libraries 181
6.1.2 MPI's Support for Libraries 182
6.2 Basic Concepts 184
6.2.1 Groups 184
6.2.2 Contexts 184
6.2.3 Intra-Communicators 185
6.2.4 Predefined Intra-Communicators 185
6.3 Group Management 186
6.3.1 Group Accessors 186
6.3.2 Group Constructors 187
6.3.3 Group Destructors 192
6.4 Communicator Management 193
6.4.1 Communicator Accessors 193

vii

6.4.2 Communicator Constructors . . . 194
6.4.3 Communicator Destructors . . . 201
6.5 Motivating Examples . . . 202
6.5.1 Current Practice #1 . . . 202
6.5.2 Current Practice #2 . . . 203
6.5.3 (Approximate) Current Practice #3 . . . 203
6.5.4 Example #4 . . . 204
6.5.5 Library Example #1 . . . 205
6.5.6 Library Example #2 . . . 207
6.6 Inter-Communication . . . 209
6.6.1 Inter-communicator Accessors . . . 210
6.6.2 Inter-communicator Operations . . . 212
6.6.3 Inter-Communication Examples . . . 214
Example 1: Three-Group "Pipeline" . . . 214
Example 2: Three-Group "Ring" . . . 216
Example 3: Building Name Service for Intercommunication . . . 217
6.7 Caching . . . 221
6.7.1 Functionality . . . 222
6.7.2 Communicators . . . 223
6.7.3 Windows . . . 227
6.7.4 Datatypes . . . 230
6.7.5 Error Class for Invalid Keyval . . . 233
6.7.6 Attributes Example . . . 233
6.8 Naming Objects . . . 235
6.9 Formalizing the Loosely Synchronous Model . . . 239
6.9.1 Basic Statements . . . 239
6.9.2 Models of Execution . . . 239
Static communicator allocation . . . 239
Dynamic communicator allocation . . . 240
The General case . . . 240

7 Process Topologies 241
7.1 Introduction . . . 241
7.2 Virtual Topologies . . . 242
7.3 Embedding in MPI . . . 242
7.4 Overview of the Functions . . . 243
7.5 Topology Constructors . . . 244
7.5.1 Cartesian Constructor . . . 244
7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE . . . 244
7.5.3 General (Graph) Constructor . . . 246
7.5.4 Topology Inquiry Functions . . . 248
7.5.5 Cartesian Shift Coordinates . . . 252
7.5.6 Partitioning of Cartesian structures . . . 254
7.5.7 Low-Level Topology Functions . . . 254
7.6 An Application Example . . . 256

viii

8 MPI **Environmental Management** **259**
8.1 Implementation Information . . . 259
8.1.1 Version Inquiries . . . 259
8.1.2 Environmental Inquiries . . . 260
Tag Values . . . 260
Host Rank . . . 260
IO Rank . . . 261
Clock Synchronization . . . 261
8.2 Memory Allocation . . . 262
8.3 Error Handling . . . 264
8.3.1 Error Handlers for Communicators . . . 265
8.3.2 Error Handlers for Windows . . . 267
8.3.3 Error Handlers for Files . . . 269
8.3.4 Freeing Errorhandlers and Retrieving Error Strings . . . 270
8.4 Error Codes and Classes . . . 271
8.5 Error Classes, Error Codes, and Error Handlers . . . 273
8.6 Timers and Synchronization . . . 277
8.7 Startup . . . 278
8.7.1 Allowing User Functions at Process Termination . . . 283
8.7.2 Determining Whether MPI Has Finished . . . 284
8.8 Portable MPI Process Startup . . . 284

9 The Info **Object** **287**

10 Process Creation and Management **293**
10.1 Introduction . . . 293
10.2 The Dynamic Process Model . . . 294
10.2.1 Starting Processes . . . 294
10.2.2 The Runtime Environment . . . 294
10.3 Process Manager Interface . . . 296
10.3.1 Processes in MPI . . . 296
10.3.2 Starting Processes and Establishing Communication . . . 296
10.3.3 Starting Multiple Executables and Establishing Communication . . . 301
10.3.4 Reserved Keys . . . 303
10.3.5 Spawn Example . . . 304
Manager-worker Example, Using MPI_COMM_SPAWN. . . . 304
10.4 Establishing Communication . . . 306
10.4.1 Names, Addresses, Ports, and All That . . . 306
10.4.2 Server Routines . . . 308
10.4.3 Client Routines . . . 309
10.4.4 Name Publishing . . . 311
10.4.5 Reserved Key Values . . . 313
10.4.6 Client/Server Examples . . . 313
Simplest Example — Completely Portable. . . . 313
Ocean/Atmosphere - Relies on Name Publishing . . . 314
Simple Client-Server Example. . . . 314
10.5 Other Functionality . . . 316
10.5.1 Universe Size . . . 316

ix

10.5.2 Singleton MPI_INIT . . . 316
10.5.3 MPI_APPNUM . . . 317
10.5.4 Releasing Connections . . . 318
10.5.5 Another Way to Establish MPI Communication . . . 319

11 One-Sided Communications **321**
11.1 Introduction . . . 321
11.2 Initialization . . . 322
11.2.1 Window Creation . . . 322
11.2.2 Window Attributes . . . 324
11.3 Communication Calls . . . 325
11.3.1 Put . . . 326
11.3.2 Get . . . 328
11.3.3 Examples . . . 328
11.3.4 Accumulate Functions . . . 331
11.4 Synchronization Calls . . . 333
11.4.1 Fence . . . 338
11.4.2 General Active Target Synchronization . . . 339
11.4.3 Lock . . . 342
11.4.4 Assertions . . . 344
11.4.5 Miscellaneous Clarifications . . . 346
11.5 Examples . . . 346
11.6 Error Handling . . . 348
11.6.1 Error Handlers . . . 348
11.6.2 Error Classes . . . 349
11.7 Semantics and Correctness . . . 349
11.7.1 Atomicity . . . 352
11.7.2 Progress . . . 352
11.7.3 Registers and Compiler Optimizations . . . 354

12 External Interfaces **357**
12.1 Introduction . . . 357
12.2 Generalized Requests . . . 357
12.2.1 Examples . . . 361
12.3 Associating Information with Status . . . 363
12.4 MPI and Threads . . . 365
12.4.1 General . . . 365
12.4.2 Clarifications . . . 366
12.4.3 Initialization . . . 368

13 I/O **373**
13.1 Introduction . . . 373
13.1.1 Definitions . . . 373
13.2 File Manipulation . . . 375
13.2.1 Opening a File . . . 375
13.2.2 Closing a File . . . 377
13.2.3 Deleting a File . . . 378
13.2.4 Resizing a File . . . 379

x

13.2.5 Preallocating Space for a File . . . 379
13.2.6 Querying the Size of a File . . . 380
13.2.7 Querying File Parameters . . . 380
13.2.8 File Info . . . 382
Reserved File Hints . . . 383
13.3 File Views . . . 385
13.4 Data Access . . . 387
13.4.1 Data Access Routines . . . 387
Positioning . . . 388
Synchronism . . . 389
Coordination . . . 389
Data Access Conventions . . . 389
13.4.2 Data Access with Explicit Offsets . . . 390
13.4.3 Data Access with Individual File Pointers . . . 394
13.4.4 Data Access with Shared File Pointers . . . 399
Noncollective Operations . . . 400
Collective Operations . . . 402
Seek . . . 403
13.4.5 Split Collective Data Access Routines . . . 404
13.5 File Interoperability . . . 410
13.5.1 Datatypes for File Interoperability . . . 412
13.5.2 External Data Representation: "external32" . . . 414
13.5.3 User-Defined Data Representations . . . 415
Extent Callback . . . 417
Datarep Conversion Functions . . . 417
13.5.4 Matching Data Representations . . . 419
13.6 Consistency and Semantics . . . 420
13.6.1 File Consistency . . . 420
13.6.2 Random Access vs. Sequential Files . . . 423
13.6.3 Progress . . . 423
13.6.4 Collective File Operations . . . 423
13.6.5 Type Matching . . . 424
13.6.6 Miscellaneous Clarifications . . . 424
13.6.7 MPI_Offset Type . . . 424
13.6.8 Logical vs. Physical File Layout . . . 424
13.6.9 File Size . . . 425
13.6.10 Examples . . . 425
Asynchronous I/O . . . 428
13.7 I/O Error Handling . . . 429
13.8 I/O Error Classes . . . 430
13.9 Examples . . . 430
13.9.1 Double Buffering with Split Collective I/O . . . 430
13.9.2 Subarray Filetype Constructor . . . 433

xi

14 Profiling Interface **435**
14.1 Requirements . . . 435
14.2 Discussion . . . 435
14.3 Logic of the Design . . . 436
14.3.1 Miscellaneous Control of Profiling . . . 436
14.4 Examples . . . 437
14.4.1 Profiler Implementation . . . 437
14.4.2 MPI Library Implementation . . . 438
Systems with Weak Symbols . . . 438
Systems Without Weak Symbols . . . 438
14.4.3 Complications . . . 439
Multiple Counting . . . 439
Linker Oddities . . . 439
14.5 Multiple Levels of Interception . . . 440

15 Deprecated Functions **441**
15.1 Deprecated since MPI-2.0 . . . 441

16 Language Bindings **449**
16.1 C++ . . . 449
16.1.1 Overview . . . 449
16.1.2 Design . . . 449
16.1.3 C++ Classes for MPI . . . 450
16.1.4 Class Member Functions for MPI . . . 450
16.1.5 Semantics . . . 451
16.1.6 C++ Datatypes . . . 453
16.1.7 Communicators . . . 455
16.1.8 Exceptions . . . 457
16.1.9 Mixed-Language Operability . . . 458
16.1.10 Profiling . . . 458
16.2 Fortran Support . . . 461
16.2.1 Overview . . . 461
16.2.2 Problems With Fortran Bindings for MPI . . . 462
Problems Due to Strong Typing . . . 463
Problems Due to Data Copying and Sequence Association . . . 463
Special Constants . . . 465
Fortran 90 Derived Types . . . 465
A Problem with Register Optimization . . . 466
16.2.3 Basic Fortran Support . . . 468
16.2.4 Extended Fortran Support . . . 469
The `mpi` Module . . . 469
No Type Mismatch Problems for Subroutines with Choice Arguments 470
16.2.5 Additional Support for Fortran Numeric Intrinsic Types . . . 470
Parameterized Datatypes with Specified Precision and Exponent Range 471
Support for Size-specific MPI Datatypes . . . 474
Communication With Size-specific Types . . . 476
16.3 Language Interoperability . . . 478
16.3.1 Introduction . . . 478

xii

16.3.2 Assumptions 478
16.3.3 Initialization 479
16.3.4 Transfer of Handles 479
16.3.5 Status 482
16.3.6 MPI Opaque Objects 483
Datatypes 483
Callback Functions 485
Error Handlers 485
Reduce Operations 485
Addresses 485
16.3.7 Attributes 486
16.3.8 Extra State 488
16.3.9 Constants 488
16.3.10 Interlanguage Communication 489

A Language Bindings Summary **491**
A.1 Defined Values and Handles 491
A.1.1 Defined Constants 491
A.1.2 Types 499
A.1.3 Prototype definitions 500
A.1.4 Deprecated prototype definitions 504
A.1.5 Info Keys 504
A.1.6 Info Values 505
A.2 C Bindings 506
A.2.1 Point-to-Point Communication C Bindings 506
A.2.2 Datatypes C Bindings 507
A.2.3 Collective Communication C Bindings 509
A.2.4 Groups, Contexts, Communicators, and Caching C Bindings 510
A.2.5 Process Topologies C Bindings 513
A.2.6 MPI Environmenta Management C Bindings 513
A.2.7 The Info Object C Bindings 514
A.2.8 Process Creation and Management C Bindings 515
A.2.9 One-Sided Communications C Bindings 515
A.2.10 External Interfaces C Bindings 516
A.2.11 I/O C Bindings 516
A.2.12 Language Bindings C Bindings 519
A.2.13 Profiling Interface C Bindings 520
A.2.14 Deprecated C Bindings 520
A.3 Fortran Bindings 521
A.3.1 Point-to-Point Communication Fortran Bindings 521
A.3.2 Datatypes Fortran Bindings 523
A.3.3 Collective Communication Fortran Bindings 526
A.3.4 Groups, Contexts, Communicators, and Caching Fortran Bindings . 527
A.3.5 Process Topologies Fortran Bindings 532
A.3.6 MPI Environmenta Management Fortran Bindings 533
A.3.7 The Info Object Fortran Bindings 534
A.3.8 Process Creation and Management Fortran Bindings 535
A.3.9 One-Sided Communications Fortran Bindings 536

xiii

A.3.10 External Interfaces Fortran Bindings . . . 537
A.3.11 I/O Fortran Bindings . . . 538
A.3.12 Language Bindings Fortran Bindings . . . 542
A.3.13 Profiling Interface Fortran Bindings . . . 542
A.3.14 Deprecated Fortran Bindings . . . 542
A.4 C++ Bindings . . . 545
A.4.1 Point-to-Point Communication C++ Bindings . . . 545
A.4.2 Datatypes C++ Bindings . . . 547
A.4.3 Collective Communication C++ Bindings . . . 549
A.4.4 Groups, Contexts, Communicators, and Caching C++ Bindings . . . 550
A.4.5 Process Topologies C++ Bindings . . . 552
A.4.6 MPI Environmenta Management C++ Bindings . . . 553
A.4.7 The Info Object C++ Bindings . . . 554
A.4.8 Process Creation and Management C++ Bindings . . . 554
A.4.9 One-Sided Communications C++ Bindings . . . 555
A.4.10 External Interfaces C++ Bindings . . . 556
A.4.11 I/O C++ Bindings . . . 556
A.4.12 Language Bindings C++ Bindings . . . 560
A.4.13 Profiling Interface C++ Bindings . . . 560
A.4.14 Deprecated C++ Bindings . . . 560
A.4.15 C++ Bindings on all MPI Classes . . . 560
A.4.16 Construction / Destruction . . . 561
A.4.17 Copy / Assignment . . . 561
A.4.18 Comparison . . . 561
A.4.19 Inter-language Operability . . . 561

B Change-Log **562**
B.1 Changes from Version 2.0 to Version 2.1 . . . 562

Bibliography **567**

Examples Index **571**

MPI Constant and Predefined Handle Index **574**

MPI Declarations Index **578**

MPI Callback Function Prototype Index **580**

MPI Function Index **581**

xiv

List of Figures

List of Figures

5.1 Collective comminucations, an overview 130
5.2 Intercommunicator allgather 134
5.3 Intercommunicator reduce-scatter 135
5.4 Gather example 141
5.5 Gatherv example with strides 142
5.6 Gatherv example, 2-dimensional 142
5.7 Gatherv example, 2-dimensional, subarrays with different sizes 143
5.8 Gatherv example, 2-dimensional, subarrays with different sizes and strides . 145
5.9 Scatter example 150
5.10 Scatterv example with strides 151
5.11 Scatterv example with different strides and counts 152
5.12 Race conditions with point-to-point and collective communications 179

6.1 Intercommunicator create using MPI_COMM_CREATE 197
6.2 Intercommunicator constructionwith MPI_COMM_SPLIT 200
6.3 Three-group pipeline. 214
6.4 Three-group ring. 215

7.1 Set-up of process structure for two-dimensional parallel Poisson solver. . . . 257

11.1 Active target communication 335
11.2 Active target communication, with weak synchronization 336
11.3 Passive target communication 337
11.4 Active target communication with several processes 341
11.5 Schematic description of window 350
11.6 Symmetric communication 353
11.7 Deadlock situation 353
11.8 No deadlock 354

13.1 Etypes and filetypes 374
13.2 Partitioning a file among parallel processes 374
13.3 Displacements 386
13.4 Example array file layout 433
13.5 Example local array filetype for process 1 433

xv

List of Tables

List of Tables

2.1 Deprecated constructs 17

3.1 Predefined MPI datatypes corresponding to Fortran datatypes 27
3.2 Predefined MPI datatypes corresponding to C datatypes 28

4.1 combiner values returned from MPI_TYPE_GET_ENVELOPE 106

6.1 MPI_COMM_* Function Behavior (in Inter-Communication Mode) 211

8.1 Error classes (Part 1) 272
8.2 Error classes (Part 2) 273

11.1 Error classes in one-sided communication routines 349

13.1 Data access routines 388
13.2 "external32" sizes of predefined datatypes 416
13.3 I/O Error Classes 431

16.1 C++ names for the MPI C and C++ predefined datatypes 454
16.2 C++ names for the MPI Fortran predefined datatypes 454
16.3 C++ names for other MPI datatypes 455

xvi

Acknowledgments

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort.

Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are:

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communications
- Al Geist, Marc Snir, Steve Otto, Collective Communications
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith

Acknowledgments

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort.

Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are:

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communications
- Al Geist, Marc Snir, Steve Otto, Collective Communications
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith

xvii

The University of Tennessee and Oak Ridge National Laboratory made the draft available by anonymous FTP mail servers and were instrumental in distributing the document.

The work on the MPI-1 standard was supported in part by ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643 (PPPE).

MPI-1.2 and MPI-2.0:

Those who served as primary coordinators in MPI-1.2 and MPI-2.0 are:

- Ewing Lusk, Convener and Meeting Chair
- Steve Huss-Lederman, Editor
- Ewing Lusk, Miscellany
- Bill Saphir, Process Creation and Management
- Marc Snir, One-Sided Communications
- Bill Gropp and Anthony Skjellum, Extended Collective Operations
- Steve Huss-Lederman, External Interfaces
- Bill Nitzberg, I/O
- Andrew Lumsdaine, Bill Saphir, and Jeff Squyres, Language Bindings
- Anthony Skjellum and Arkady Kanevsky, Real-Time

The following list includes some of the active participants who attended MPI-2 Forum meetings and are not mentioned above.

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen
Ying Chen	Albert Cheng	Yong Cho	Joel Clark
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie
Zhenqian Cui	Suresh Damodaran-Kamal		Raja Daoud
Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman
Ian Foster	Hubertus Franke	Richard Frost	Al Geist
Robert George	David Greenberg	John Hagedorn	Kei Harada
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica
Steve Landherr	Mario Lauria	Mark Law	Juan Leon
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams

The University of Tennessee and Oak Ridge National Laboratory made the draft available by anonymous FTP mail servers and were instrumental in distributing the document.

The work on the MPI-1 standard was supported in part by ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643 (PPPE).

MPI-1.2 and MPI-2.0:

Those who served as primary coordinators in MPI-1.2 and MPI-2.0 are:

- Ewing Lusk, Convener and Meeting Chair
- Steve Huss-Lederman, Editor
- Ewing Lusk, Miscellany
- Bill Saphir, Process Creation and Management
- Marc Snir, One-Sided Communications
- Bill Gropp and Anthony Skjellum, Extended Collective Operations
- Steve Huss-Lederman, External Interfaces
- Bill Nitzberg, I/O
- Andrew Lumsdaine, Bill Saphir, and Jeff Squyres, Language Bindings
- Anthony Skjellum and Arkady Kanevsky, Real-Time

The following list includes some of the active participants who attended MPI-2 Forum meetings and are not mentioned above.

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen
Ying Chen	Albert Cheng	Yong Cho	Joel Clark
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie
Zhenqian Cui	Suresh Damodaran-Kamal		Raja Daoud
Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman
Ian Foster	Hubertus Franke	Richard Frost	Al Geist
Robert George	David Greenberg	John Hagedorn	Kei Harada
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica
Steve Landherr	Mario Lauria	Mark Law	Juan Leon
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams

xviii

John May	Oliver McBryan	Brian McCandless	Tyce McLarty
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong
Dave Wright			

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Bolt, Beranek, and Newman
California Institute of Technology
Center for Computing Sciences
Convex Computer Corporation
Cray Research
Digital Equipment Corporation
Dolphin Interconnect Solutions, Inc.
Edinburgh Parallel Computing Centre
General Electric Company
German National Research Center for Information Technology
Hewlett-Packard
Hitachi
Hughes Aircraft Company
Intel Corporation
International Business Machines
Khoral Research
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
MPI Software Techology, Inc.
Mississippi State University
NEC Corporation
National Aeronautics and Space Administration
National Energy Research Scientific Computing Center
National Institute of Standards and Technology
National Oceanic and Atmospheric Adminstration
Oak Ridge National Laboratory
Ohio State University
PALLAS GmbH

John May	Oliver McBryan	Brian McCandless	Tyce McLarty
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong
Dave Wright			

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Bolt, Beranek, and Newman
California Institute of Technology
Center for Computing Sciences
Convex Computer Corporation
Cray Research
Digital Equipment Corporation
Dolphin Interconnect Solutions, Inc.
Edinburgh Parallel Computing Centre
General Electric Company
German National Research Center for Information Technology
Hewlett-Packard
Hitachi
Hughes Aircraft Company
Intel Corporation
International Business Machines
Khoral Research
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
MPI Software Techology, Inc.
Mississippi State University
NEC Corporation
National Aeronautics and Space Administration
National Energy Research Scientific Computing Center
National Institute of Standards and Technology
National Oceanic and Atmospheric Adminstration
Oak Ridge National Laboratory
Ohio State University
PALLAS GmbH

xix

Pacific Northwest National Laboratory
Pratt & Whitney
San Diego Supercomputer Center
Sanders, A Lockheed-Martin Company
Sandia National Laboratories
Schlumberger
Scientific Computing Associates, Inc.
Silicon Graphics Incorporated
Sky Computers
Sun Microsystems Computer Corporation
Syracuse University
The MITRE Corporation
Thinking Machines Corporation
United States Navy
University of Colorado
University of Denver
University of Houston
University of Illinois
University of Maryland
University of Notre Dame
University of San Fransisco
University of Stuttgart Computing Center
University of Wisconsin

MPI-2 operated on a very tight budget (in reality, it had no budget when the first meeting was announced). Many institutions helped the MPI-2 effort by supporting the efforts and travel of the members of the MPI Forum. Direct support was given by NSF and DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

MPI-1.3 and MPI-2.1:

The editors and organizers of the combined documents have been:

- Richard Graham, Convener and Meeting Chair
- Jack Dongarra, Steering Committee
- Al Geist, Steering Committee
- Bill Gropp, Steering Committee
- Rainer Keller, Merge of MPI-1.3
- Andrew Lumsdaine, Steering Committee
- Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)

Pacific Northwest National Laboratory
Pratt & Whitney
San Diego Supercomputer Center
Sanders, A Lockheed-Martin Company
Sandia National Laboratories
Schlumberger
Scientific Computing Associates, Inc.
Silicon Graphics Incorporated
Sky Computers
Sun Microsystems Computer Corporation
Syracuse University
The MITRE Corporation
Thinking Machines Corporation
United States Navy
University of Colorado
University of Denver
University of Houston
University of Illinois
University of Maryland
University of Notre Dame
University of San Fransisco
University of Stuttgart Computing Center
University of Wisconsin

MPI-2 operated on a very tight budget (in reality, it had no budget when the first meeting was announced). Many institutions helped the MPI-2 effort by supporting the efforts and travel of the members of the MPI Forum. Direct support was given by NSF and DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

MPI-1.3 and MPI-2.1:

The editors and organizers of the combined documents have been:

- Richard Graham, Convener and Meeting Chair
- Jack Dongarra, Steering Committee
- Al Geist, Steering Committee
- Bill Gropp, Steering Committee
- Rainer Keller, Merge of MPI-1.3
- Andrew Lumsdaine, Steering Committee
- Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)

xx

- Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)

All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:

- Bill Gropp, Frontmatter, Introduction, and Bibliography
- Richard Graham, Point-to-Point Communications
- Adam Moody, Collective Communication
- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communications
- George Bosilca, Environmental Management
- David Solt, Process Creation and Management
- Bronis de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeff Squyres, Language Bindings
- Rolf Rabenseifner, Deprecated Functions, and Annex Change-Log
- Alexander Supalov and Denis Nagomy, Annex Language bindings

The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett	Richard Barrett
Christian Bell	Robert Blackmore	Gil Bloch	Ron Brightwell
Jeffrey Brown	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
Terry Dontje	Gabor Dozsa	Edric Ellis	Karl Feind
Edgar Gabriel	Patrick Geoffray	David Gingold	Dave Goodell
Erez Haba	Robert Harrison	Thomas Herault	Steve Hodson
Torsten Hoefler	Joshua Hursey	Yann Kalemkarian	Matthew Koop
Quincey Koziol	Sameer Kumar	Miron Livny	Kannan Narasimhan
Mark Pagel	Avneesh Pant	Steve Poole	Howard Pritchard
Craig Rasmussen	Hubert Ritzdorf	Rob Ross	Tony Skjellum
Brian Smith	Vinod Tipparaju	Jesper Larsson Träff	Keith Underwood

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Bull

- Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)

All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:

- Bill Gropp, Frontmatter, Introduction, and Bibliography
- Richard Graham, Point-to-Point Communications
- Adam Moody, Collective Communication
- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communications
- George Bosilca, Environmental Management
- David Solt, Process Creation and Management
- Bronis de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeff Squyres, Language Bindings
- Rolf Rabenseifner, Deprecated Functions, and Annex Change-Log
- Alexander Supalov and Denis Nagomy, Annex Language bindings

The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett	Richard Barrett
Christian Bell	Robert Blackmore	Gil Bloch	Ron Brightwell
Jeffrey Brown	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
Terry Dontje	Gabor Dozsa	Edric Ellis	Karl Feind
Edgar Gabriel	Patrick Geoffray	David Gingold	Dave Goodell
Erez Haba	Robert Harrison	Thomas Herault	Steve Hodson
Torsten Hoefler	Joshua Hursey	Yann Kalemkarian	Matthew Koop
Quincey Koziol	Sameer Kumar	Miron Livny	Kannan Narasimhan
Mark Pagel	Avneesh Pant	Steve Poole	Howard Pritchard
Craig Rasmussen	Hubert Ritzdorf	Rob Ross	Tony Skjellum
Brian Smith	Vinod Tipparaju	Jesper Larsson Träff	Keith Underwood

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Bull

xxi

Cisco Systems, Inc.
Cray Incorporation
HDF Group
Hewlett-Packard
IBM T.J. Watson Research
Indiana University
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Laboratories Europe, NEC Europe Ltd.
Oak Ridge National Laboratory
Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
Sandia National Laboratories
SiCortex
Silicon Graphics Incorporation
Sun Microsystem
University of Barcelona (UAB), Dept. of Computer and Information Sciences
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Wisconsin

In addition, the HDF Group provided travel support for one U.S. academic.

Cisco Systems, Inc.
Cray Incorporation
HDF Group
Hewlett-Packard
IBM T.J. Watson Research
Indiana University
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Laboratories Europe, NEC Europe Ltd.
Oak Ridge National Laboratory
Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
Sandia National Laboratories
SiCortex
Silicon Graphics Incorporation
Sun Microsystem
University of Barcelona (UAB), Dept. of Computer and Information Sciences
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Wisconsin

In addition, the HDF Group provided travel support for one U.S. academic.

xxii

Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification.* All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.) MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message-passing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.

Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification.* All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.) MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message-passing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.

- Allow convenient C, C++, Fortran-77, and Fortran-95 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [38], Express [12], nCUBE's Vertex [34], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [40, 41], Chimp [16, 17], PVM [4, 14], Chameleon [25], and PICL [24].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message-Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [48]. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [15]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

2 CHAPTER 1. INTRODUCTION TO MPI

- Allow convenient C, C++, Fortran-77, and Fortran-95 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [38], Express [12], nCUBE's Vertex [34], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [40, 41], Chimp [16, 17], PVM [4, 14], Chameleon [25], and PICL [24].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message-Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [48]. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [15]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [21]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 [22] (see `http://www.mpi-forum.org` for official MPI document releases). At that time, effort focused in five areas.

1. Further corrections and clarifications for the MPI-1.1 document.
2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).
3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."
4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.
5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g. zero-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) were collected in Chapter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the above list) are in the remaining chapters of the MPI-2 document, and constitute the specification for MPI-2. Items of type 5 in the above list have been moved to a separate document, the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard.

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.
- MPI-2 compliance will mean compliance with all of MPI-2.1.
- The MPI Journal of Development is not part of the MPI Standard.

It is to be emphasized that forward compatibility is preserved. That is, a valid MPI-1.1 program is both a valid MPI-1.3 program and a valid MPI-2.1 program, and a valid MPI-1.3 program is a valid MPI-2.1 program.

1.4 Background of MPI-1.3 and MPI-2.1

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1" was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for

1.3. BACKGROUND OF MPI-1.1, MPI-1.2, AND MPI-2.0 3

1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [21]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 [22] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

1. Further corrections and clarifications for the MPI-1.1 document.
2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).
3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."
4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.
5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g. zero-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) were collected in Chapter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the above list) are in the remaining chapters of the MPI-2 document, and constitute the specification for MPI-2. Items of type 5 in the above list have been moved to a separate document, the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard.

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.
- MPI-2 compliance will mean compliance with all of MPI-2.1.
- The MPI Journal of Development is not part of the MPI Standard.

It is to be emphasized that forward compatibility is preserved. That is, a valid MPI-1.1 program is both a valid MPI-1.3 program and a valid MPI-2.1 program, and a valid MPI-1.3 program is a valid MPI-2.1 program.

1.4 Background of MPI-1.3 and MPI-2.1

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1" was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for

MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done electronically. Both ballots were combined into one document: “Errata for MPI-2”, May 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors kept working on new requests for clarification.

Restarting regular work of the MPI Forum was initiated in three meetings, at EuroPVM/MPI’06 in Bonn, at EuroPVM/MPI’07 in Paris, and at SC’07 in Reno. In December 2007, a steering committee started the organization of new MPI Forum meetings at regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum decided to combine the existing and future MPI documents to one single document for each version of the MPI standard. For technical and historical reasons, this series was started with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, for each chapter, a chapter author and review team were defined. They cleaned up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document was finished in June 2008, and finally released with a second vote in September 2008 in the meeting at Dublin, just before EuroPVM/MPI’08. The major work of the current MPI Forum is the preparation of MPI-3.

1.5 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran, C and C++. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

1.6 What Platforms Are Targets For Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those “machines” consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations.

4 CHAPTER 1. INTRODUCTION TO MPI

MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done electronically. Both ballots were combined into one document: “Errata for MPI-2”, May 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors kept working on new requests for clarification.

Restarting regular work of the MPI Forum was initiated in three meetings, at EuroPVM/MPI’06 in Bonn, at EuroPVM/MPI’07 in Paris, and at SC’07 in Reno. In December 2007, a steering committee started the organization of new MPI Forum meetings at regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum decided to combine the existing and future MPI documents to one single document for each version of the MPI standard. For technical and historical reasons, this series was started with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, for each chapter, a chapter author and review team were defined. They cleaned up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document was finished in June 2008, and finally released with a second vote in September 2008 in the meeting at Dublin, just before EuroPVM/MPI’08. The major work of the current MPI Forum is the preparation of MPI-3.

1.5 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran, C and C++. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

1.6 What Platforms Are Targets For Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those “machines” consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations.

1.7 What Is Included In The Standard?

The standard includes:

- Point-to-point communication
- Datatypes
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Environmental Management and inquiry
- The info object
- Process creation and management
- One-sided communication
- External interfaces
- Parallel file I/O
- Language Bindings for Fortran, C and C++
- Profiling interface

1.8 What Is Not Included In The Standard?

The standard does not specify:

- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
- Program construction tools,
- Debugging facilities.

There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self-imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.

1.7. WHAT IS INCLUDED IN THE STANDARD? 5

1.7 What Is Included In The Standard?

The standard includes:

- Point-to-point communication
- Datatypes
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Environmental Management and inquiry
- The info object
- Process creation and management
- One-sided communication
- External interfaces
- Parallel file I/O
- Language Bindings for Fortran, C and C++
- Profiling interface

1.8 What Is Not Included In The Standard?

The standard does not specify:

- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
- Program construction tools,
- Debugging facilities.

There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self-imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.

1.9 Organization of this Document

The following is a list of the remaining chapters in this document, along with a brief description of each.

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point to Point Communication, defines the basic, pairwise communication subset of MPI. *Send* and *receive* are found here, along with many associated functions designed to make basic communication powerful and efficient.
- Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
- Chapter 5, Collective Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations.
- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.

- Chapter 9, The Info Object, defines an opaque object, that is used as input of several MPI routines.
- Chapter 10, Process Creation and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.

6 CHAPTER 1. INTRODUCTION TO MPI

1.9 Organization of this Document

The following is a list of the remaining chapters in this document, along with a brief description of each.

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point to Point Communication, defines the basic, pairwise communication subset of MPI. *Send* and *receive* are found here, along with many associated functions designed to make basic communication powerful and efficient.
- Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
- Chapter 5, Collective Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations.
- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.

- Chapter 9, The Info Object, defines an opaque object, that is used as input of several MPI routines.
- Chapter 10, Process Creation and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.

- Chapter 14, Profiling Interface, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.
- Chapter 15, Deprecated Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, Language Bindings, describes the C++ binding, discusses Fortran issues, and describes language interoperability aspects between C, C++, and Fortran.

The Appendices are:

- Annex A, Language Bindings Summary, gives specific syntax in C, C++, and Fortran, for all MPI functions, constants, and types.
- Annex B, Change-Log, summarizes major changes since the previous version of the standard.
- Several Index pages are showing the locations of examples, constants and predefined handles, callback routines' prototypes, and all MPI functions.

MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual binding of these interfaces that will enable interoperability is outside the scope of this document.

A separate document consists of ideas that were discussed in the MPI Forum and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are

- Chapter 2, Spawning Independent Processes, includes some elements of dynamic process management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
- Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
- Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.
- Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
- Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.

1.9. ORGANIZATION OF THIS DOCUMENT 7

- Chapter 14, Profiling Interface, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.
- Chapter 15, Deprecated Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 16, Language Bindings, describes the C++ binding, discusses Fortran issues, and describes language interoperability aspects between C, C++, and Fortran.

The Appendices are:

- Annex A, Language Bindings Summary, gives specific syntax in C, C++, and Fortran, for all MPI functions, constants, and types.
- Annex B, Change-Log, summarizes major changes since the previous version of the standard.
- Several Index pages are showing the locations of examples, constants and predefined handles, callback routines' prototypes, and all MPI functions.

MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual binding of these interfaces that will enable interoperability is outside the scope of this document.

A separate document consists of ideas that were discussed in the MPI Forum and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are

- Chapter 2, Spawning Independent Processes, includes some elements of dynamic process management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
- Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
- Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.
- Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
- Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.

- Chapter 7, Split Collective Communication, describes a specification for certain non-blocking collective operations.
- Chapter 8, Real-Time MPI, discusses MPI support for real time processing.

8 CHAPTER 1. INTRODUCTION TO MPI

- Chapter 7, Split Collective Communication, describes a specification for certain non-blocking collective operations.
- Chapter 8, Real-Time MPI, discusses MPI support for real time processing.

Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices. It is similar to the MPI-1 Terms and Conventions chapter but differs in some major and minor ways. Some of the major areas of difference are the naming conventions, some semantic definitions, file objects, Fortran 90 *vs* Fortran 77, C++, processes, and interaction with signals.

2.1 Document Notation

Rationale. Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (*End of advice to implementors.*)

2.2 Naming Conventions

In many cases MPI names for C functions are of the form Class_action_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules. The C++ bindings in particular follow these rules (see Section 2.6.4 on page 18).

1. In C, all routines associated with a particular type of MPI object should be of the form Class_action_subset or, if no subset exists, of the form Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form CLASS_ACTION_SUBSET or, if no subset exists, of the form CLASS_ACTION. For C

Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices. It is similar to the MPI-1 Terms and Conventions chapter but differs in some major and minor ways. Some of the major areas of difference are the naming conventions, some semantic definitions, file objects, Fortran 90 *vs* Fortran 77, C++, processes, and interaction with signals.

2.1 Document Notation

Rationale. Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (*End of advice to implementors.*)

2.2 Naming Conventions

In many cases MPI names for C functions are of the form Class_action_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules. The C++ bindings in particular follow these rules (see Section 2.6.4 on page 18).

1. In C, all routines associated with a particular type of MPI object should be of the form Class_action_subset or, if no subset exists, of the form Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form CLASS_ACTION_SUBSET or, if no subset exists, of the form CLASS_ACTION. For C

9

and Fortran we use the C++ terminology to define the Class. In C++, the routine is a method on **Class** and is named **MPI::Class::Action_subset**. If the routine is associated with a certain class, but does not make sense as an object method, it is a static member function of the class.

2. If the routine is not associated with a class, the name should be of the form Action_subset in C and ACTION_SUBSET in Fortran, and in C++ should be scoped in the **MPI** namespace, **MPI::Action_subset**.

3. The names of certain actions have been standardized. In particular, **Create** creates a new object, **Get** retrieves information about an object, **Set** sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the **Class** name from the routine and the omission of the **Action** where one can be inferred.

MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.

2.3 Procedure Specification

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT or INOUT. The meanings of these are:

- IN: the call may use the input value but does not update the argument,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.5.1), and the object is updated by the procedure call, then the argument is marked INOUT or OUT. It is marked this way even though the handle itself is not modified — we use the INOUT or OUT attribute to denote that what the handle *references* is updated. Thus, in C++, IN arguments are usually either references or pointers to `const` objects.

Rationale. The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (*End of rationale.*)

MPI's use of IN, OUT and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., `INTENT` in Fortran 90 bindings or `const` in C bindings). For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument.

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an

10 CHAPTER 2. MPI TERMS AND CONVENTIONS

and Fortran we use the C++ terminology to define the Class. In C++, the routine is a method on **Class** and is named **MPI::Class::Action_subset**. If the routine is associated with a certain class, but does not make sense as an object method, it is a static member function of the class.

2. If the routine is not associated with a class, the name should be of the form Action_subset in C and ACTION_SUBSET in Fortran, and in C++ should be scoped in the **MPI** namespace, **MPI::Action_subset**.

3. The names of certain actions have been standardized. In particular, **Create** creates a new object, **Get** retrieves information about an object, **Set** sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the **Class** name from the routine and the omission of the **Action** where one can be inferred.

MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.

2.3 Procedure Specification

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT or INOUT. The meanings of these are:

- IN: the call may use the input value but does not update the argument,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.5.1), and the object is updated by the procedure call, then the argument is marked INOUT or OUT. It is marked this way even though the handle itself is not modified — we use the INOUT or OUT attribute to denote that what the handle *references* is updated. Thus, in C++, IN arguments are usually either references or pointers to `const` objects.

Rationale. The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (*End of rationale.*)

MPI's use of IN, OUT and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., `INTENT` in Fortran 90 bindings or `const` in C bindings). For instance, the "constant" MPI_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI_STATUS_IGNORE can be passed as the OUT status argument.

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an

INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument.

Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased with any other argument passed to an MPI procedure. An example of argument aliasing in C appears below. If we define a C procedure like this,

```
void copyIntBuffer( int *pin, int *pout, int len )
{   int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}
```

then a call to it in the following code fragment has aliased arguments.

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, the ISO C version of the function is shown followed by a version of the same function in Fortran and then the C++ binding. Fortran in this document refers to Fortran 90; see Section 2.6.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

nonblocking A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is **started** by the call that initiates it, e.g., MPI_ISEND. The word complete is used with respect to operations, requests, and communications. An **operation completes** when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to MPI_TEST will return flag = true. A **request is completed** by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is **freed**, and becomes **inactive** if it was persistent. A **communication completes** when all participating operations complete.

blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.

local A procedure is local if completion of the procedure depends only on the local executing process.

non-local A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.

2.4. SEMANTIC TERMS 11

INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument.

Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased with any other argument passed to an MPI procedure. An example of argument aliasing in C appears below. If we define a C procedure like this,

void copyIntBuffer(int *pin, int *pout, int len)
{   int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}

then a call to it in the following code fragment has aliased arguments.

int a[10];
copyIntBuffer(a, a+3, 7);

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, the ISO C version of the function is shown followed by a version of the same function in Fortran and then the C++ binding. Fortran in this document refers to Fortran 90; see Section 2.6.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

nonblocking A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is **started** by the call that initiates it, e.g., MPI_ISEND. The word complete is used with respect to operations, requests, and communications. An **operation completes** when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to MPI_TEST will return flag = true. A **request is completed** by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is **freed**, and becomes **inactive** if it was persistent. A **communication completes** when all participating operations complete.

blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.

local A procedure is local if completion of the procedure depends only on the local executing process.

non-local A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.

collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group.

predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_UB) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are **named** whereas the latter are **unnamed**.

derived A derived datatype is any datatype that is not predefined.

portable A datatype is portable, if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture.

equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

2.5 Data Types

2.5.1 Opaque Objects

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran, all handles have type `INTEGER`. In C and C++, a different handle type is defined for each category of objects. In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

12 CHAPTER 2. MPI TERMS AND CONVENTIONS

collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group.

predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_UB) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are **named** whereas the latter are **unnamed**.

derived A derived datatype is any datatype that is not predefined.

portable A datatype is portable, if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture.

equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

2.5 Data Types

2.5.1 Opaque Objects

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran, all handles have type `INTEGER`. In C and C++, a different handle type is defined for each category of objects. In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply "wrap up" a table index or pointer.

(*End of advice to implementors.*)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an "invalid handle" value. MPI provides an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created and cannot be transferred to another process.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects. In C++, this is enforced by declaring the handles to these predefined objects to be `static const`.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C, C++, and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (*End of rationale.*)

Advice to users. A user may accidently create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (*End of advice to users.*)

2.5. DATA TYPES 13

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply "wrap up" a table index or pointer.

(*End of advice to implementors.*)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an "invalid handle" value. MPI provides an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created and cannot be transferred to another process.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects. In C++, this is enforced by declaring the handles to these predefined objects to be `static const`.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in C, C++, and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (*End of rationale.*)

Advice to users. A user may accidently create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (*End of advice to users.*)

Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components; a call to MPI_COMM_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE.

2.5.3 State

MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document). The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE).

The constants that cannot be used in initialization expressions or assignments in Fortran are:

```
MPI_BOTTOM
MPI_STATUS_IGNORE
```

14 CHAPTER 2. MPI TERMS AND CONVENTIONS

Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components; a call to MPI_COMM_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE.

2.5.3 State

MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document). The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE).

The constants that cannot be used in initialization expressions or assignments in Fortran are:

MPI_BOTTOM
MPI_STATUS_IGNORE

```
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE
MPI_ARGV_NULL
MPI_ARGVS_NULL
```

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through `parameter` statements) is not possible because an implementation cannot distinguish these values from legal data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared `COMMON` block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (*End of advice to implementors.*)

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses <type> to represent a choice variable; for C and C++, we use void *.

2.5.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C, MPI::Aint in C++ and `INTEGER (KIND=MPI_ADDRESS_KIND)` in Fortran. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be `INTEGER (KIND=MPI_OFFSET_KIND)` in Fortran. In C one uses MPI_Offset whereas in C++ one uses MPI::Offset.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90, though they are designed to be usable in Fortran 77 environments.

Since the word `PARAMETER` is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will

2.6. LANGUAGE BINDING 15

MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE
MPI_ARGV_NULL
MPI_ARGVS_NULL

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through `parameter` statements) is not possible because an implementation cannot distinguish these values from legal data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared `COMMON` block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (*End of advice to implementors.*)

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses <type> to represent a choice variable; for C and C++, we use void *.

2.5.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C, MPI::Aint in C++ and `INTEGER (KIND=MPI_ADDRESS_KIND)` in Fortran. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be `INTEGER (KIND=MPI_OFFSET_KIND)` in Fortran. In C one uses MPI_Offset whereas in C++ one uses MPI::Offset.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90, though they are designed to be usable in Fortran 77 environments.

Since the word `PARAMETER` is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will

understand the word "argument" (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the "mpi_" and "pmpi_" prefixes.

2.6.1 Deprecated Names and Functions

A number of chapters refer to deprecated or replaced MPI-1 constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with MPI-2. For example, the Fortran binding for MPI-1 functions that have address arguments uses INTEGER. This is not consistent with the C binding, and causes problems on machines with 32 bit INTEGERs and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated. Another example is provided by the MPI-1 predefined datatypes MPI_UB and MPI_LB. They are deprecated, since their use is awkward and error-prone. The MPI-2 function MPI_TYPE_CREATE_RESIZED provides a more convenient mechanism to achieve the same effect.

Table 2.1 shows a list of all of the deprecated constructs. Note that the constants MPI_LB and MPI_UB are replaced by the function MPI_TYPE_CREATE_RESIZED; this is because their principal use was as input datatypes to MPI_TYPE_STRUCT to create resized datatypes. Also note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran 90.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 16.3.9.

Handles are represented in Fortran as INTEGERs. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for

16 CHAPTER 2. MPI TERMS AND CONVENTIONS

understand the word "argument" (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the "mpi_" and "pmpi_" prefixes.

2.6.1 Deprecated Names and Functions

A number of chapters refer to deprecated or replaced MPI-1 constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with MPI-2. For example, the Fortran binding for MPI-1 functions that have address arguments uses INTEGER. This is not consistent with the C binding, and causes problems on machines with 32 bit INTEGERs and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated. Another example is provided by the MPI-1 predefined datatypes MPI_UB and MPI_LB. They are deprecated, since their use is awkward and error-prone. The MPI-2 function MPI_TYPE_CREATE_RESIZED provides a more convenient mechanism to achieve the same effect.

Table 2.1 shows a list of all of the deprecated constructs. Note that the constants MPI_LB and MPI_UB are replaced by the function MPI_TYPE_CREATE_RESIZED; this is because their principal use was as input datatypes to MPI_TYPE_STRUCT to create resized datatypes. Also note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran 90.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 16.3.9.

Handles are represented in Fortran as INTEGERs. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for

Deprecated	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_LB	MPI_TYPE_CREATE_RESIZED
MPI_UB	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function	MPI_Comm_errhandler_fn
MPI_KEYVAL_CREATE	MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI_COMM_FREE_KEYVAL
MPI_DUP_FN	MPI_COMM_DUP_FN
MPI_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
MPI_NULL_DELETE_FN	MPI_COMM_NULL_DELETE_FN
MPI_Copy_function	MPI_Comm_copy_attr_function
COPY_FUNCTION	COMM_COPY_ATTR_FN
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_FN
MPI_ATTR_DELETE	MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI_COMM_SET_ATTR

Table 2.1: Deprecated constructs

user codes that are discussed in detail in Section 16.2.2. They are also inconsistent with Fortran 77.

- An MPI subroutine with a choice argument may be called with different argument types.
- An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.
- Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.
- An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program executing outside MPI calls.
- Several named "constants," such as MPI_BOTTOM, MPI_STATUS_IGNORE, and MPI_ERRCODES_IGNORE, are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.

2.6. LANGUAGE BINDING 17

Deprecated	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_LB	MPI_TYPE_CREATE_RESIZED
MPI_UB	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function	MPI_Comm_errhandler_fn
MPI_KEYVAL_CREATE	MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI_COMM_FREE_KEYVAL
MPI_DUP_FN	MPI_COMM_DUP_FN
MPI_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
MPI_NULL_DELETE_FN	MPI_COMM_NULL_DELETE_FN
MPI_Copy_function	MPI_Comm_copy_attr_function
COPY_FUNCTION	COMM_COPY_ATTR_FN
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_FN
MPI_ATTR_DELETE	MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI_COMM_SET_ATTR

Table 2.1: Deprecated constructs

user codes that are discussed in detail in Section 16.2.2. They are also inconsistent with Fortran 77.

- An MPI subroutine with a choice argument may be called with different argument types.
- An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.
- Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.
- An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program executing outside MPI calls.
- Several named "constants," such as MPI_BOTTOM, MPI_STATUS_IGNORE, and MPI_ERRCODES_IGNORE, are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.

Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, `INTEGER*8` or `INTEGER` should be used instead.
- The memory allocation routine MPI_ALLOC_MEM cannot be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix `MPI_`. To support the profiling interface, programs should not declare functions with names beginning with the prefix `PMPI_`.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Almost all C functions return an error code. The successful return code will be MPI_SUCCESS, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects.

Array arguments are indexed from zero.

Logical flags are integers with value 0 meaning "false" and a non-zero value meaning "true."

Choice arguments are pointers of type `void *`.

Address arguments are of MPI defined type MPI_Aint. File displacements are of type MPI_Offset. MPI_Aint is defined to be an integer of the size needed to hold any valid address on the target architecture. MPI_Offset is defined to be an integer of the size needed to hold any valid file size on the target architecture.

2.6.4 C++ Binding Issues

There are places in the standard that give rules for C and not for C++. In these cases, the C rule should be applied to the C++ case, as appropriate. In particular, the values of constants given in the text are the ones for C and Fortran. A cross index of these with the C++ names is given in Annex A.

We use the ISO C++ declaration format. All MPI names are declared within the scope of a namespace called `MPI` and therefore are referenced with an `MPI::` prefix. Defined constants are in all capital letters, and class names, defined types, and functions have only their first letter capitalized. Programs must not declare variables or functions in the `MPI` namespace. This is mandated to avoid possible name collisions.

18 CHAPTER 2. MPI TERMS AND CONVENTIONS

Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, `INTEGER*8` or `INTEGER` should be used instead.
- The memory allocation routine MPI_ALLOC_MEM cannot be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix `MPI_`. To support the profiling interface, programs should not declare functions with names beginning with the prefix `PMPI_`.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Almost all C functions return an error code. The successful return code will be MPI_SUCCESS, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects.

Array arguments are indexed from zero.

Logical flags are integers with value 0 meaning "false" and a non-zero value meaning "true."

Choice arguments are pointers of type `void *`.

Address arguments are of MPI defined type MPI_Aint. File displacements are of type MPI_Offset. MPI_Aint is defined to be an integer of the size needed to hold any valid address on the target architecture. MPI_Offset is defined to be an integer of the size needed to hold any valid file size on the target architecture.

2.6.4 C++ Binding Issues

There are places in the standard that give rules for C and not for C++. In these cases, the C rule should be applied to the C++ case, as appropriate. In particular, the values of constants given in the text are the ones for C and Fortran. A cross index of these with the C++ names is given in Annex A.

We use the ISO C++ declaration format. All MPI names are declared within the scope of a namespace called `MPI` and therefore are referenced with an `MPI::` prefix. Defined constants are in all capital letters, and class names, defined types, and functions have only their first letter capitalized. Programs must not declare variables or functions in the `MPI` namespace. This is mandated to avoid possible name collisions.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Advice to implementors. The file mpi.h may contain both the C and C++ definitions. Usually one can simply use the defined value (generally `__cplusplus`, but not required) to see if one is using C++ to protect the C++ definitions. It is possible that a C compiler will require that the source protected this way be legal C code. In this case, all the C++ definitions can be placed in a different include file and the "`#include`" directive can be used to include the necessary C++ definitions in the mpi.h file. (*End of advice to implementors.*)

C++ functions that create objects or return information usually place the object or information in the return value. Since the language neutral prototypes of MPI functions include the C++ return value as an OUT parameter, semantic descriptions of MPI functions refer to the C++ return value by that parameter name. The remaining C++ functions return `void`.

In some circumstances, MPI permits users to indicate that they do not want a return value. For example, the user may indicate that the status is not filled in. Unlike C and Fortran where this is achieved through a special input value, in C++ this is done by having two bindings where one has the optional argument and one does not.

C++ functions do not return error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an `MPI::Exception` object.

It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a given type has not changed. User error handlers are also permitted. MPI::ERRORS_RETURN simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

Advice to users. C++ programmers that want to handle MPI errors on their own should use the MPI::ERRORS_THROW_EXCEPTIONS error handler, rather than MPI::ERRORS_RETURN, that is used for that purpose in C. Care should be taken using exceptions in mixed language situations. (*End of advice to users.*)

Opaque object handles must be objects in themselves, and have the assignment and equality operators overridden to perform semantically like their C and Fortran counterparts.

Array arguments are indexed from zero.

Logical flags are of type `bool`.

Choice arguments are pointers of type `void *`.

Address arguments are of MPI-defined integer type MPI::Aint, defined to be an integer of the size needed to hold any valid address on the target architecture. Analogously, MPI::Offset is an integer to hold file offsets.

Most MPI functions are methods of MPI C++ classes. MPI class names are generated from the language neutral MPI types by dropping the `MPI_` prefix and scoping the type within the `MPI` namespace. For example, MPI_DATATYPE becomes `MPI::Datatype`.

The names of MPI functions generally follow the naming rules given. In some circumstances, the MPI function is related to a function defined already for MPI-1 with a name

2.6. LANGUAGE BINDING 19

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.h.

Advice to implementors. The file mpi.h may contain both the C and C++ definitions. Usually one can simply use the defined value (generally `__cplusplus`, but not required) to see if one is using C++ to protect the C++ definitions. It is possible that a C compiler will require that the source protected this way be legal C code. In this case, all the C++ definitions can be placed in a different include file and the "`#include`" directive can be used to include the necessary C++ definitions in the mpi.h file. (*End of advice to implementors.*)

C++ functions that create objects or return information usually place the object or information in the return value. Since the language neutral prototypes of MPI functions include the C++ return value as an OUT parameter, semantic descriptions of MPI functions refer to the C++ return value by that parameter name. The remaining C++ functions return `void`.

In some circumstances, MPI permits users to indicate that they do not want a return value. For example, the user may indicate that the status is not filled in. Unlike C and Fortran where this is achieved through a special input value, in C++ this is done by having two bindings where one has the optional argument and one does not.

C++ functions do not return error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an `MPI::Exception` object.

It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a given type has not changed. User error handlers are also permitted. MPI::ERRORS_RETURN simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

Advice to users. C++ programmers that want to handle MPI errors on their own should use the MPI::ERRORS_THROW_EXCEPTIONS error handler, rather than MPI::ERRORS_RETURN, that is used for that purpose in C. Care should be taken using exceptions in mixed language situations. (*End of advice to users.*)

Opaque object handles must be objects in themselves, and have the assignment and equality operators overridden to perform semantically like their C and Fortran counterparts.

Array arguments are indexed from zero.

Logical flags are of type `bool`.

Choice arguments are pointers of type `void *`.

Address arguments are of MPI-defined integer type MPI::Aint, defined to be an integer of the size needed to hold any valid address on the target architecture. Analogously, MPI::Offset is an integer to hold file offsets.

Most MPI functions are methods of MPI C++ classes. MPI class names are generated from the language neutral MPI types by dropping the `MPI_` prefix and scoping the type within the `MPI` namespace. For example, MPI_DATATYPE becomes `MPI::Datatype`.

The names of MPI functions generally follow the naming rules given. In some circumstances, the MPI function is related to a function defined already for MPI-1 with a name

that does not follow the naming conventions. In this circumstance, the language neutral name is in analogy to the MPI name even though this gives an MPI-2 name that violates the naming conventions. The C and Fortran names are the same as the language neutral name in this case. However, the C++ names do reflect the naming rules and can differ from the C and Fortran names. Thus, the analogous name in C++ to the MPI name may be different than the language neutral name. This results in the C++ name differing from the language neutral name. An example of this is the language neutral name of MPI_FINALIZED and a C++ name of MPI::Is_finalized.

In C++, function `typedef`s are made publicly within appropriate classes. However, these declarations then become somewhat cumbersome, as with the following:
`typedef MPI::Grequest::Query_function();`

would look like the following:

```
namespace MPI {
  class Request {
    // ...
  };

  class Grequest : public MPI::Request {
    // ...
    typedef Query_function(void* extra_state, MPI::Status& status);
  };
};
```

Rather than including this scaffolding when declaring C++ `typedef`s, we use an abbreviated form. In particular, we explicitly indicate the class and namespace scope for the `typedef` of the function. Thus, the example above is shown in the text as follows:

```
typedef int MPI::Grequest::Query_function(void* extra_state,
                                          MPI::Status& status)
```

The C++ bindings presented in Annex A.4 and throughout this document were generated by applying a simple set of name generation rules to the MPI function specifications. While these guidelines may be sufficient in most cases, they may not be suitable for all situations. In cases of ambiguity or where a specific semantic statement is desired, these guidelines may be superseded as the situation dictates.

1. All functions, types, and constants are declared within the scope of a `namespace` called `MPI`.

2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).

3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "`MPI_`" prefix and without the object name prefix (if applicable). In addition:

20 CHAPTER 2. MPI TERMS AND CONVENTIONS

that does not follow the naming conventions. In this circumstance, the language neutral name is in analogy to the MPI name even though this gives an MPI-2 name that violates the naming conventions. The C and Fortran names are the same as the language neutral name in this case. However, the C++ names do reflect the naming rules and can differ from the C and Fortran names. Thus, the analogous name in C++ to the MPI name may be different than the language neutral name. This results in the C++ name differing from the language neutral name. An example of this is the language neutral name of MPI_FINALIZED and a C++ name of MPI::Is_finalized.

In C++, function `typedef`s are made publicly within appropriate classes. However, these declarations then become somewhat cumbersome, as with the following:
`typedef MPI::Grequest::Query_function();`

would look like the following:

```
namespace MPI {
  class Request {
    // ...
  };

  class Grequest : public MPI::Request {
    // ...
    typedef Query_function(void* extra_state, MPI::Status& status);
  };
};
```

Rather than including this scaffolding when declaring C++ `typedef`s, we use an abbreviated form. In particular, we explicitly indicate the class and namespace scope for the `typedef` of the function. Thus, the example above is shown in the text as follows:

```
typedef int MPI::Grequest::Query_function(void* extra_state,
                                          MPI::Status& status)
```

The C++ bindings presented in Annex A.4 and throughout this document were generated by applying a simple set of name generation rules to the MPI function specifications. While these guidelines may be sufficient in most cases, they may not be suitable for all situations. In cases of ambiguity or where a specific semantic statement is desired, these guidelines may be superseded as the situation dictates.

1. All functions, types, and constants are declared within the scope of a `namespace` called `MPI`.

2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).

3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "`MPI_`" prefix and without the object name prefix (if applicable). In addition:

   (a) The scalar IN handle is dropped from the argument list, and `this` corresponds to the dropped argument.
   (b) The function is declared `const`.

4. MPI functions are made into class functions (static) when they belong on a class but do not have a unique scalar IN or INOUT parameter of that class.
5. If the argument list contains a single OUT argument that is not of type MPI_STATUS (or an array), that argument is dropped from the list and the function returns that value.

   Example 2.1 The C++ binding for MPI_COMM_SIZE is
   int MPI::Comm::Get_size(void) const.

6. If there are multiple OUT arguments in the argument list, one is chosen as the return value and is removed from the list.
7. If the argument list does not contain any OUT arguments, the function returns `void`.

   Example 2.2 The C++ binding for MPI_REQUEST_FREE is
   void MPI::Request::Free(void)

8. MPI functions to which the above rules do not apply are not members of any class, but are defined in the `MPI` namespace.

   Example 2.3 The C++ binding for MPI_BUFFER_ATTACH is
   void MPI::Attach_buffer(void* buffer, int size).

9. All class names, defined types, and function names have only their first letter capitalized. Defined constants are in all capital letters.
10. Any IN pointer, reference, or array argument must be declared `const`.
11. Handles are passed by reference.
12. Array arguments are denoted with square brackets (`[]`), not pointers, as this is more semantically precise.

2.6.5 Functions and Macros

An implementation is allowed to implement MPI_WTIME, MPI_WTICK, PMPI_WTIME, PMPI_WTICK, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 16.3.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (*End of advice to implementors.*)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (*End of advice to users.*)

2.6. LANGUAGE BINDING 21

   (a) The scalar IN handle is dropped from the argument list, and `this` corresponds to the dropped argument.
   (b) The function is declared `const`.

4. MPI functions are made into class functions (static) when they belong on a class but do not have a unique scalar IN or INOUT parameter of that class.
5. If the argument list contains a single OUT argument that is not of type MPI_STATUS (or an array), that argument is dropped from the list and the function returns that value.

   Example 2.1 The C++ binding for MPI_COMM_SIZE is
   int MPI::Comm::Get_size(void) const.

6. If there are multiple OUT arguments in the argument list, one is chosen as the return value and is removed from the list.
7. If the argument list does not contain any OUT arguments, the function returns `void`.

   Example 2.2 The C++ binding for MPI_REQUEST_FREE is
   void MPI::Request::Free(void)

8. MPI functions to which the above rules do not apply are not members of any class, but are defined in the `MPI` namespace.

   Example 2.3 The C++ binding for MPI_BUFFER_ATTACH is
   void MPI::Attach_buffer(void* buffer, int size).

9. All class names, defined types, and function names have only their first letter capitalized. Defined constants are in all capital letters.
10. Any IN pointer, reference, or array argument must be declared `const`.
11. Handles are passed by reference.
12. Array arguments are denoted with square brackets (`[]`), not pointers, as this is more semantically precise.

2.6.5 Functions and Macros

An implementation is allowed to implement MPI_WTIME, MPI_WTICK, PMPI_WTIME, PMPI_WTICK, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 16.3.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (*End of advice to implementors.*)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (*End of advice to users.*)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in any implementation. In addition, a **resource error** may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself

22 CHAPTER 2. MPI TERMS AND CONVENTIONS

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in any implementation. In addition, a **resource error** may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself

or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3. The return values of C++ functions are not error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an `MPI::Exception` object. See also Section 16.1.8 on page 457.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller in a consistent state.

Another subtle issue arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode). Such an error must be treated as fatal, since information cannot be returned for the user to recover from it.

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as `write` in Fortran and `printf` and `malloc` in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

2.9. IMPLEMENTATION ISSUES 23

or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3. The return values of C++ functions are not error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an `MPI::Exception` object. See also Section 16.1.8 on page 457.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller in a consistent state.

Another subtle issue arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode). Such an error must be treated as fatal, since information cannot be returned for the user to recover from it.

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as `write` in Fortran and `printf` and `malloc` in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that `printf` is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran and C++ programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

In addition, calls that fail because of resource exhaustion or other error are not considered a violation of the requirements here (however, they are required to complete, just not to complete successfully).

2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended that it not use `SIGALRM`, `SIGFPE`, or `SIGIO`. Implementations may also prohibit the use of MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

24 CHAPTER 2. MPI TERMS AND CONVENTIONS

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that `printf` is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran and C++ programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

In addition, calls that fail because of resource exhaustion or other error are not considered a violation of the requirements here (however, they are required to complete, just not to complete successfully).

2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended that it not use `SIGALRM`, `SIGFPE`, or `SIGIO`. Implementations may also prohibit the use of MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

Chapter 3

Point-to-Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are **send** and **receive**. Their use is illustrated in the example below.

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1)  /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

In this example, process zero (myrank = 0) sends a message to process one using the **send** operation MPI_SEND. The operation specifies a **send buffer** in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition,

Chapter 3

Point-to-Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are **send** and **receive**. Their use is illustrated in the example below.

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1)  /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

In this example, process zero (myrank = 0) sends a message to process one using the **send** operation MPI_SEND. The operation specifies a **send buffer** in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition,

25

the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive** operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (myrank = 1) receives this message with the **receive** operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations, Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations, ending with a description of the "dummy" process, MPI_PROC_NULL.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Send(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

The blocking semantics of this call are described in Section 3.4.

26 CHAPTER 3. POINT-TO-POINT COMMUNICATION

the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive** operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (myrank = 1) receives this message with the **receive** operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations, Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations, ending with a description of the "dummy" process, MPI_PROC_NULL.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (nonnegative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Comm::Send(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const

The blocking semantics of this call are described in Section 3.4.

3.2.2 Message Data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of count values, each of the type indicated by datatype. count may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 4.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

3.2. BLOCKING SEND AND RECEIVE OPERATIONS 27

3.2.2 Message Data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of count values, each of the type indicated by datatype. count may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1.

MPI datatype | Fortran datatype
MPI_INTEGER | INTEGER
MPI_REAL | REAL
MPI_DOUBLE_PRECISION | DOUBLE PRECISION
MPI_COMPLEX | COMPLEX
MPI_LOGICAL | LOGICAL
MPI_CHARACTER | CHARACTER(1)
MPI_BYTE |
MPI_PACKED |

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 4.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

MPI datatype	C datatype
MPI_CHAR	`signed char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_BYTE	
MPI_PACKED	

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

3.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

source
destination
tag
communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The comm argument specifies the **communicator** that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

28 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI datatype	C datatype
MPI_CHAR	`signed char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_BYTE	
MPI_PACKED	

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

3.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

source
destination
tag
communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The comm argument specifies the **communicator** that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (*End of advice to users.*)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (*End of advice to implementors.*)

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

3.2. BLOCKING SEND AND RECEIVE OPERATIONS 29

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (*End of advice to users.*)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (*End of advice to implementors.*)

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT buf initial address of receive buffer (choice)
IN count number of elements in receive buffer (non-negative integer)
IN datatype datatype of each receive buffer element (handle)
IN source rank of source (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag, MPI::Status& status) const
```

```
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag) const
```

The blocking semantics of this call are described in Section 3.4.

The receive buffer consists of the storage containing count consecutive elements of the type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The MPI_PROBE function described in Section 3.8 can be used to receive messages of unknown length. (*End of advice to users.*)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI_ANY_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is {0,...,n-1}∪{MPI_ANY_SOURCE}, where n is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify

30 CHAPTER 3. POINT-TO-POINT COMMUNICATION

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag, MPI::Status& status) const
```

```
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag) const
```

The blocking semantics of this call are described in Section 3.4.

The receive buffer consists of the storage containing count consecutive elements of the type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The MPI_PROBE function described in Section 3.8 can be used to receive messages of unknown length. (*End of advice to users.*)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI_ANY_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is {0,...,n-1}∪{MPI_ANY_SOURCE}, where n is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify

a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran, status is an array of `INTEGER`s of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

In C++, the status object is handled through the following methods:

```
int MPI::Status::Get_source() const

void MPI::Status::Set_source(int source)

int MPI::Status::Get_tag() const

void MPI::Status::Set_tag(int tag)

int MPI::Status::Get_error() const

void MPI::Status::Set_error(int error)
```

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such

3.2. BLOCKING SEND AND RECEIVE OPERATIONS 31

a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran, status is an array of `INTEGER`s of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

In C++, the status object is handled through the following methods:

```
int MPI::Status::Get_source() const

void MPI::Status::Set_source(int source)

int MPI::Status::Get_tag() const

void MPI::Status::Set_tag(int tag)

int MPI::Status::Get_error() const

void MPI::Status::Set_error(int error)
```

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such

cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to “decode” this information.

MPI_GET_COUNT(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype of each receive buffer entry (handle)
OUT	count	number of received entries (integer)

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

```
int MPI::Status::Get_count(const MPI::Datatype& datatype) const
```

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable. (We shall later see, in Section 4.1.11, that MPI_GET_COUNT may return, in certain situations, the value MPI_UNDEFINED.)

Rationale. Some message-passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the “last message received.” This is not thread safe.

The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE or MPI_IPROBE. With a status from MPI_PROBE or MPI_IPROBE, the same datatypes are allowed as in a call to MPI_RECV to receive this message. (*End of rationale.*)

The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transfered is greater than zero, MPI_UNDEFINED is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (*End of rationale.*)

32 CHAPTER 3. POINT-TO-POINT COMMUNICATION

cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to “decode” this information.

MPI_GET_COUNT(status, datatype, count)

IN status return status of receive operation (Status)
IN datatype datatype of each receive buffer entry (handle)
OUT count number of received entries (integer)

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

int MPI::Status::Get_count(const MPI::Datatype& datatype) const

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable. (We shall later see, in Section 4.1.11, that MPI_GET_COUNT may return, in certain situations, the value MPI_UNDEFINED.)

Rationale. Some message-passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the “last message received.” This is not thread safe.

The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE or MPI_IPROBE. With a status from MPI_PROBE or MPI_IPROBE, the same datatypes are allowed as in a call to MPI_RECV to receive this message. (*End of rationale.*)

The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transfered is greater than zero, MPI_UNDEFINED is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (*End of rationale.*)

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI_GET_COUNT and the receive. (*End of advice to users.*)

All send and receive operations use the buf, count, datatype, source, dest, tag, comm and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

3.2.6 Passing MPI_STATUS_IGNORE for Status

Every call to MPI_RECV includes a status argument, wherein the system can return details about the message received. There are also a number of other MPI calls where status is returned. An object of type MPI_STATUS is not an MPI opaque object; its structure is declared in `mpi.h` and `mpif.h`, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, which when passed to a receive, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_STATUS.

MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for MPI_{TEST|WAIT}{ALL|SOME} functions set to MPI_STATUS_IGNORE; one either specifies ignoring *all* of the statuses in such a call with MPI_STATUSES_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses.

There are no C++ bindings for MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE. To allow an OUT or INOUT `MPI::Status` argument to be ignored, all MPI C++ bindings that have OUT or INOUT `MPI::Status` parameters are overloaded with a second version that omits the OUT or INOUT `MPI::Status` parameter.

Example 3.1 The C++ bindings for MPI_PROBE are:

3.2. BLOCKING SEND AND RECEIVE OPERATIONS 33

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI_GET_COUNT and the receive. (*End of advice to users.*)

All send and receive operations use the buf, count, datatype, source, dest, tag, comm and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

3.2.6 Passing MPI_STATUS_IGNORE for Status

Every call to MPI_RECV includes a status argument, wherein the system can return details about the message received. There are also a number of other MPI calls where status is returned. An object of type MPI_STATUS is not an MPI opaque object; its structure is declared in `mpi.h` and `mpif.h`, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, which when passed to a receive, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_STATUS.

MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for MPI_{TEST|WAIT}{ALL|SOME} functions set to MPI_STATUS_IGNORE; one either specifies ignoring *all* of the statuses in such a call with MPI_STATUSES_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses.

There are no C++ bindings for MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE. To allow an OUT or INOUT `MPI::Status` argument to be ignored, all MPI C++ bindings that have OUT or INOUT `MPI::Status` parameters are overloaded with a second version that omits the OUT or INOUT `MPI::Status` parameter.

Example 3.1 The C++ bindings for MPI_PROBE are:

```
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const
```

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Section 4.2, the type MPI_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI_INTEGER matches a Fortran variable of type `INTEGER`. A table giving this correspondence for Fortran and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI_BYTE allows one to transfer the binary value of a byte in memory unchanged.

To summarize, the type matching rules fall into the three categories below.

- Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI_PACKED is used.

34 CHAPTER 3. POINT-TO-POINT COMMUNICATION

void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Section 4.2, the type MPI_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI_BYTE or MPI_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI_BYTE allows one to transfer the binary value of a byte in memory unchanged.

To summarize, the type matching rules fall into the three categories below.

- Communication of typed values (e.g., with datatype different from MPI_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender and receiver use the datatype MPI_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI_PACKED is used.

The following examples illustrate the first two cases.

Example 3.2 Sender and receiver specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both a and b are real arrays of size ≥ 10. (In Fortran, it might be correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced to an array with ten reals.)

Example 3.3 Sender and receiver do not specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is erroneous, since sender and receiver do not provide matching datatype arguments.

Example 3.4 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of a and b (unless this results in an out of bound memory access).

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

3.3. DATA TYPE MATCHING AND DATA CONVERSION 35

The following examples illustrate the first two cases.

Example 3.2 Sender and receiver specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both a and b are real arrays of size ≥ 10. (In Fortran, it might be correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced to an array with ten reals.)

Example 3.3 Sender and receiver do not specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is erroneous, since sender and receiver do not provide matching datatype arguments.

Example 3.4 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of a and b (unless this results in an out of bound memory access).

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Type MPI_CHARACTER

The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather then the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 3.5 Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

The last five characters of string b at process 1 are replaced by the first five characters of string a at process 0.

> *Rationale.* The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems.
>
> A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 4.1). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.
>
> This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*)

> *Advice to implementors.* Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (*End of advice to implementors.*)

3.3.2 Data Conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.

36 CHAPTER 3. POINT-TO-POINT COMMUNICATION

Type MPI_CHARACTER

The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather then the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 3.5 Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

The last five characters of string b at process 1 are replaced by the first five characters of string a at process 0.

Rationale. The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems.

A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 4.1). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*)

Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (*End of advice to implementors.*)

3.3.2 Data Conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.

representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system.

Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 3.2–3.4. The first program is correct, assuming that a and b are REAL arrays of size ≥ 10. If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

The second program is erroneous, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If a and b are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

3.3. DATA TYPE MATCHING AND DATA CONVERSION 37

representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system.

Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type MPI_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type MPI_CHARACTER or MPI_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 3.2–3.4. The first program is correct, assuming that a and b are REAL arrays of size ≥ 10. If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

The second program is erroneous, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If a and b are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 16.3 on page 478.

3.4 Communication Modes

The send call described in Section 3.2.1 is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

The send call described in Section 3.2.1 uses the **standard** communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

Rationale. The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the user-provided buffer system of Section 3.6 should be used, along with the buffered-mode send. (*End of rationale.*)

There are three additional communication modes.

A **buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered mode to be effective.

38 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 16.3 on page 478.

3.4 Communication Modes

The send call described in Section 3.2.1 is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

The send call described in Section 3.2.1 uses the **standard** communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

Rationale. The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the user-provided buffer system of Section 3.6 should be used, along with the buffered-mode send. (*End of rationale.*)

There are three additional communication modes.

A **buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered mode to be effective.

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**.

A send that uses the **ready** communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Bsend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Send in buffered mode.

3.4. COMMUNICATION MODES 39

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**.

A send that uses the **ready** communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Three additional send functions are provided for the three additional communication modes. The communication mode is indicated by a one letter prefix: B for buffered, S for synchronous, and R for ready.

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Bsend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Send in buffered mode.

MPI_SSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Ssend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Send in synchronous mode.

MPI_RSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Rsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is **blocking**: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

40 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_SSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Ssend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Send in synchronous mode.

MPI_RSEND (buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Rsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive operation described in the last section is **blocking**: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to access or modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

> *Rationale.* We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems — consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. (*End of rationale.*)

> *Advice to implementors.* Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.
>
> It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.
>
> A possible communication protocol for the various communication modes is outlined below.
>
> ready send: The message is sent as soon as possible.
>
> synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.
>
> standard send: First protocol may be used for short messages, and second protocol for long messages.
>
> buffered send: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).
>
> Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.
>
> Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.
>
> A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.
>
> In a multi-threaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (*End of advice to implementors.*)

3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

3.5. SEMANTICS OF POINT-TO-POINT COMMUNICATION 41

In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to access or modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Rationale. We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems — consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. (*End of rationale.*)

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

ready send: The message is sent as soon as possible.

synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

standard send: First protocol may be used for short messages, and second protocol for long messages.

buffered send: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).

Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.

Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.

In a multi-threaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (*End of advice to implementors.*)

3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Example 3.6 An example of non-overtaking messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

Progress If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Example 3.7 An example of two, intertwined matching pairs.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

42 CHAPTER 3. POINT-TO-POINT COMMUNICATION

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Example 3.6 An example of non-overtaking messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

Progress If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Example 3.7 An example of two, intertwined matching pairs.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 3.6.

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the producer will be automatically throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

Example 3.8 An exchange of messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
```

3.5. SEMANTICS OF POINT-TO-POINT COMMUNICATION 43

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 3.6.

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the producer will be automatically throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

Example 3.8 An exchange of messages.

CALL MPI_COMM_RANK(comm, rank, ierr)

```
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 3.9 An errant attempt to exchange messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

Example 3.10 An exchange that relies on buffering.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least `count` words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

44 CHAPTER 3. POINT-TO-POINT COMMUNICATION

```
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

Example 3.9 An errant attempt to exchange messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

Example 3.10 An exchange that relies on buffering.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least `count` words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is “safe” if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the “unsafe” programming style shown in example 3.10. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that “common practice” programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

3.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

MPI_BUFFER_ATTACH(buffer, size)

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (non-negative integer)

```
int MPI_Buffer_attach(void* buffer, int size)
```

```
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
```

```
void MPI::Attach_buffer(void* buffer, int size)
```

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

MPI_BUFFER_DETACH(buffer_addr, size)

OUT	buffer_addr	initial buffer address (choice)
OUT	size	buffer size, in bytes (non-negative integer)

```
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

3.6. BUFFER ALLOCATION AND USAGE 45

A program is “safe” if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the “unsafe” programming style shown in example 3.10. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that “common practice” programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

3.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

MPI_BUFFER_ATTACH(buffer, size)

IN buffer initial buffer address (choice)
IN size buffer size, in bytes (non-negative integer)

int MPI_Buffer_attach(void* buffer, int size)

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

void MPI::Attach_buffer(void* buffer, int size)

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

MPI_BUFFER_DETACH(buffer_addr, size)

OUT buffer_addr initial buffer address (choice)
OUT size buffer size, in bytes (non-negative integer)

int MPI_Buffer_detach(void* buffer_addr, int* size)

```
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR
```

```
int MPI::Detach_buffer(void*& buffer)
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.11 Calls to attach and detach buffers.

```
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type void* (rather than void* and void**, respectively), so as to avoid complex type casts. E.g., in the last example, &buff, which is of type char**, can be passed as argument to MPI_Buffer_detach without type casting. If the formal parameter had type void** then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be

46 CHAPTER 3. POINT-TO-POINT COMMUNICATION

```
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR
```

```
int MPI::Detach_buffer(void*& buffer)
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.11 Calls to attach and detach buffers.

```
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type void* (rather than void* and void**, respectively), so as to avoid complex type casts. E.g., in the last example, &buff, which is of type char**, can be passed as argument to MPI_Buffer_detach without type casting. If the formal parameter had type void** then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

Rationale. There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be

dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 4.2 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI_PACK_SIZE(count, datatype, comm, size), with the `count, datatype` and `comm` arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use **nonblocking communication**. A nonblocking **send start** call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate **send complete** call is needed

3.7. NONBLOCKING COMMUNICATION 47

dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 4.2 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI_PACK_SIZE(count, datatype, comm, size), with the `count, datatype` and `comm` arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use **nonblocking communication**. A nonblocking **send start** call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate **send complete** call is needed

to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: standard, buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready excepted, can be started whether a matching receive has been posted or not; a nonblocking ready send can be started only if a matching receive is posted. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in “pathological” cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is synchronous, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender “knows” the transfer will complete, but before the receiver “knows” the transfer will complete.)

If the send mode is buffered then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is standard then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a send operation may be delayed, for standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.

48 CHAPTER 3. POINT-TO-POINT COMMUNICATION

to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate **receive complete** call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: standard, buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready excepted, can be started whether a matching receive has been posted or not; a nonblocking ready send can be started only if a matching receive is posted. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in “pathological” cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is synchronous, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender “knows” the transfer will complete, but before the receiver “knows” the transfer will complete.)

If the send mode is buffered then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

If the send mode is standard then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Advice to users. The completion of a send operation may be delayed, for standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

3.7.1 Communication Request Objects

Nonblocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

3.7.2 Communication Initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Isend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Start a standard mode, nonblocking send.

3.7. NONBLOCKING COMMUNICATION 49

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

3.7.1 Communication Request Objects

Nonblocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

3.7.2 Communication Initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Isend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const

Start a standard mode, nonblocking send.

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Ibsend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Start a buffered mode, nonblocking send.

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Issend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Start a synchronous mode, nonblocking send.

50 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Ibsend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Start a buffered mode, nonblocking send.

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Issend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const
```

Start a synchronous mode, nonblocking send.

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Start a ready mode nonblocking send.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

```
MPI::Request MPI::Comm::Irecv(void* buf, int count, const
             MPI::Datatype& datatype, int source, int tag) const
```

Start a nonblocking receive.

3.7. NONBLOCKING COMMUNICATION 51

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const

Start a ready mode nonblocking send.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

OUT buf initial address of receive buffer (choice)
IN count number of elements in receive buffer (non-negative integer)
IN datatype datatype of each receive buffer element (handle)
IN source rank of source (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Request *request)

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Irecv(void* buf, int count, const
             MPI::Datatype& datatype, int source, int tag) const

Start a nonblocking receive.

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A **null** handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single MPI_STATUS value, the normal MPI error return process should be used (not the MPI_ERROR field in the MPI_STATUS argument).

52 CHAPTER 3. POINT-TO-POINT COMMUNICATION

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A **null** handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single MPI_STATUS value, the normal MPI error return process should be used (not the MPI_ERROR field in the MPI_STATUS argument).

MPI_WAIT(request, status)

INOUT	request	request (handle)
OUT	status	status object (Status)

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

```
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::Request::Wait(MPI::Status& status)
```

```
void MPI::Request::Wait()
```

A call to MPI_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

Advice to users. Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (*End of advice to users.*)

Advice to implementors. In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

MPI_TEST(request, flag, status)

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

3.7. NONBLOCKING COMMUNICATION 53

MPI_WAIT(request, status)

INOUT request request (handle)
OUT status status object (Status)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Request::Wait(MPI::Status& status)

void MPI::Request::Wait()

A call to MPI_WAIT returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local operation.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_WAIT with a null or inactive request argument. In this case the operation returns immediately with empty status.

Advice to users. Successful return of MPI_WAIT after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (End of advice to users.)

Advice to implementors. In a multi-threaded environment, a call to MPI_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)

MPI_TEST(request, flag, status)

INOUT request communication request (handle)
OUT flag true if operation completed (logical)
OUT status status object (Status)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```
bool MPI::Request::Test(MPI::Status& status)

bool MPI::Request::Test()
```

A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false, otherwise. In this case, the value of the status object is undefined. MPI_TEST is a local operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_TEST with a null or inactive request argument. In such a case the operation returns with flag = true and empty status.

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives.

> *Advice to users.* The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (*End of advice to users.*)

> *Rationale.* The function MPI_TEST returns with flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (*End of rationale.*)

Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)

INOUT	request	communication request (handle)

```
int MPI_Request_free(MPI_Request *request)
```

54 CHAPTER 3. POINT-TO-POINT COMMUNICATION

bool MPI::Request::Test(MPI::Status& status)

bool MPI::Request::Test()

A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false, otherwise. In this case, the value of the status object is undefined. MPI_TEST is a local operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_TEST with a null or inactive request argument. In such a case the operation returns with flag = true and empty status.

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives.

Advice to users. The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (*End of advice to users.*)

Rationale. The function MPI_TEST returns with flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (*End of rationale.*)

Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)

INOUT request communication request (handle)

int MPI_Request_free(MPI_Request *request)

```
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

```
void MPI::Request::Free()
```

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

> *Rationale.* The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

> *Advice to users.* Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

Example 3.13 An example using MPI_REQUEST_FREE.

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank.EQ.0) THEN
    DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
    DO I=1, n-1
       CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_REQUEST_FREE(req, ierr)
       CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_WAIT(req, status, ierr)
    END DO
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
END IF
```

3.7. NONBLOCKING COMMUNICATION 55

```
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

```
void MPI::Request::Free()
```

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

> *Rationale.* The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

> *Advice to users.* Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

Example 3.13 An example using MPI_REQUEST_FREE.

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank.EQ.0) THEN
    DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
    DO I=1, n-1
       CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_REQUEST_FREE(req, ierr)
       CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_WAIT(req, status, ierr)
    END DO
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
END IF
```

3.7.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5.

Order Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

Example 3.14 Message ordering for nonblocking operations.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
      CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
      CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)
```

The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

Progress A call to MPI_WAIT that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

Example 3.15 An illustration of progress semantics.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
      CALL MPI_WAIT(r, status, ierr)
END IF
```

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

56 CHAPTER 3. POINT-TO-POINT COMMUNICATION

3.7.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5.

Order Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

Example 3.14 Message ordering for nonblocking operations.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
      CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
      CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)

The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

Progress A call to MPI_WAIT that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

Example 3.15 An illustration of progress semantics.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
      CALL MPI_WAIT(r, status, ierr)
END IF

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

If an MPI_TEST that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or MPI_TESTALL can be used to wait for all pending operations in a list. A call to MPI_WAITSOME or MPI_TESTSOME can be used to complete all enabled operations in a list.

MPI_WAITANY (count, array_of_requests, index, status)

IN	count	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of handle for operation that completed (integer)
OUT	status	status object (Status)

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
             MPI_Status *status)
```

```
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[], MPI::Status& status)
```

```
static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[])
```

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and a empty status.

The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.

3.7. NONBLOCKING COMMUNICATION 57

If an MPI_TEST that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or MPI_TESTALL can be used to wait for all pending operations in a list. A call to MPI_WAITSOME or MPI_TESTSOME can be used to complete all enabled operations in a list.

MPI_WAITANY (count, array_of_requests, index, status)

IN count list length (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of handle for operation that completed (integer)
OUT status status object (Status)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
             MPI_Status *status)

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR

static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[], MPI::Status& status)

static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[])

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and a empty status.

The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.

MPI_TESTANY(count, array_of_requests, index, flag, status)

IN	count	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of operation that completed, or MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete (logical)
OUT	status	status object (Status)

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
             int *flag, MPI_Status *status)
```

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
static bool MPI::Request::Testany(int count,
             MPI::Request array_of_requests[], int& index,
             MPI::Status& status)
```

```
static bool MPI::Request::Testany(int count,
             MPI::Request array_of_requests[], int& index)
```

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true, index = MPI_UNDEFINED, and an empty status.

If the array of requests contains active handles then the execution of MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1 ,..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

Rationale. The function MPI_TESTANY returns with flag = true exactly in those situations where the function MPI_WAITANY returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking MPI_WAITANY can be easily replaced by a nonblocking MPI_TESTANY. The same relation holds for the other pairs of Wait and Test functions defined in this section. (*End of rationale.*)

58 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_TESTANY(count, array_of_requests, index, flag, status)

IN	count	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of operation that completed, or MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete (logical)
OUT	status	status object (Status)

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
             int *flag, MPI_Status *status)
```

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
static bool MPI::Request::Testany(int count,
             MPI::Request array_of_requests[], int& index,
             MPI::Status& status)
```

```
static bool MPI::Request::Testany(int count,
             MPI::Request array_of_requests[], int& index)
```

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true, index = MPI_UNDEFINED, and an empty status.

If the array of requests contains active handles then the execution of MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1 ,..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

Rationale. The function MPI_TESTANY returns with flag = true exactly in those situations where the function MPI_WAITANY returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking MPI_WAITANY can be easily replaced by a nonblocking MPI_TESTANY. The same relation holds for the other pairs of Wait and Test functions defined in this section. (*End of rationale.*)

MPI_WAITALL(count, array_of_requests, array_of_statuses)

IN	count	lists length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	array_of_statuses	array of status objects (array of Status)

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
             MPI_Status *array_of_statuses)
```

```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*)
    INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static void MPI::Request::Waitall(int count,
             MPI::Request array_of_requests[],
             MPI::Status array_of_statuses[])
```

```
static void MPI::Request::Waitall(int count,
             MPI::Request array_of_requests[])
```

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array_of_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI_WAITALL(count, array_of_requests, array_of_statuses) has the same effect as the execution of MPI_WAIT(&array_of_request[i], &array_of_statuses[i]), for i=0 ,..., count-1, in some arbitrary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

When one or more of the communications completed by a call to MPI_WAITALL fail, it is desireable to return specific information on each communication. The function MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL will return MPI_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

Rationale. This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End of rationale.*)

3.7. NONBLOCKING COMMUNICATION 59

MPI_WAITALL(count, array_of_requests, array_of_statuses)

IN count lists length (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT array_of_statuses array of status objects (array of Status)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

static void MPI::Request::Waitall(int count,
MPI::Request array_of_requests[],
MPI::Status array_of_statuses[])

static void MPI::Request::Waitall(int count,
MPI::Request array_of_requests[])

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array_of_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI_WAITALL(count, array_of_requests, array_of_statuses) has the same effect as the execution of MPI_WAIT(&array_of_request[i], &array_of_statuses[i]), for i=0 ,..., count-1, in some arbitrary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

When one or more of the communications completed by a call to MPI_WAITALL fail, it is desireable to return specific information on each communication. The function MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL will return MPI_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

Rationale. This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End of rationale.*)

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN	count	lists length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	flag	(logical)
OUT	array_of_statuses	array of status objects (array of Status)

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
             MPI_Status *array_of_statuses)
```

```
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[],
             MPI::Status array_of_statuses[])
```

```
static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[])
```

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a local operation.

Errors that occurred during the execution of MPI_TESTALL are handled as errors in MPI_WAITALL.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	length of array_of_requests (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
             int *outcount, int *array_of_indices,
             MPI_Status *array_of_statuses)
```

60 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN	count	lists length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	flag	(logical)
OUT	array_of_statuses	array of status objects (array of Status)

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
             MPI_Status *array_of_statuses)
```

```
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[],
             MPI::Status array_of_statuses[])
```

```
static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[])
```

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a local operation.

Errors that occurred during the execution of MPI_TESTALL are handled as errors in MPI_WAITALL.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	length of array_of_requests (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
             int *outcount, int *array_of_indices,
             MPI_Status *array_of_statuses)
```

```
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[])
```

```
static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[])
```

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	length of array_of_requests (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Testsome(int incount, MPI_Request *array_of_requests,
             int *outcount, int *array_of_indices,
             MPI_Status *array_of_statuses)
```

3.7. NONBLOCKING COMMUNICATION 61

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[])

static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[])

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN incount length of array_of_requests (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT outcount number of completed requests (integer)
OUT array_of_indices array of indices of operations that completed (array of integers)
OUT array_of_statuses array of status objects for operations that completed (array of Status)

int MPI_Testsome(int incount, MPI_Request *array_of_requests,
             int *outcount, int *array_of_indices,
             MPI_Status *array_of_statuses)

```
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[])
```

```
static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[])
```

Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has completed it returns outcount = 0. If there is no active handle in the list it returns outcount = MPI_UNDEFINED.

MPI_TESTSOME is a local operation, which returns immediately, whereas MPI_WAITSOME will block until a communication completes, if it was passed a list that contains at least one active handle. Both calls fulfill a fairness requirement: If a request for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or MPI_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of MPI_TESTSOME are handled as for MPI_WAITSOME.

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications, with the latter, a new call is required for each communication that completes.

A server with multiple clients can use MPI_WAITSOME so as not to starve any client. Clients send messages to the server with service requests. The server calls MPI_WAITSOME with one receive request for each client, and then handles all receives that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

Advice to implementors. MPI_TESTSOME should complete as many pending communications as possible. (*End of advice to implementors.*)

Example 3.16 Client-server code (starvation can occur).

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
       DO i=1, size-1
          CALL MPI_IRECV(a(1,i), n, MPI_REAL, i tag,
                   comm, request_list(i), ierr)
```

62 CHAPTER 3. POINT-TO-POINT COMMUNICATION

```
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
             ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

```
static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[])
```

```
static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[])
```

Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has completed it returns outcount = 0. If there is no active handle in the list it returns outcount = MPI_UNDEFINED.

MPI_TESTSOME is a local operation, which returns immediately, whereas MPI_WAITSOME will block until a communication completes, if it was passed a list that contains at least one active handle. Both calls fulfill a fairness requirement: If a request for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or MPI_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of MPI_TESTSOME are handled as for MPI_WAITSOME.

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all completed communications, with the latter, a new call is required for each communication that completes.

A server with multiple clients can use MPI_WAITSOME so as not to starve any client. Clients send messages to the server with service requests. The server calls MPI_WAITSOME with one receive request for each client, and then handles all receives that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

Advice to implementors. MPI_TESTSOME should complete as many pending communications as possible. (*End of advice to implementors.*)

Example 3.16 Client-server code (starvation can occur).

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
       DO i=1, size-1
          CALL MPI_IRECV(a(1,i), n, MPI_REAL, i tag,
                   comm, request_list(i), ierr)
```

```
      END DO
      DO WHILE(.TRUE.)
         CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
         CALL DO_SERVICE(a(1,index))  ! handle one message
         CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                  comm, request_list(index), ierr)
      END DO
END IF
```

Example 3.17 Same code, using MPI_WAITSOME.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                      comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                        indices, statuses, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, indices(i)))
          CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                       comm, request_list(indices(i)), ierr)
       END DO
    END DO
END IF
```

3.7.6 Non-destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.

3.7. NONBLOCKING COMMUNICATION 63

```
      END DO
      DO WHILE(.TRUE.)
         CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
         CALL DO_SERVICE(a(1,index))  ! handle one message
         CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                  comm, request_list(index), ierr)
      END DO
END IF
```

Example 3.17 Same code, using MPI_WAITSOME.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                      comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                        indices, statuses, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, indices(i)))
          CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                       comm, request_list(indices(i)), ierr)
       END DO
    END DO
END IF
```

3.7.6 Non-destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from MPI_TEST (logical)
OUT	status	MPI_STATUS object if flag is true (Status)

```
int MPI_Request_get_status(MPI_Request request, int *flag,
             MPI_Status *status)
```

```
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

```
bool MPI::Request::Get_status(MPI::Status& status) const
```

```
bool MPI::Request::Get_status() const
```

Sets flag=true if the operation is complete, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to test, wait or free should be executed with that request. It sets flag=false if the operation is not complete.

3.8 Probe and Cancel

The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The MPI_CANCEL operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

MPI_IPROBE(source, tag, comm, flag, status)

IN	source	source rank, or MPI_ANY_SOURCE (integer)
IN	tag	tag value or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	flag	(logical)
OUT	status	status object (Status)

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
             MPI_Status *status)
```

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

64 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_REQUEST_GET_STATUS(request, flag, status)

IN request request (handle)
OUT flag boolean flag, same as from MPI_TEST (logical)
OUT status MPI_STATUS object if flag is true (Status)

int MPI_Request_get_status(MPI_Request request, int *flag,
             MPI_Status *status)

MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

bool MPI::Request::Get_status(MPI::Status& status) const

bool MPI::Request::Get_status() const

Sets flag=true if the operation is complete, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to test, wait or free should be executed with that request. It sets flag=false if the operation is not complete.

3.8 Probe and Cancel

The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The MPI_CANCEL operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

MPI_IPROBE(source, tag, comm, flag, status)

IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT flag (logical)
OUT status status object (Status)

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
             MPI_Status *status)

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

```
bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const
```

```
bool MPI::Comm::Iprobe(int source, int tag) const
```

MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.

If MPI_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multi-threaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

MPI_PROBE(source, tag, comm, status)

IN	source	source rank, or MPI_ANY_SOURCE (integer)
IN	tag	tag value, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
```

```
void MPI::Comm::Probe(int source, int tag) const
```

MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: if a call to MPI_PROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and

3.8. PROBE AND CANCEL 65

bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const

bool MPI::Comm::Iprobe(int source, int tag) const

MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.

If MPI_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multi-threaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

MPI_PROBE(source, tag, comm, status)

IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const

void MPI::Comm::Probe(int source, int tag) const

MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: if a call to MPI_PROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and

a matching message has been issued, then the call to MPI_IPROBE will eventually return flag = true unless the message is received by another concurrent receive operation.

Example 3.18 Use blocking probe to wait for an incoming message.

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.2) THEN
          DO i=1, 2
             CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                             comm, status, ierr)
             IF (status(MPI_SOURCE) .EQ. 0) THEN
100               CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
             ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
             END IF
          END DO
      END IF
```

Each message is received with the right type.

Example 3.19 A similar program to the previous example, but now it has a problem.

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.2) THEN
          DO i=1, 2
             CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                             comm, status, ierr)
             IF (status(MPI_SOURCE) .EQ. 0) THEN
100               CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                0, comm, status, ierr)
             ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                0, comm, status, ierr)
             END IF
          END DO
      END IF
```

We slightly modified example 3.18, using MPI_ANY_SOURCE as the source argument in the two receive calls in statements labeled 100 and 200. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI_PROBE.

66 CHAPTER 3. POINT-TO-POINT COMMUNICATION

a matching message has been issued, then the call to MPI_IPROBE will eventually return flag = true unless the message is received by another concurrent receive operation.

Example 3.18 Use blocking probe to wait for an incoming message.

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.2) THEN
          DO i=1, 2
             CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                             comm, status, ierr)
             IF (status(MPI_SOURCE) .EQ. 0) THEN
100               CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
             ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
             END IF
          END DO
      END IF
```

Each message is received with the right type.

Example 3.19 A similar program to the previous example, but now it has a problem.

```
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE IF (rank.EQ.2) THEN
          DO i=1, 2
             CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                             comm, status, ierr)
             IF (status(MPI_SOURCE) .EQ. 0) THEN
100               CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                0, comm, status, ierr)
             ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                0, comm, status, ierr)
             END IF
          END DO
      END IF
```

We slightly modified example 3.18, using MPI_ANY_SOURCE as the source argument in the two receive calls in statements labeled 100 and 200. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI_PROBE.

Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match the message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point. Suppose that this message has source s, tag t and communicator c. If the tag argument in the probe call has value MPI_ANY_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator c, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

MPI_CANCEL(request)

IN	request	communication request (handle)

```
int MPI_Cancel(MPI_Request *request)
```

```
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

```
void MPI::Request::Cancel() const
```

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. It is still necessary to complete a communication that has been marked for cancellation, using a call to MPI_REQUEST_FREE, MPI_WAIT or MPI_TEST (or any of the derived operations).

If a communication is marked for cancellation, then a MPI_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a busy wait loop for a canceled communication, then MPI_TEST will eventually be successful.

MPI_CANCEL can be used to cancel a communication that uses a persistent request (see Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI_CANCEL and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message.

Either the cancellation succeeds, or the communication succeeds, but not both. If a send is marked for cancellation, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally,

3.8. PROBE AND CANCEL 67

Advice to implementors. A call to MPI_PROBE(source, tag, comm, status) will match the message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point. Suppose that this message has source s, tag t and communicator c. If the tag argument in the probe call has value MPI_ANY_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator c, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors.*)

MPI_CANCEL(request)

IN request communication request (handle)

int MPI_Cancel(MPI_Request *request)

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

void MPI::Request::Cancel() const

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled. It is still necessary to complete a communication that has been marked for cancellation, using a call to MPI_REQUEST_FREE, MPI_WAIT or MPI_TEST (or any of the derived operations).

If a communication is marked for cancellation, then a MPI_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a busy wait loop for a canceled communication, then MPI_TEST will eventually be successful.

MPI_CANCEL can be used to cancel a communication that uses a persistent request (see Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI_CANCEL and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message.

Either the cancellation succeeds, or the communication succeeds, but not both. If a send is marked for cancellation, then it must be the case that either the send completes normally, in which case the message sent was received at the destination process, or that the send is successfully canceled, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive completes normally,

or that the receive is successfully canceled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

If the operation has been canceled, then information to that effect will be returned in the status argument of the operation that completes the communication.

MPI_TEST_CANCELLED(status, flag)

IN	status	status object (Status)
OUT	flag	(logical)

```
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

```
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

```
bool MPI::Status::Is_cancelled() const
```

Returns flag = true if the communication associated with the status object was canceled successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns flag = false, otherwise. If a receive operation might be canceled then one should call MPI_TEST_CANCELLED first, to check whether the operation was canceled, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (*End of advice to users.*)

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another.

68 CHAPTER 3. POINT-TO-POINT COMMUNICATION

or that the receive is successfully canceled, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

If the operation has been canceled, then information to that effect will be returned in the status argument of the operation that completes the communication.

MPI_TEST_CANCELLED(status, flag)

IN status status object (Status)
OUT flag (logical)

int MPI_Test_cancelled(MPI_Status *status, int *flag)

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR

bool MPI::Status::Is_cancelled() const

Returns flag = true if the communication associated with the status object was canceled successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns flag = false, otherwise. If a receive operation might be canceled then one should call MPI_TEST_CANCELLED first, to check whether the operation was canceled, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (*End of advice to users.*)

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another.

It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

3.9. PERSISTENT COMMUNICATION REQUESTS 69

It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements sent (non-negative integer)
IN datatype type of each element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements sent (non-negative integer)
IN datatype type of each element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

```
MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication request for a buffered mode send.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication object for a synchronous mode send operation.

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

70 CHAPTER 3. POINT-TO-POINT COMMUNICATION

```
MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication request for a buffered mode send.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication object for a synchronous mode send operation.

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const
```

Creates a persistent communication object for a ready mode send operation.

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements received (non-negative integer)
IN	datatype	type of each element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

```
MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const
              MPI::Datatype& datatype, int source, int tag) const
```

Creates a persistent communication request for a receive operation. The argument buf is marked as OUT because the user gives permission to write on the receive buffer by passing the argument to MPI_RECV_INIT.

A persistent communication request is inactive after it was created — no active communication is attached to the request.

A communication (send or receive) that uses a persistent request is initiated by the function MPI_START.

MPI_START(request)

INOUT	request	communication request (handle)

```
int MPI_Start(MPI_Request *request)
```

```
MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

```
void MPI::Prequest::Start()
```

The argument, request, is a handle returned by one of the previous five calls. The associated request should be inactive. The request becomes active once the call is made.

If the request is for a send with ready mode, then a matching receive should be posted before the call is made. The communication buffer should not be accessed after the call, and until the operation completes.

The call is local, with similar semantics to the nonblocking communication operations described in Section 3.7. That is, a call to MPI_START with a request created by

3.9. PERSISTENT COMMUNICATION REQUESTS 71

Creates a persistent communication object for a ready mode send operation.

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

OUT buf initial address of receive buffer (choice)
IN count number of elements received (non-negative integer)
IN datatype type of each element (handle)
IN source rank of source or MPI_ANY_SOURCE (integer)
IN tag message tag or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request)

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const
              MPI::Datatype& datatype, int source, int tag) const

Creates a persistent communication request for a receive operation. The argument buf is marked as OUT because the user gives permission to write on the receive buffer by passing the argument to MPI_RECV_INIT.

A persistent communication request is inactive after it was created — no active communication is attached to the request.

A communication (send or receive) that uses a persistent request is initiated by the function MPI_START.

MPI_START(request)

INOUT request communication request (handle)

int MPI_Start(MPI_Request *request)

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

void MPI::Prequest::Start()

The argument, request, is a handle returned by one of the previous five calls. The associated request should be inactive. The request becomes active once the call is made.

If the request is for a send with ready mode, then a matching receive should be posted before the call is made. The communication buffer should not be accessed after the call, and until the operation completes.

The call is local, with similar semantics to the nonblocking communication operations described in Section 3.7. That is, a call to MPI_START with a request created by

MPI_SEND_INIT starts a communication in the same manner as a call to MPI_ISEND; a call to MPI_START with a request created by MPI_BSEND_INIT starts a communication in the same manner as a call to MPI_IBSEND; and so on.

MPI_STARTALL(count, array_of_requests)

IN	count	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handle)

```
int MPI_Startall(int count, MPI_Request *array_of_requests)
```

```
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

```
static void MPI::Prequest::Startall(int count,
             MPI::Prequest array_of_requests[])
```

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to MPI_START (&array_of_requests[i]), executed for i=0 ,..., count-1, in some arbitrary order.

A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL call.

A persistent request is deallocated by a call to MPI_REQUEST_FREE (Section 3.7.3).

The call to MPI_REQUEST_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

Create (**Start Complete**)* **Free**

where $*$ indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

> *Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

72 CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI_SEND_INIT starts a communication in the same manner as a call to MPI_ISEND; a call to MPI_START with a request created by MPI_BSEND_INIT starts a communication in the same manner as a call to MPI_IBSEND; and so on.

MPI_STARTALL(count, array_of_requests)

IN count list length (non-negative integer)
INOUT array_of_requests array of requests (array of handle)

int MPI_Startall(int count, MPI_Request *array_of_requests)

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

static void MPI::Prequest::Startall(int count,
             MPI::Prequest array_of_requests[])

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to MPI_START (&array_of_requests[i]), executed for i=0 ,..., count-1, in some arbitrary order.

A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL call.

A persistent request is deallocated by a call to MPI_REQUEST_FREE (Section 3.7.3).

The call to MPI_REQUEST_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

Create (**Start Complete**)* **Free**

where $*$ indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

3.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	sendbuf	initial address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	type of elements in send buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send tag (integer)
OUT	recvbuf	initial address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	type of elements in receive buffer (handle)
IN	source	rank of source (integer)
IN	recvtag	receive tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
             int dest, int sendtag, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
             MPI_Status *status)
```

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

3.10. SEND-RECEIVE 73

3.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN sendbuf initial address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype type of elements in send buffer (handle)
IN dest rank of destination (integer)
IN sendtag send tag (integer)
OUT recvbuf initial address of receive buffer (choice)
IN recvcount number of elements in receive buffer (non-negative integer)
IN recvtype type of elements in receive buffer (handle)
IN source rank of source (integer)
IN recvtag receive tag (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

```
void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
             int recvcount, const MPI::Datatype& recvtype, int source,
             int recvtag, MPI::Status& status) const
```

```
void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
             int recvcount, const MPI::Datatype& recvtype, int source,
             int recvtag) const
```

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT	buf	initial address of send and receive buffer (choice)
IN	count	number of elements in send and receive buffer (non-negative integer)
IN	datatype	type of elements in send and receive buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send message tag (integer)
IN	source	rank of source (integer)
IN	recvtag	receive message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
             int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
             MPI_Status *status)
```

```
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::Comm::Sendrecv_replace(void* buf, int count, const
             MPI::Datatype& datatype, int dest, int sendtag, int source,
             int recvtag, MPI::Status& status) const
```

```
void MPI::Comm::Sendrecv_replace(void* buf, int count, const
             MPI::Datatype& datatype, int dest, int sendtag, int source,
             int recvtag) const
```

74 CHAPTER 3. POINT-TO-POINT COMMUNICATION

void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag, MPI::Status& status) const

void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag) const

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT buf initial address of send and receive buffer (choice)
IN count number of elements in send and receive buffer (non-negative integer)
IN datatype type of elements in send and receive buffer (handle)
IN dest rank of destination (integer)
IN sendtag send message tag (integer)
IN source rank of source (integer)
IN recvtag receive message tag (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag, MPI::Status& status) const

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag) const

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

> *Advice to implementors.* Additional intermediate buffering is needed for the "replace" variant. (*End of advice to implementors.*)

3.11 Null Processes

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0.

3.11. NULL PROCESSES 75

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Advice to implementors. Additional intermediate buffering is needed for the "replace" variant. (*End of advice to implementors.*)

3.11 Null Processes

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0.

76 CHAPTER 3. POINT-TO-POINT COMMUNICATION

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 27 and in Section 3.3 Data Type Matching and Data Conversion on page 34. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

Up to here, all point to point communication have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A **general datatype** is an opaque object that specifies two things:

- A sequence of basic datatypes

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 27 and in Section 3.3 Data Type Matching and Data Conversion on page 34. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

Up to here, all point to point communication have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A **general datatype** is an opaque object that specifies two things:

- A sequence of basic datatypes

77

- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

Let

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = \{type_0, ..., type_{n-1}\}$$

be the associated type signature. This type map, together with a base address *buf*, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by $Typesig$.

Most datatype constructors have replication count or block length arguments. Allowed values are nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(\mathsf{int}, 0)\}$, with one entry of type int and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i, then ϵ is the least nonnegative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. The complete definition of **extent** is given on page 96.

78 CHAPTER 4. DATATYPES

- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

Let

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = \{type_0, ..., type_{n-1}\}$$

be the associated type signature. This type map, together with a base address *buf*, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by $Typesig$.

Most datatype constructors have replication count or block length arguments. Allowed values are nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument count has value > 1.

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(\mathsf{int}, 0)\}$, with one entry of type int and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i, then ϵ is the least nonnegative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. The complete definition of **extent** is given on page 96.

Example 4.1 Assume that $Type = \{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

> *Rationale.* The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (*End of rationale.*)

4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint and MPI::Aint are used in C and C++. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8.

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (nonnegative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_contiguous(int count) const
```

newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using *extent* as the size of the concatenated copies.

Example 4.2 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16, and let $\mathsf{count} = 3$. The type map of the datatype returned by newtype is

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40)\};$$

i.e., alternating double and char elements, with displacements $0, 8, 16, 24, 32, 40$.

4.1. DERIVED DATATYPES 79

Example 4.1 Assume that $Type = \{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (*End of rationale.*)

4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint and MPI::Aint are used in C and C++. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER*8.

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN count replication count (nonnegative integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_contiguous(int count) const

newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using *extent* as the size of the concatenated copies.

Example 4.2 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16, and let $\mathsf{count} = 3$. The type map of the datatype returned by newtype is

$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40)\};$

i.e., alternating double and char elements, with displacements $0, 8, 16, 24, 32, 40$.

In general, assume that the type map of oldtype is

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Then newtype has a type map with $\mathsf{count} \cdot \mathsf{n}$ entries defined by:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex),$$

$$..., (type_0, disp_0 + ex \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + ex \cdot (\mathsf{count} - 1))\}.$$

Vector The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (nonnegative integer)
IN	blocklength	number of elements in each block (nonnegative integer)
IN	stride	number of elements between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_vector(int count, int blocklength, int stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,
             int stride) const
```

Example 4.3 Assume, again, that oldtype has type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40),$$

$$(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104)\}.$$

That is, two blocks with three copies each of the old type, with a stride of 4 elements ($4 \cdot 16$ bytes) between the blocks.

80 CHAPTER 4. DATATYPES

In general, assume that the type map of oldtype is

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Then newtype has a type map with $\mathsf{count} \cdot \mathsf{n}$ entries defined by:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex),$$

$$..., (type_0, disp_0 + ex \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + ex \cdot (\mathsf{count} - 1))\}.$$

Vector The function MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN count number of blocks (nonnegative integer)
IN blocklength number of elements in each block (nonnegative integer)
IN stride number of elements between start of each block (integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)

int MPI_Type_vector(int count, int blocklength, int stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,
             int stride) const

Example 4.3 Assume, again, that oldtype has type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{double}, 32), (\mathsf{char}, 40),$$

$$(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104)\}.$$

That is, two blocks with three copies each of the old type, with a stride of 4 elements ($4 \cdot 16$ bytes) between the blocks.

Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, -32), (\mathsf{char}, -24), (\mathsf{double}, -64), (\mathsf{char}, -56)\}.$$

In general, assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$
$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$
$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$
$$(type_0, disp_0 + \mathsf{stride} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), ...,$$
$$(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ...,$$
$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$
$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$
$$(type_0, disp_0 + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex), ...,$$
$$(type_{n-1}, disp_{n-1} + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex)\}.$$

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.

Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for "heterogeneous").

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (nonnegative integer)
IN	blocklength	number of elements in each block (nonnegative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

4.1. DERIVED DATATYPES 81

Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8), (\mathsf{double}, -32), (\mathsf{char}, -24), (\mathsf{double}, -64), (\mathsf{char}, -56)\}.$$

In general, assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$
$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$
$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$
$$(type_0, disp_0 + \mathsf{stride} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), ...,$$
$$(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), ...,$$
$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$
$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), ...,$$
$$(type_0, disp_0 + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex), ...,$$
$$(type_{n-1}, disp_{n-1} + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex)\}.$$

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.

Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for "heterogeneous").

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (nonnegative integer)
IN	blocklength	number of elements in each block (nonnegative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
```

```
MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
             MPI::Aint stride) const
```

This function replaces MPI_TYPE_HVECTOR, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride}), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride}), ...,$$

$$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$$

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

82 CHAPTER 4. DATATYPES

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
```

```
MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
             MPI::Aint stride) const
```

This function replaces MPI_TYPE_HVECTOR, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride}), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride}), ...,$$

$$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$$

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements per block (array of nonnegative integers)
IN	array_of_displacements	displacement for each block, in multiples of oldtype extent (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
             int *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_indexed(int count,
             const int array_of_blocklengths[],
             const int array_of_displacements[]) const
```

Example 4.5 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. Let $\mathsf{B} = (3, 1)$ and let $\mathsf{D} = (4, 0)$. A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

$$\{(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104),$$

$$(\mathsf{double}, 0), (\mathsf{char}, 8)\}.$$

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent *ex*. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D}[0] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[0] \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D}[0] + \mathsf{B}[0] - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{D}[0] + \mathsf{B}[0] - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D}[\mathsf{count} - 1] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count} - 1] \cdot ex), ...,$$

4.1. DERIVED DATATYPES 83

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN count: number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN array_of_blocklengths: number of elements per block (array of nonnegative integers)
IN array_of_displacements: displacement for each block, in multiples of oldtype extent (array of integer)
IN oldtype: old datatype (handle)
OUT newtype: new datatype (handle)

int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[]) const

Example 4.5 Let oldtype have type map $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, with extent 16. Let $\mathsf{B} = (3, 1)$ and let $\mathsf{D} = (4, 0)$. A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

$$\{(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104),$$

$$(\mathsf{double}, 0), (\mathsf{char}, 8)\}.$$

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent *ex*. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D}[0] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[0] \cdot ex), ...,$$

$$(type_0, disp_0 + (\mathsf{D}[0] + \mathsf{B}[0] - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{D}[0] + \mathsf{B}[0] - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D}[\mathsf{count} - 1] \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count} - 1] \cdot ex), ...,$$

$(type_0, disp_0 + (\mathsf{D}[\mathsf{count}-1] + \mathsf{B}[\mathsf{count}-1] - 1) \cdot ex), ...,$

$(type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count}-1] + \mathsf{B}[\mathsf{count}-1] - 1) \cdot ex)\}.$

A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where

$$\mathsf{D}[\mathsf{j}] = j \cdot \mathsf{stride},\ j = 0, ..., \mathsf{count}-1,$$

and

$$\mathsf{B}[\mathsf{j}] = \mathsf{blocklength},\ j = 0, ..., \mathsf{count}-1.$$

Hindexed The function MPI_TYPE_CREATE_HINDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements in each block (array of nonnegative integers)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
MPI::Datatype MPI::Datatype::Create_hindexed(int count,
             const int array_of_blocklengths[],
             const MPI::Aint array_of_displacements[]) const
```

This function replaces MPI_TYPE_HINDEXED, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

84 CHAPTER 4. DATATYPES

$(type_0, disp_0 + (\mathsf{D}[\mathsf{count}-1] + \mathsf{B}[\mathsf{count}-1] - 1) \cdot ex), ...,$

$(type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count}-1] + \mathsf{B}[\mathsf{count}-1] - 1) \cdot ex)\}.$

A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where

$$\mathsf{D}[\mathsf{j}] = j \cdot \mathsf{stride},\ j = 0, ..., \mathsf{count}-1,$$

and

$$\mathsf{B}[\mathsf{j}] = \mathsf{blocklength},\ j = 0, ..., \mathsf{count}-1.$$

Hindexed The function MPI_TYPE_CREATE_HINDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements in each block (array of nonnegative integers)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
MPI::Datatype MPI::Datatype::Create_hindexed(int count,
             const int array_of_blocklengths[],
             const MPI::Aint array_of_displacements[]) const
```

This function replaces MPI_TYPE_HINDEXED, whose use is deprecated. See also Chapter 15.

Assume that oldtype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent ex. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has a type map with $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D[0]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex)\}.$$

Indexed_block This function is the same as MPI_TYPE_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

IN	count	length of array of displacements (non-negative integer)
IN	blocklength	size of block (non-negative integer)
IN	array_of_displacements	array of displacements (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_indexed_block(int count, int blocklength,
              int array_of_displacements[], MPI_Datatype oldtype,
              MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_indexed_block(int count,
              int blocklength, const int array_of_displacements[]) const
```

4.1. DERIVED DATATYPES 85

with extent ex. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. The newly created datatype has a type map with $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

$$\{(type_0, disp_0 + \mathsf{D[0]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot ex)\}.$$

Indexed_block This function is the same as MPI_TYPE_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

IN	count	length of array of displacements (non-negative integer)
IN	blocklength	size of block (non-negative integer)
IN	array_of_displacements	array of displacements (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_indexed_block(int count, int blocklength,
              int array_of_displacements[], MPI_Datatype oldtype,
              MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_indexed_block(int count,
              int blocklength, const int array_of_displacements[]) const
```

Struct MPI_TYPE_STRUCT is the most general type constructor. It further generalizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of replications of different datatypes.

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (nonnegative integer) — also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN	array_of_blocklength	number of elements in each block (array of nonnegative integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handles to datatype objects)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[],
             MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
static MPI::Datatype MPI::Datatype::Create_struct(int count,
             const int array_of_blocklengths[], const MPI::Aint
             array_of_displacements[],
             const MPI::Datatype array_of_types[])
```

This function replaces MPI_TYPE_STRUCT, whose use is deprecated. See also Chapter 15.

Example 4.6 Let type1 have type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8)\},$$

with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR). Then a call to MPI_TYPE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

$$\{(\mathsf{float}, 0), (\mathsf{float}, 4), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{char}, 26), (\mathsf{char}, 27), (\mathsf{char}, 28)\}.$$

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)

86 CHAPTER 4. DATATYPES

Struct MPI_TYPE_STRUCT is the most general type constructor. It further generalizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of replications of different datatypes.

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (nonnegative integer) — also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN	array_of_blocklength	number of elements in each block (array of nonnegative integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handles to datatype objects)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[],
             MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
             ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

```
static MPI::Datatype MPI::Datatype::Create_struct(int count,
             const int array_of_blocklengths[], const MPI::Aint
             array_of_displacements[],
             const MPI::Datatype array_of_types[])
```

This function replaces MPI_TYPE_STRUCT, whose use is deprecated. See also Chapter 15.

Example 4.6 Let type1 have type map,

$$\{(\mathsf{double}, 0), (\mathsf{char}, 8)\},$$

with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR). Then a call to MPI_TYPE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

$$\{(\mathsf{float}, 0), (\mathsf{float}, 4), (\mathsf{double}, 16), (\mathsf{char}, 24), (\mathsf{char}, 26), (\mathsf{char}, 27), (\mathsf{char}, 28)\}.$$

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)

In general, let T be the array_of_types argument, where T[i] is a handle to,

$$typemap_i = \{(type^i_0, disp^i_0), ..., (type^i_{n_i-1}, disp^i_{n_i-1})\},$$

with extent ex_i. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{\mathsf{c}-1} \mathsf{B[i]} \cdot n_i$ entries:

$$\{(type^0_0, disp^0_0 + \mathsf{D[0]}), ..., (type^0_{n_0}, disp^0_{n_0} + \mathsf{D[0]}), ...,$$

$$(type^0_0, disp^0_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ..., (type^0_{n_0}, disp^0_{n_0} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ...,$$

$$(type^{\mathsf{c}-1}_0, disp^{\mathsf{c}-1}_0 + \mathsf{D[c-1]}), ..., (type^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1}, disp^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1} + \mathsf{D[c-1]}), ...,$$

$$(type^{\mathsf{c}-1}_0, disp^{\mathsf{c}-1}_0 + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1}), ...,$$

$$(type^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1}, disp^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1} + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1})\}.$$

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN	ndims	number of array dimensions (positive integer)
IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of nonnegative integers)
IN	order	array storage order flag (state)
IN	oldtype	array element datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
             int array_of_subsizes[], int array_of_starts[], int order,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
             ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

4.1. DERIVED DATATYPES 87

In general, let T be the array_of_types argument, where T[i] is a handle to,

$$typemap_i = \{(type^i_0, disp^i_0), ..., (type^i_{n_i-1}, disp^i_{n_i-1})\},$$

with extent ex_i. Let B be the array_of_blocklength argument and D be the array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{\mathsf{c}-1} \mathsf{B[i]} \cdot n_i$ entries:

$$\{(type^0_0, disp^0_0 + \mathsf{D[0]}), ..., (type^0_{n_0}, disp^0_{n_0} + \mathsf{D[0]}), ...,$$

$$(type^0_0, disp^0_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ..., (type^0_{n_0}, disp^0_{n_0} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), ...,$$

$$(type^{\mathsf{c}-1}_0, disp^{\mathsf{c}-1}_0 + \mathsf{D[c-1]}), ..., (type^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1}, disp^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1} + \mathsf{D[c-1]}), ...,$$

$$(type^{\mathsf{c}-1}_0, disp^{\mathsf{c}-1}_0 + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1}), ...,$$

$$(type^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1}, disp^{\mathsf{c}-1}_{n_{\mathsf{c}-1}-1} + \mathsf{D[c-1]} + (\mathsf{B[c-1]} - 1) \cdot ex_{\mathsf{c}-1})\}.$$

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN	ndims	number of array dimensions (positive integer)
IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of nonnegative integers)
IN	order	array storage order flag (state)
IN	oldtype	array element datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
             int array_of_subsizes[], int array_of_starts[], int order,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
             ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_subarray(int ndims,
             const int array_of_sizes[], const int array_of_subsizes[],
             const int array_of_starts[], int order) const
```

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 373.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

The number of elements of type oldtype in each dimension of the n-dimensional array and the requested subarray are specified by array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] $<$ 1 or array_of_subsizes[i] $>$ array_of_sizes[i].

The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array_of_starts[i] < 0 or array_of_starts[i] $>$ (array_of_sizes[i] $-$ array_of_subsizes[i]).

> *Advice to users.* In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is n, then the entry in array_of_starts for that dimension is n-1. (*End of advice to users.*)

The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)

MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)

A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:

$$\begin{aligned}\mathsf{newtype} \;=\; & \mathrm{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\},\\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype})\end{aligned}$$

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 4.4 defines the recursion step when order = MPI_ORDER_C.

88 CHAPTER 4. DATATYPES

```
MPI::Datatype MPI::Datatype::Create_subarray(int ndims,
             const int array_of_sizes[], const int array_of_subsizes[],
             const int array_of_starts[], int order) const
```

The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 373.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

The number of elements of type oldtype in each dimension of the n-dimensional array and the requested subarray are specified by array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] $<$ 1 or array_of_subsizes[i] $>$ array_of_sizes[i].

The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array_of_starts[i] < 0 or array_of_starts[i] $>$ (array_of_sizes[i] $-$ array_of_subsizes[i]).

> *Advice to users.* In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is n, then the entry in array_of_starts for that dimension is n-1. (*End of advice to users.*)

The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)

MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)

A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:

$$\begin{aligned}\mathsf{newtype} \;=\; & \mathrm{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\},\\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype})\end{aligned}$$

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 4.4 defines the recursion step when order = MPI_ORDER_C.

$$\begin{aligned}
&\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \\
&\qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}) \\
&= \{(\mathsf{MPI\_LB}, 0), \\
&\quad (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex), \\
&\quad (type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1}, \\
&\qquad disp_{n-1} + (start_0 + 1) \times ex), \ldots \\
&\quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex), \\
&\quad (\mathsf{MPI\_UB}, size_0 \times ex)\}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_1, size_2, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_1, start_2, \ldots, start_{ndims-1}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \mathsf{oldtype}))
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_0, size_1, \ldots, size_{ndims-2}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-2}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \mathsf{oldtype}))
\end{aligned} \tag{4.4}$$

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [30] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 on page 373 and Section 13.3 on page 385. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

4.1. DERIVED DATATYPES 89

$$\begin{aligned}
&\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \\
&\qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}) \\
&= \{(\mathsf{MPI\_LB}, 0), \\
&\quad (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex), \\
&\quad (type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1}, \\
&\qquad disp_{n-1} + (start_0 + 1) \times ex), \ldots \\
&\quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex), \\
&\quad (\mathsf{MPI\_UB}, size_0 \times ex)\}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_1, size_2, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_1, start_2, \ldots, start_{ndims-1}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \mathsf{oldtype}))
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \text{Subarray}(ndims - 1, \{size_0, size_1, \ldots, size_{ndims-2}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-2}\}, \\
&\qquad\quad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \mathsf{oldtype}))
\end{aligned} \tag{4.4}$$

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [30] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 on page 373 and Section 13.3 on page 385. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype)

IN	size	size of process group (positive integer)
IN	rank	rank in process group (nonnegative integer)
IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)
IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)
IN	array_of_distribs	distribution of array in each dimension (array of state)
IN	array_of_dargs	distribution argument in each dimension (array of positive integers)
IN	array_of_psizes	size of process grid in each dimension (array of positive integers)
IN	order	array storage order flag (state)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_create_darray(int size, int rank, int ndims,
              int array_of_gsizes[], int array_of_distribs[], int
              array_of_dargs[], int array_of_psizes[], int order,
              MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
              ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims,
              const int array_of_gsizes[], const int array_of_distribs[],
              const int array_of_dargs[], const int array_of_psizes[],
              int order) const
```

MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See Example 4.7, page 93.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array\_of\_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies .

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7 on page 241. (*End of advice to users.*)

90 CHAPTER 4. DATATYPES

MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype)

IN size — size of process group (positive integer)
IN rank — rank in process group (nonnegative integer)
IN ndims — number of array dimensions as well as process grid dimensions (positive integer)
IN array_of_gsizes — number of elements of type oldtype in each dimension of global array (array of positive integers)
IN array_of_distribs — distribution of array in each dimension (array of state)
IN array_of_dargs — distribution argument in each dimension (array of positive integers)
IN array_of_psizes — size of process grid in each dimension (array of positive integers)
IN order — array storage order flag (state)
IN oldtype — old datatype (handle)
OUT newtype — new datatype (handle)

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR)
INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims, const int array_of_gsizes[], const int array_of_distribs[], const int array_of_dargs[], const int array_of_psizes[], int order) const

MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See Example 4.7, page 93.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array\_of\_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies .

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7 on page 241. (*End of advice to users.*)

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK - Block distribution
- MPI_DISTRIBUTE_CYCLIC - Cyclic distribution
- MPI_DISTRIBUTE_NONE - Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] $*$ array_of_psizes[i] $<$ array_of_gsizes[i].

For example, the HPF layout `ARRAY(CYCLIC(15))` corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout ARRAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to

(array_of_gsizes[i] $+$ array_of_psizes[i] $- 1)/$array_of_psizes[i].

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 1.

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i+1] = cyclic(array_of_dargs[i],
                         array_of_gsizes[i],
                         r[i],
                         array_of_psizes[i],
                         oldtype[i]);
}
newtype = oldtype[ndims];
```

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK - Block distribution
- MPI_DISTRIBUTE_CYCLIC - Cyclic distribution
- MPI_DISTRIBUTE_NONE - Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] $*$ array_of_psizes[i] $<$ array_of_gsizes[i].

For example, the HPF layout `ARRAY(CYCLIC(15))` corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout ARRAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

This routine creates a new MPI datatype with a typemap defined in terms of a function called "cyclic()" (see below).

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to

(array_of_gsizes[i] $+$ array_of_psizes[i] $- 1)/$array_of_psizes[i].

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 1.

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i+1] = cyclic(array_of_dargs[i],
                         array_of_gsizes[i],
                         r[i],
                         array_of_psizes[i],
                         oldtype[i]);
}
newtype = oldtype[ndims];
```

For MPI_ORDER_C, the code is:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                           array_of_gsizes[ndims - i - 1],
                           r[ndims - i - 1],
                           array_of_psizes[ndims - i - 1],
                           oldtype[i]);
}
newtype = oldtype[ndims];
```

where $r[i]$ is the position of the process (with rank rank) in the process grid at dimension i. The values of $r[i]$ are given by the following code fragment:

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
        t_size *= array_of_psizes[i];
for (i = 0; i < ndims; i++) {
    t_size = t_size / array_of_psizes[i];
    r[i] = t_rank / t_size;
    t_rank = t_rank % t_size;
}
```

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype.

Given the above, the function cyclic() is defined as follows:

$$\begin{aligned}
&\mathrm{cyclic}(darg, gsize, r, psize, \mathsf{oldtype}) \\
&\quad = \{(\mathsf{MPI\_LB}, 0), \\
&\qquad (type_0, disp_0 + r \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\qquad \ldots \\
&\qquad (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex), \\
&\\
&\qquad (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
\end{aligned}$$

92 CHAPTER 4. DATATYPES

For MPI_ORDER_C, the code is:

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                           array_of_gsizes[ndims - i - 1],
                           r[ndims - i - 1],
                           array_of_psizes[ndims - i - 1],
                           oldtype[i]);
}
newtype = oldtype[ndims];
```

where $r[i]$ is the position of the process (with rank rank) in the process grid at dimension i. The values of $r[i]$ are given by the following code fragment:

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
        t_size *= array_of_psizes[i];
for (i = 0; i < ndims; i++) {
    t_size = t_size / array_of_psizes[i];
    r[i] = t_rank / t_size;
    t_rank = t_rank % t_size;
}
```

Let the typemap of oldtype have the form:

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

where $type_i$ is a predefined MPI datatype, and let ex be the extent of oldtype.

Given the above, the function cyclic() is defined as follows:

$$\begin{aligned}
&\mathrm{cyclic}(darg, gsize, r, psize, \mathsf{oldtype}) \\
&\quad = \{(\mathsf{MPI\_LB}, 0), \\
&\qquad (type_0, disp_0 + r \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\qquad \ldots \\
&\qquad (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex), \\
&\\
&\qquad (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
\end{aligned}$$

$$\begin{array}{l}
\ldots \\
(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \\
\qquad \vdots \\
(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \\
(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
\ldots \\
(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
(\mathsf{MPI\_UB}, gsize * ex)\}
\end{array}$$

where *count* is defined by this code fragment:

```
nblocks = (gsize + (darg - 1)) / darg;
count = nblocks / psize;
left_over = nblocks - count * psize;
if (r < left_over)
    count = count + 1;
```

Here, *nblocks* is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
     darg_last = darg;
else
     darg_last = num_in_last_cyclic - darg * r;
     if (darg_last > darg)
            darg_last = darg;
     if (darg_last <= 0)
            darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```
    ndims = 3
    array_of_gsizes(1) = 100
```

4.1. DERIVED DATATYPES 93

$$\begin{array}{l}
\ldots \\
(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \\
\qquad \vdots \\
(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \\
(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
\ldots \\
(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \ldots, \\
\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex \\
\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
(\mathsf{MPI\_UB}, gsize * ex)\}
\end{array}$$

where *count* is defined by this code fragment:

```
nblocks = (gsize + (darg - 1)) / darg;
count = nblocks / psize;
left_over = nblocks - count * psize;
if (r < left_over)
    count = count + 1;
```

Here, *nblocks* is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
     darg_last = darg;
else
     darg_last = num_in_last_cyclic - darg * r;
     if (darg_last > darg)
            darg_last = darg;
     if (darg_last <= 0)
            darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```
    ndims = 3
    array_of_gsizes(1) = 100
    array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
    array_of_dargs(1) = 10
    array_of_gsizes(2) = 200
    array_of_distribs(2) = MPI_DISTRIBUTE_NONE
    array_of_dargs(2) = 0
    array_of_gsizes(3) = 300
    array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
    array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_ARG
    array_of_psizes(1) = 2
    array_of_psizes(2) = 1
    array_of_psizes(3) = 3
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
         array_of_distribs, array_of_dargs, array_of_psizes,       &
         MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS.

MPI_GET_ADDRESS(location, address)

IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
```

```
MPI::Aint MPI::Get_address(void* location)
```

This function replaces MPI_ADDRESS, whose use is deprecated. See also Chapter 15.

Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions.

94 CHAPTER 4. DATATYPES

```
    array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
    array_of_dargs(1) = 10
    array_of_gsizes(2) = 200
    array_of_distribs(2) = MPI_DISTRIBUTE_NONE
    array_of_dargs(2) = 0
    array_of_gsizes(3) = 300
    array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
    array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_ARG
    array_of_psizes(1) = 2
    array_of_psizes(2) = 1
    array_of_psizes(3) = 3
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
         array_of_distribs, array_of_dargs, array_of_psizes,       &
         MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS.

MPI_GET_ADDRESS(location, address)

IN location location in caller memory (choice)
OUT address address of location (integer)

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
```

```
MPI::Aint MPI::Get_address(void* location)
```

This function replaces MPI_ADDRESS, whose use is deprecated. See also Chapter 15.

Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions.

This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```
   REAL A(100,100)
   INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
   CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such machines as well. (*End of advice to users.*)

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

```
int MPI::Datatype::Get_size() const
```

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

4.1. DERIVED DATATYPES 95

This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```
   REAL A(100,100)
   INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
   CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such machines as well. (*End of advice to users.*)

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 463 and 466. (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

```
int MPI::Datatype::Get_size() const
```

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 96. This allows one to define a datatype that has "holes" at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional "pseudo-datatypes," MPI_LB and MPI_UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space ($\mathit{extent}(\mathsf{MPI\_LB}) = \mathit{extent}(\mathsf{MPI\_UB}) = 0$). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let D = (-3, 0, 6); T = (MPI_LB, MPI_INT, MPI_UB), and B = (1, 1, 1). Then a call to MPI_TYPE_STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence {(lb, -3), (int, 0), (ub, 6)} . If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence {(lb, -3), (int, 0), (int,9), (ub, 15)} . (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

$$\mathit{Typemap} = \{(\mathit{type}_0, \mathit{disp}_0), ..., (\mathit{type}_{n-1}, \mathit{disp}_{n-1})\},$$

then the **lower bound** of *Typemap* is defined to be

$$\mathit{lb}(\mathit{Typemap}) = \begin{cases} \min_j \mathit{disp}_j & \text{if no entry has basic type } \mathsf{lb} \\ \min_j \{\mathit{disp}_j \text{ such that } \mathit{type}_j = \mathsf{lb}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be

$$\mathit{ub}(\mathit{Typemap}) = \begin{cases} \max_j \mathit{disp}_j + \mathit{sizeof}(\mathit{type}_j) + \epsilon & \text{if no entry has basic type } \mathsf{ub} \\ \max_j \{\mathit{disp}_j \text{ such that } \mathit{type}_j = \mathsf{ub}\} & \text{otherwise} \end{cases}$$

Then

$$\mathit{extent}(\mathit{Typemap}) = \mathit{ub}(\mathit{Typemap}) - \mathit{lb}(\mathit{Typemap})$$

If type_i requires alignment to a byte address that is a multiple of k_i, then ϵ is the least nonnegative increment needed to round $\mathit{extent}(\mathit{Typemap})$ to the next multiple of $\max_i k_i$.

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

96 CHAPTER 4. DATATYPES

4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 96. This allows one to define a datatype that has "holes" at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional "pseudo-datatypes," MPI_LB and MPI_UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space ($\mathit{extent}(\mathsf{MPI\_LB}) = \mathit{extent}(\mathsf{MPI\_UB}) = 0$). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let D = (-3, 0, 6); T = (MPI_LB, MPI_INT, MPI_UB), and B = (1, 1, 1). Then a call to MPI_TYPE_STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence {(lb, -3), (int, 0), (ub, 6)} . If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence {(lb, -3), (int, 0), (int,9), (ub, 15)} . (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

$$\mathit{Typemap} = \{(\mathit{type}_0, \mathit{disp}_0), ..., (\mathit{type}_{n-1}, \mathit{disp}_{n-1})\},$$

then the **lower bound** of *Typemap* is defined to be

$$\mathit{lb}(\mathit{Typemap}) = \begin{cases} \min_j \mathit{disp}_j & \text{if no entry has basic type } \mathsf{lb} \\ \min_j \{\mathit{disp}_j \text{ such that } \mathit{type}_j = \mathsf{lb}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be

$$\mathit{ub}(\mathit{Typemap}) = \begin{cases} \max_j \mathit{disp}_j + \mathit{sizeof}(\mathit{type}_j) + \epsilon & \text{if no entry has basic type } \mathsf{ub} \\ \max_j \{\mathit{disp}_j \text{ such that } \mathit{type}_j = \mathsf{ub}\} & \text{otherwise} \end{cases}$$

Then

$$\mathit{extent}(\mathit{Typemap}) = \mathit{ub}(\mathit{Typemap}) - \mathit{lb}(\mathit{Typemap})$$

If type_i requires alignment to a byte address that is a multiple of k_i, then ϵ is the least nonnegative increment needed to round $\mathit{extent}(\mathit{Typemap})$ to the next multiple of $\max_i k_i$.

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN	datatype	datatype to get information on (handle)
OUT	lb	lower bound of datatype (integer)
OUT	extent	extent of datatype (integer)

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
             MPI_Aint *extent)
```

```
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
```

```
void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const
```

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 96).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are “sticky”: once present in a datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB is deprecated.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN	oldtype	input datatype (handle)
IN	lb	new lower bound of datatype (integer)
IN	extent	new extent of datatype (integer)
OUT	newtype	output datatype (handle)

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
             extent, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

```
MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
             const MPI::Aint extent) const
```

4.1. DERIVED DATATYPES 97

4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN datatype datatype to get information on (handle)
OUT lb lower bound of datatype (integer)
OUT extent extent of datatype (integer)

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
             MPI_Aint *extent)

MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT

void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 96).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are “sticky”: once present in a datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB is deprecated.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN oldtype input datatype (handle)
IN lb new lower bound of datatype (integer)
IN extent new extent of datatype (integer)
OUT newtype output datatype (handle)

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
             extent, MPI_Datatype *newtype)

MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
             const MPI::Aint extent) const

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous **lb** and **ub** markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

Advice to users. It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (*End of advice to users.*)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 137) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_UB and MPI_LB values. A function is provided which returns the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	true_extent	true size of datatype (integer)

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
             MPI_Aint *true_extent)
```

```
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
```

```
void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
             MPI::Aint& true_extent) const
```

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

Then

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

98 CHAPTER 4. DATATYPES

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous **lb** and **ub** markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

Advice to users. It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (*End of advice to users.*)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 137) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_UB and MPI_LB values. A function is provided which returns the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

IN datatype datatype to get information on (handle)
OUT true_lb true lower bound of datatype (integer)
OUT true_extent true size of datatype (integer)

int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)

MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT

void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb, MPI::Aint& true_extent) const

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

Typemap = {(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})}

Then

true_lb(Typemap) = min_j{disp_j : type_j ≠ lb, ub},

true_ub(Typemap) = max_j{disp_j + sizeof(type_j) : type_j ≠ lb, ub},

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 on page 96 and Section 4.1.7 on page 97, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

INOUT	datatype	datatype that is committed (handle)

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

```
void MPI::Datatype::Commit()
```

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

> *Advice to implementors.* The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used for communication
type2 = type1
              ! type2 can be used for communication
              ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
              ! new uncommitted type object created
```

4.1. DERIVED DATATYPES 99

and

true_extent(Typemap) = true_ub(Typemap) − true_lb(typemap).

(Readers should compare this with the definitions in Section 4.1.6 on page 96 and Section 4.1.7 on page 97, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

INOUT datatype datatype that is committed (handle)

int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit()

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

Advice to implementors. The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.

INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used for communication
type2 = type1
              ! type2 can be used for communication
              ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
              ! new uncommitted type object created

```
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used anew for communication
```

MPI_TYPE_FREE(datatype)

INOUT	datatype	datatype that is freed (handle)

```
int MPI_Type_free(MPI_Datatype *datatype)
```

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

```
void MPI::Datatype::Free()
```

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(type, newtype)

IN	type	datatype (handle)
OUT	newtype	copy of type (handle)

```
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
```

```
MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR
```

```
MPI::Datatype MPI::Datatype::Dup() const
```

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as type and any copied cached information, see Section 6.7.4 on page 230. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded

100 CHAPTER 4. DATATYPES

CALL MPI_TYPE_COMMIT(type1, ierr)
! now type1 can be used anew for communication

MPI_TYPE_FREE(datatype)
INOUT datatype datatype that is freed (handle)

int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

void MPI::Datatype::Free()

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(type, newtype)
IN type datatype (handle)
OUT newtype copy of type (handle)

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
INTEGER TYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Dup() const

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as type and any copied cached information, see Section 6.7.4 on page 230. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded

with the functions in Section 4.1.13. The newtype has the same committed state as the old type.

4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype , ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

and extent *extent*. (Empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) The send operation sends $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = \mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, ..., \mathsf{count} - 1$ and $j = 0, ..., n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address $addr_{i,j}$ in the calling program should be of a type that matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

with extent *extent*. (Again, empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $\mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \mathsf{count}$; the $i \cdot n + j$-th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
```

4.1. DERIVED DATATYPES 101

with the functions in Section 4.1.13. The newtype has the same committed state as the old type.

4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype , ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).

Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$

and extent *extent*. (Empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) The send operation sends $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = \mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, ..., \mathsf{count} - 1$ and $j = 0, ..., n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address $addr_{i,j}$ in the calling program should be of a type that matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$

with extent *extent*. (Again, empty entries of "pseudo-type" MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot \mathsf{count}$ entries, where entry $i \cdot n + j$ is at location $\mathsf{buf} + extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \mathsf{count}$; the $i \cdot n + j$-th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)

```
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}.$$

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query function MPI_GET_ELEMENTS.

MPI_GET_ELEMENTS(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

```
int MPI::Status::Get_elements(const MPI::Datatype& datatype) const
```

The previously defined function, MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level entries" received, i.e. the number of "copies" of type datatype. In the previous example, MPI_GET_COUNT may return any integer value k, where $0 \leq k \leq \text{count}$. If MPI_GET_COUNT returns k, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is $n \cdot k$. If the number of basic elements received is not a multiple of n, that is, if the receive operation has not received an integral number of datatype "copies," then MPI_GET_COUNT returns the value MPI_UNDEFINED. The datatype argument should match the argument provided by the receive call that set the status variable.

102 CHAPTER 4. DATATYPES

```
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}.$$

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query function MPI_GET_ELEMENTS.

MPI_GET_ELEMENTS(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

```
int MPI::Status::Get_elements(const MPI::Datatype& datatype) const
```

The previously defined function, MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level entries" received, i.e. the number of "copies" of type datatype. In the previous example, MPI_GET_COUNT may return any integer value k, where $0 \leq k \leq \text{count}$. If MPI_GET_COUNT returns k, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is $n \cdot k$. If the number of basic elements received is not a multiple of n, that is, if the receive operation has not received an integral number of datatype "copies," then MPI_GET_COUNT returns the value MPI_UNDEFINED. The datatype argument should match the argument provided by the receive call that set the status variable.

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=3
END IF
```

The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

Rationale. The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS. (*End of rationale.*)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

4.1. DERIVED DATATYPES 103

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=3
END IF
```

The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

Rationale. The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS. (*End of rationale.*)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

1. The function MPI_GET_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
2. The buf argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
4. If v is a valid address then MPI_BOTTOM + v is a valid address.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users. It is not expected that MPI implementations will be able to detect erroneous, “out of bound” displacements — unless those overflow the user address space — since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

104 CHAPTER 4. DATATYPES

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

1. The function MPI_GET_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
2. The buf argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
4. If v is a valid address then MPI_BOTTOM + v is a valid address.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users. It is not expected that MPI implementations will be able to detect erroneous, “out of bound” displacements — unless those overflow the user address space — since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	datatype to access (handle)
OUT	num_integers	number of input integers used in the call constructing combiner (nonnegative integer)
OUT	num_addresses	number of input addresses used in the call constructing combiner (nonnegative integer)
OUT	num_datatypes	number of input datatypes used in the call constructing combiner (nonnegative integer)
OUT	combiner	combiner (state)

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)
```

```
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

```
void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const
```

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from MPI_TYPE_GET_CONTENTS may be used with either to produce the same outcome. C calls MPI_Type_hindexed and MPI_Type_create_hindexed are always effectively the same while the Fortran call MPI_TYPE_HINDEXED will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list below has the values that can be returned in combiner on the left and the call associated with them on the right.

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

4.1. DERIVED DATATYPES 105

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	datatype to access (handle)
OUT	num_integers	number of input integers used in the call constructing combiner (nonnegative integer)
OUT	num_addresses	number of input addresses used in the call constructing combiner (nonnegative integer)
OUT	num_datatypes	number of input datatypes used in the call constructing combiner (nonnegative integer)
OUT	combiner	combiner (state)

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)
```

```
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

```
void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const
```

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from MPI_TYPE_GET_CONTENTS may be used with either to produce the same outcome. C calls MPI_Type_hindexed and MPI_Type_create_hindexed are always effectively the same while the Fortran call MPI_TYPE_HINDEXED will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list below has the values that can be returned in combiner on the left and the call associated with them on the right.

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

MPI_COMBINER_NAMED	a named predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT from C or C++ and in some case Fortran or MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

For deprecated calls with address arguments, we sometimes need to differentiate whether the call used an integer or an address size argument. For example, there are two combiners for hvector: MPI_COMBINER_HVECTOR_INTEGER and MPI_COMBINER_HVECTOR. The former is used if it was the MPI-1 call from Fortran, and the latter is used if it was the MPI-1 call from C or C++. However, on systems where MPI_ADDRESS_KIND = MPI_INTEGER_KIND (i.e., where integer arguments and address size arguments are the same), the combiner MPI_COMBINER_HVECTOR may be returned for a datatype constructed by a call to MPI_TYPE_HVECTOR from Fortran. Similarly, MPI_COMBINER_HINDEXED may be returned for a datatype constructed by a call to MPI_TYPE_HINDEXED from Fortran, and MPI_COMBINER_STRUCT may be returned for a datatype constructed by a call to MPI_TYPE_STRUCT from Fortran. On such systems, one need not differentiate constructors that take address size arguments from constructors that take integer arguments, since these are the same. The preferred calls all use address sized arguments so two combiners are not required for them.

> *Rationale.* For recreating the original call, it is important to know if address information may have been truncated. The deprecated calls from Fortran for a few routines could be subject to truncation in the case where the default `INTEGER` size is smaller than the size of an address. (*End of rationale.*)

106 CHAPTER 4. DATATYPES

MPI_COMBINER_NAMED	a named predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT from C or C++ and in some case Fortran or MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

For deprecated calls with address arguments, we sometimes need to differentiate whether the call used an integer or an address size argument. For example, there are two combiners for hvector: MPI_COMBINER_HVECTOR_INTEGER and MPI_COMBINER_HVECTOR. The former is used if it was the MPI-1 call from Fortran, and the latter is used if it was the MPI-1 call from C or C++. However, on systems where MPI_ADDRESS_KIND = MPI_INTEGER_KIND (i.e., where integer arguments and address size arguments are the same), the combiner MPI_COMBINER_HVECTOR may be returned for a datatype constructed by a call to MPI_TYPE_HVECTOR from Fortran. Similarly, MPI_COMBINER_HINDEXED may be returned for a datatype constructed by a call to MPI_TYPE_HINDEXED from Fortran, and MPI_COMBINER_STRUCT may be returned for a datatype constructed by a call to MPI_TYPE_STRUCT from Fortran. On such systems, one need not differentiate constructors that take address size arguments from constructors that take integer arguments, since these are the same. The preferred calls all use address sized arguments so two combiners are not required for them.

Rationale. For recreating the original call, it is important to know if address information may have been truncated. The deprecated calls from Fortran for a few routines could be subject to truncation in the case where the default `INTEGER` size is smaller than the size of an address. (*End of rationale.*)

The actual arguments used in the creation call for a datatype can be obtained from the call:

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	datatype to access (handle)
IN	max_integers	number of elements in array_of_integers (nonnegative integer)
IN	max_addresses	number of elements in array_of_addresses (nonnegative integer)
IN	max_datatypes	number of elements in array_of_datatypes (nonnegative integer)
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
             int max_addresses, int max_datatypes, int array_of_integers[],
             MPI_Aint array_of_addresses[],
             MPI_Datatype array_of_datatypes[])
```

```
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
             ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
             IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
```

```
void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
             int max_datatypes, int array_of_integers[],
             MPI::Aint array_of_addresses[],
             MPI::Datatype array_of_datatypes[]) const
```

datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.

The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively, in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived

The actual arguments used in the creation call for a datatype can be obtained from the call:

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	datatype to access (handle)
IN	max_integers	number of elements in array_of_integers (nonnegative integer)
IN	max_addresses	number of elements in array_of_addresses (nonnegative integer)
IN	max_datatypes	number of elements in array_of_datatypes (nonnegative integer)
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
             int max_addresses, int max_datatypes, int array_of_integers[],
             MPI_Aint array_of_addresses[],
             MPI_Datatype array_of_datatypes[])
```

```
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
             ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
             IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
```

```
void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
             int max_datatypes, int array_of_integers[],
             MPI::Aint array_of_addresses[],
             MPI::Datatype array_of_datatypes[]) const
```

datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.

The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively, in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived

datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the `==` or `.EQ.` comparison operator to determine the datatype involved. (*End of rationale.*)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type `INTEGER`. In the preferred calls, the address arguments are of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type `INTEGER(KIND=MPI_ADDRESS_KIND)` will be at least as large as the `INTEGER` argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (*End of rationale.*)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

108 CHAPTER 4. DATATYPES

datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the `==` or `.EQ.` comparison operator to determine the datatype involved. (*End of rationale.*)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type `INTEGER`. In the preferred calls, the address arguments are of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type `INTEGER(KIND=MPI_ADDRESS_KIND)`. This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type `INTEGER(KIND=MPI_ADDRESS_KIND)` will be at least as large as the `INTEGER` argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (*End of rationale.*)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

```
      PARAMETER (LARGE = 1000)
      INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
!     CONSTRUCT DATATYPE TYPE (NOT SHOWN)
      CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
      IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
        WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
        " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
        CALL MPI_ABORT(MPI_COMM_WORLD, 99)
      ENDIF
      CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
```

or in C the analogous calls of:

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
  fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
  fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
          LARGE);
  MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

The C++ code is in analogy to the C code above with the same values returned.

In the descriptions that follow, the lower case name of arguments is used.

If combiner is MPI_COMBINER_NAMED then it is erroneous to call MPI_TYPE_GET_CONTENTS.

If combiner is MPI_COMBINER_DUP then

Constructor argument	C & C++ location	Fortran location
oldtype	d[0]	D(1)

and ni = 0, na = 0, nd = 1.

If combiner is MPI_COMBINER_CONTIGUOUS then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

and ni = 1, na = 0, nd = 1.

If combiner is MPI_COMBINER_VECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	i[2]	I(3)
oldtype	d[0]	D(1)

4.1. DERIVED DATATYPES 109

```
      PARAMETER (LARGE = 1000)
      INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
!     CONSTRUCT DATATYPE TYPE (NOT SHOWN)
      CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
      IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
        WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
        " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
        CALL MPI_ABORT(MPI_COMM_WORLD, 99)
      ENDIF
      CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
```

or in C the analogous calls of:

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
  fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
  fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
          LARGE);
  MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

The C++ code is in analogy to the C code above with the same values returned.

In the descriptions that follow, the lower case name of arguments is used.

If combiner is MPI_COMBINER_NAMED then it is erroneous to call MPI_TYPE_GET_CONTENTS.

If combiner is MPI_COMBINER_DUP then

Constructor argument	C & C++ location	Fortran location
oldtype	d[0]	D(1)

and ni = 0, na = 0, nd = 1.

If combiner is MPI_COMBINER_CONTIGUOUS then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

and ni = 1, na = 0, nd = 1.

If combiner is MPI_COMBINER_VECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	i[2]	I(3)
oldtype	d[0]	D(1)

and ni = 3, na = 0, nd = 1.

If combiner is MPI_COMBINER_HVECTOR_INTEGER or MPI_COMBINER_HVECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
oldtype	d[0]	D(1)

and ni = 2*count+1, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_INTEGER or MPI_COMBINER_HINDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
oldtype	d[0]	D(1)

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_STRUCT_INTEGER or MPI_COMBINER_STRUCT then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
array_of_types	d[0] to d[i[0]-1]	D(1) to D(I(1))

and ni = count+1, na = count, nd = count.

If combiner is MPI_COMBINER_SUBARRAY then

Constructor argument	C & C++ location	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

110 CHAPTER 4. DATATYPES

and ni = 3, na = 0, nd = 1.

If combiner is MPI_COMBINER_HVECTOR_INTEGER or MPI_COMBINER_HVECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
oldtype	d[0]	D(1)

and ni = 2*count+1, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_INTEGER or MPI_COMBINER_HINDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
oldtype	d[0]	D(1)

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_STRUCT_INTEGER or MPI_COMBINER_STRUCT then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
array_of_types	d[0] to d[i[0]-1]	D(1) to D(I(1))

and ni = count+1, na = count, nd = count.

If combiner is MPI_COMBINER_SUBARRAY then

Constructor argument	C & C++ location	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

Constructor argument	C & C++ location	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

Constructor argument	C & C++ location	Fortran location
r	i[0]	I(1)

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

Constructor argument	C & C++ location	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

and ni = 0, na = 2, nd = 1.

4.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 4.13 Send and receive a section of a 3D array.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
```

4.1. DERIVED DATATYPES 111

and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

Constructor argument	C & C++ location	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

Constructor argument	C & C++ location	Fortran location
r	i[0]	I(1)

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

Constructor argument	C & C++ location	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

and ni = 0, na = 2, nd = 1.

4.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 4.13 Send and receive a section of a 3D array.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
```

```
C      and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice,
                             threeslice, ierr)

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.14 Copy the (strictly) lower triangular part of a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        block(i) = 100-i
      END DO

C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED( 100, block, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                    ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.15 Transpose a matrix.

```
      REAL a(100,100), b(100,100)
```

112 CHAPTER 4. DATATYPES

```
C      and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice,
                             threeslice, ierr)

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.14 Copy the (strictly) lower triangular part of a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        block(i) = 100-i
      END DO

C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED( 100, block, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                    ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.15 Transpose a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
               MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.16 Another approach to the transpose problem:

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)
```

4.1. DERIVED DATATYPES 113

```
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
               MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.16 Another approach to the transpose problem:

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)
```

```
C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.17 We manipulate an array of structures.

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];

int                  i, dest, rank;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
MPI_Aint     base;


/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

   /* If compiler does padding in mysterious ways,
   the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
```

114 CHAPTER 4. DATATYPES

```
C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.17 We manipulate an array of structures.

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];

int                  i, dest, rank;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
MPI_Aint     base;


/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

   /* If compiler does padding in mysterious ways,
   the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i <4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);


              /* 4.1:
        send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);


              /* 4.2:
        send only the entries of class zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zdisp[j] = i;
     zblock[j] = 1;
     j++;
     }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
```

4.1. DERIVED DATATYPES 115

```
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i <4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);


              /* 4.1:
        send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);


              /* 4.2:
        send only the entries of class zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zdisp[j] = i;
     zblock[j] = 1;
     j++;
     }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);


       /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


                /* 4.3:
          send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                          the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};
```

116 CHAPTER 4. DATATYPES

```
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);


       /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


                /* 4.3:
          send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                          the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};
```

```
MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

Example 4.18 The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
   {
   int class;
   double d[6];
   char b[7];
   };

struct Partstruct particle[1000];

           /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                  /* 5.1:
            send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


                 /* 5.2:
         send the entries of class zero,
```

4.1. DERIVED DATATYPES 117

```
MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

Example 4.18 The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
   {
   int class;
   double d[6];
   char b[7];
   };

struct Partstruct particle[1000];

           /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                  /* 5.1:
            send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


                 /* 5.2:
         send the entries of class zero,
         preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint zdisp[1000]
int zblock[1000], i, j, k;
int zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

Example 4.19 Handling of unions.

```
union {
   int     ival;
   float   fval;
      } u[1000]

int     utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
```

118 CHAPTER 4. DATATYPES

```
         preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint zdisp[1000]
int zblock[1000], i, j, k;
int zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
    zdisp[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

Example 4.19 Handling of unions.

```
union {
   int     ival;
   float   fval;
      } u[1000]

int     utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
```

```
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

Example 4.20 This example shows how a datatype can be decoded. The routine printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for datatypes that are not predefined.

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;

    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
```

4.1. DERIVED DATATYPES 119

```
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

Example 4.20 This example shows how a datatype can be decoded. The routine printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for datatypes that are not predefined.

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;

    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
```

```
    case MPI_COMBINER_NAMED:
        printf( "Datatype is named:" );
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if      (datatype == MPI_INT)    printf( "MPI_INT\n" );
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
        ... else test for other types ...
        return 0;
        break;
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf( "Datatype is struct containing" );
        array_of_ints   = (int *)malloc( num_ints * sizeof(int) );
        array_of_adds   =
                   (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
        array_of_dtypes = (MPI_Datatype *)
            malloc( num_dtypes * sizeof(MPI_Datatype) );
        MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                         array_of_ints, array_of_adds, array_of_dtypes );
        printf( " %d datatypes:\n", array_of_ints[0] );
        for (i=0; i<array_of_ints[0]; i++) {
            printf( "blocklength %d, displacement %ld, type:\n",
                    array_of_ints[i+1], array_of_adds[i] );
            if (printdatatype( array_of_dtypes[i] )) {
                /* Note that we free the type ONLY if it
                   is not predefined */
                MPI_Type_free( &array_of_dtypes[i] );
            }
        }
        free( array_of_ints );
        free( array_of_adds );
        free( array_of_dtypes );
        break;
        ... other combiner values ...
    default:
        printf( "Unrecognized combiner type\n" );
    }
    return 1;
}
```

4.2 Pack and Unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending

120 CHAPTER 4. DATATYPES

```
    case MPI_COMBINER_NAMED:
        printf( "Datatype is named:" );
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if      (datatype == MPI_INT)    printf( "MPI_INT\n" );
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
        ... else test for other types ...
        return 0;
        break;
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf( "Datatype is struct containing" );
        array_of_ints   = (int *)malloc( num_ints * sizeof(int) );
        array_of_adds   =
                   (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
        array_of_dtypes = (MPI_Datatype *)
            malloc( num_dtypes * sizeof(MPI_Datatype) );
        MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                         array_of_ints, array_of_adds, array_of_dtypes );
        printf( " %d datatypes:\n", array_of_ints[0] );
        for (i=0; i<array_of_ints[0]; i++) {
            printf( "blocklength %d, displacement %ld, type:\n",
                    array_of_ints[i+1], array_of_adds[i] );
            if (printdatatype( array_of_dtypes[i] )) {
                /* Note that we free the type ONLY if it
                   is not predefined */
                MPI_Type_free( &array_of_dtypes[i] );
            }
        }
        free( array_of_ints );
        free( array_of_adds );
        free( array_of_dtypes );
        break;
        ... other combiner values ...
    default:
        printf( "Unrecognized combiner type\n" );
    }
    return 1;
}
```

4.2 Pack and Unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending

it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (non-negative integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (non-negative integer)
INOUT	position	current position in buffer, in bytes (integer)
IN	comm	communicator for packed message (handle)

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

```
void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,
             int outsize, int& position, const MPI::Comm &comm) const
```

Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer space specified by outbuf and outsize. The input buffer can be any communication buffer allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were a communication buffer for a message of type MPI_PACKED).

The input value of position is the first location in the output buffer to be used for packing. position is incremented by the size of the packed message, and the output value of position is the first location in the output buffer following the locations occupied by the packed message. The comm argument is the communicator that will be subsequently used for sending the packed message.

4.2. PACK AND UNPACK 121

it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (non-negative integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (non-negative integer)
INOUT	position	current position in buffer, in bytes (integer)
IN	comm	communicator for packed message (handle)

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

```
void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,
             int outsize, int& position, const MPI::Comm &comm) const
```

Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer space specified by outbuf and outsize. The input buffer can be any communication buffer allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were a communication buffer for a message of type MPI_PACKED).

The input value of position is the first location in the output buffer to be used for packing. position is incremented by the size of the packed message, and the output value of position is the first location in the output buffer following the locations occupied by the packed message. The comm argument is the communicator that will be subsequently used for sending the packed message.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN	inbuf	input buffer start (choice)
IN	insize	size of input buffer, in bytes (non-negative integer)
INOUT	position	current position in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of items to be unpacked (integer)
IN	datatype	datatype of each output data item (handle)
IN	comm	communicator for packed message (handle)

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
              int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
              IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

```
void MPI::Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
              int outcount, int& position, const MPI::Comm& comm) const
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

Advice to users. Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined *a priori*. (*End of advice to users.*)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

122 CHAPTER 4. DATATYPES

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN inbuf input buffer start (choice)
IN insize size of input buffer, in bytes (non-negative integer)
INOUT position current position in bytes (integer)
OUT outbuf output buffer start (choice)
IN outcount number of items to be unpacked (integer)
IN datatype datatype of each output data item (handle)
IN comm communicator for packed message (handle)

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
              int outcount, MPI_Datatype datatype, MPI_Comm comm)

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
              IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

void MPI::Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
              int outcount, int& position, const MPI::Comm& comm) const

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

Advice to users. Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined *a priori*. (*End of advice to users.*)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

> *Rationale.* The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

MPI_PACK_SIZE(incount, datatype, comm, size)

IN	incount	count argument to packing call (non-negative integer)
IN	datatype	datatype argument to packing call (handle)
IN	comm	communicator argument to packing call (handle)
OUT	size	upper bound on size of packed message, in bytes (non-negative integer)

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)
```

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

```
int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const
```

4.2. PACK AND UNPACK 123

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

Rationale. The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

MPI_PACK_SIZE(incount, datatype, comm, size)

IN incount count argument to packing call (non-negative integer)
IN datatype datatype argument to packing call (handle)
IN comm communicator argument to packing call (handle)
OUT size upper bound on size of packed message, in bytes (non-negative integer)

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR

int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const

A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm).

Rationale. The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

Example 4.21 An example using MPI_PACK.

```
int position, i, j, a[2];
char buff[1000];
....

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
   / * SENDER CODE */

  position = 0;
  MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else  /* RECEIVER CODE */
  MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)

}
```

Example 4.22 An elaborate example.

```
int position, i;
float a[1000];
char buff[1000]
....

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
  / * SENDER CODE */

  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* build datatype for i followed by a[0]...a[i-1] */

  len[0] = 1;
  len[1] = i;
```

124 CHAPTER 4. DATATYPES

A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm).

Rationale. The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

Example 4.21 An example using MPI_PACK.

```
int position, i, j, a[2];
char buff[1000];
....

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
   / * SENDER CODE */

  position = 0;
  MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else  /* RECEIVER CODE */
  MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)

}
```

Example 4.22 An elaborate example.

```
int position, i;
float a[1000];
char buff[1000]
....

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
  / * SENDER CODE */

  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* build datatype for i followed by a[0]...a[i-1] */

  len[0] = 1;
  len[1] = i;
  MPI_Address( &i, disp);
  MPI_Address( a, disp+1);
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype);
  MPI_Type_commit( &newtype);

  /* Pack i followed by a[0]...a[i-1]*/

  position = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

  /* Send */

  MPI_Send( buff, position, MPI_PACKED, 1, 0,
            MPI_COMM_WORLD)

/* *****
   One can replace the last three lines with
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else if (myrank == 1)
{
   /* RECEIVER CODE */

  MPI_Status status;

  /* Receive */

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);

  /* Unpack i */

  position = 0;
  MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

  /* Unpack a[0]...a[i-1] */
  MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

Example 4.23 Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```
int count, gsize, counts[64], totalcount, k1, k2, k,
    displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
...
```

4.2. PACK AND UNPACK 125

```
  MPI_Address( &i, disp);
  MPI_Address( a, disp+1);
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype);
  MPI_Type_commit( &newtype);

  /* Pack i followed by a[0]...a[i-1]*/

  position = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

  /* Send */

  MPI_Send( buff, position, MPI_PACKED, 1, 0,
            MPI_COMM_WORLD)

/* *****
   One can replace the last three lines with
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else if (myrank == 1)
{
   /* RECEIVER CODE */

  MPI_Status status;

  /* Receive */

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);

  /* Unpack i */

  position = 0;
  MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

  /* Unpack a[0]...a[i-1] */
  MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

Example 4.23 Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

```
int count, gsize, counts[64], totalcount, k1, k2, k,
    displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
...
```

```
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);

if (myrank != root) {
      /* gather at root sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, NULL, NULL,
             NULL, root, comm);

      /* gather at root packed messages */
   MPI_Gatherv( &buf, position, MPI_PACKED, NULL,
             NULL, NULL, NULL, root, comm);

} else {   /* root code */
      /* gather sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, counts, 1,
             MPI_INT, root, comm);

      /* gather all packed messages */
   displs[0] = 0;
   for (i=1; i < gsize; i++)
     displs[i] = displs[i-1] + counts[i-1];
   totalcount = dipls[gsize-1] + counts[gsize-1];
   rbuf = (char *)malloc(totalcount);
   cbuf = (char *)malloc(totalcount);
   MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
            counts, displs, MPI_PACKED, root, comm);

       /* unpack all messages and concatenate strings */
   concat_pos = 0;
   for (i=0; i < gsize; i++) {
      position = 0;
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, &count, 1, MPI_INT, comm);
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
      concat_pos += count;
   }
```

126 CHAPTER 4. DATATYPES

```
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);

if (myrank != root) {
      /* gather at root sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, NULL, NULL,
             NULL, root, comm);

      /* gather at root packed messages */
   MPI_Gatherv( &buf, position, MPI_PACKED, NULL,
             NULL, NULL, NULL, root, comm);

} else {   /* root code */
      /* gather sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, counts, 1,
             MPI_INT, root, comm);

      /* gather all packed messages */
   displs[0] = 0;
   for (i=1; i < gsize; i++)
     displs[i] = displs[i-1] + counts[i-1];
   totalcount = dipls[gsize-1] + counts[gsize-1];
   rbuf = (char *)malloc(totalcount);
   cbuf = (char *)malloc(totalcount);
   MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
            counts, displs, MPI_PACKED, root, comm);

       /* unpack all messages and concatenate strings */
   concat_pos = 0;
   for (i=0; i < gsize; i++) {
      position = 0;
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, &count, 1, MPI_INT, comm);
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
      concat_pos += count;
   }
```

```
    cbuf[concat_pos] = '\0';
}
```

4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is "external32."

> *Advice to users.* These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (*End of advice to users.*)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

> *Rationale.* MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (*End of rationale.*)

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN	datarep	data representation (string)
IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)

```
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)
```

```
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
             POSITION, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
             int incount, void* outbuf, MPI::Aint outsize,
             MPI::Aint& position) const
```

4.3. *CANONICAL* MPI_PACK *AND* MPI_UNPACK 127

```
    cbuf[concat_pos] = '\0';
}
```

4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is "external32."

> *Advice to users.* These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (*End of advice to users.*)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

> *Rationale.* MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (*End of rationale.*)

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN	datarep	data representation (string)
IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)

```
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)
```

```
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
             POSITION, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
             int incount, void* outbuf, MPI::Aint outsize,
             MPI::Aint& position) const
```

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN	datarep	data representation (string)
IN	inbuf	input buffer start (choice)
IN	insize	input buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of output data items (integer)
IN	datatype	datatype of output data item (handle)

```
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
             MPI_Aint *position, void *outbuf, int outcount,
             MPI_Datatype datatype)
```

```
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
             DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
             MPI::Aint insize, MPI::Aint& position, void* outbuf,
             int outcount) const
```

MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)

IN	datarep	data representation (string)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	size	output buffer size, in bytes (integer)

```
int MPI_Pack_external_size(char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)
```

```
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP
```

```
MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
             int incount) const
```

128 CHAPTER 4. DATATYPES

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN	datarep	data representation (string)
IN	inbuf	input buffer start (choice)
IN	insize	input buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of output data items (integer)
IN	datatype	datatype of output data item (handle)

```
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
             MPI_Aint *position, void *outbuf, int outcount,
             MPI_Datatype datatype)
```

```
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
             DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

```
void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
             MPI::Aint insize, MPI::Aint& position, void* outbuf,
             int outcount) const
```

MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)

IN	datarep	data representation (string)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	size	output buffer size, in bytes (integer)

```
int MPI_Pack_external_size(char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)
```

```
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP
```

```
MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
             int incount) const
```

Chapter 5

Collective Communication

5.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- MPI_BARRIER: Barrier synchronization across all members of a group (Section 5.3).
- MPI_BCAST: Broadcast from one member to all members of a group (Section 5.4). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHER, MPI_GATHERV: Gather data from all members of a group to one member (Section 5.5). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_SCATTERV: Scatter data from one member to all members of a group (Section 5.6). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_ALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 5.8). This is shown as "alltoall" in Figure 5.1.
- MPI_ALLREDUCE, MPI_REDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group and a variation where the result is returned to only one member (Section 5.9).
- MPI_REDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10).
- MPI_SCAN, MPI_EXSCAN: Scan across all members of a group (also called prefix) (Section 5.11).

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 5.2. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 4. Several collective routines such as broadcast

Chapter 5

Collective Communication

5.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- MPI_BARRIER: Barrier synchronization across all members of a group (Section 5.3).
- MPI_BCAST: Broadcast from one member to all members of a group (Section 5.4). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHER, MPI_GATHERV: Gather data from all members of a group to one member (Section 5.5). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_SCATTERV: Scatter data from one member to all members of a group (Section 5.6). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_ALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 5.8). This is shown as "alltoall" in Figure 5.1.
- MPI_ALLREDUCE, MPI_REDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group and a variation where the result is returned to only one member (Section 5.9).
- MPI_REDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10).
- MPI_SCAN, MPI_EXSCAN: Scan across all members of a group (also called prefix) (Section 5.11).

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 5.2. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 4. Several collective routines such as broadcast

129

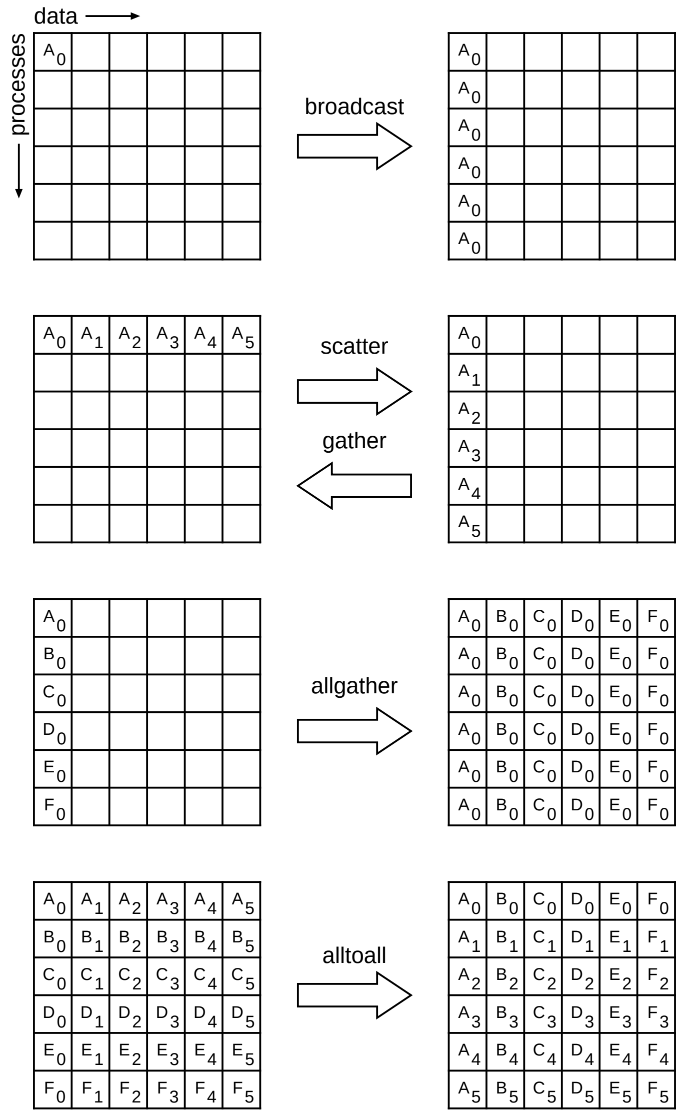


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

130 CHAPTER 5. COLLECTIVE COMMUNICATION

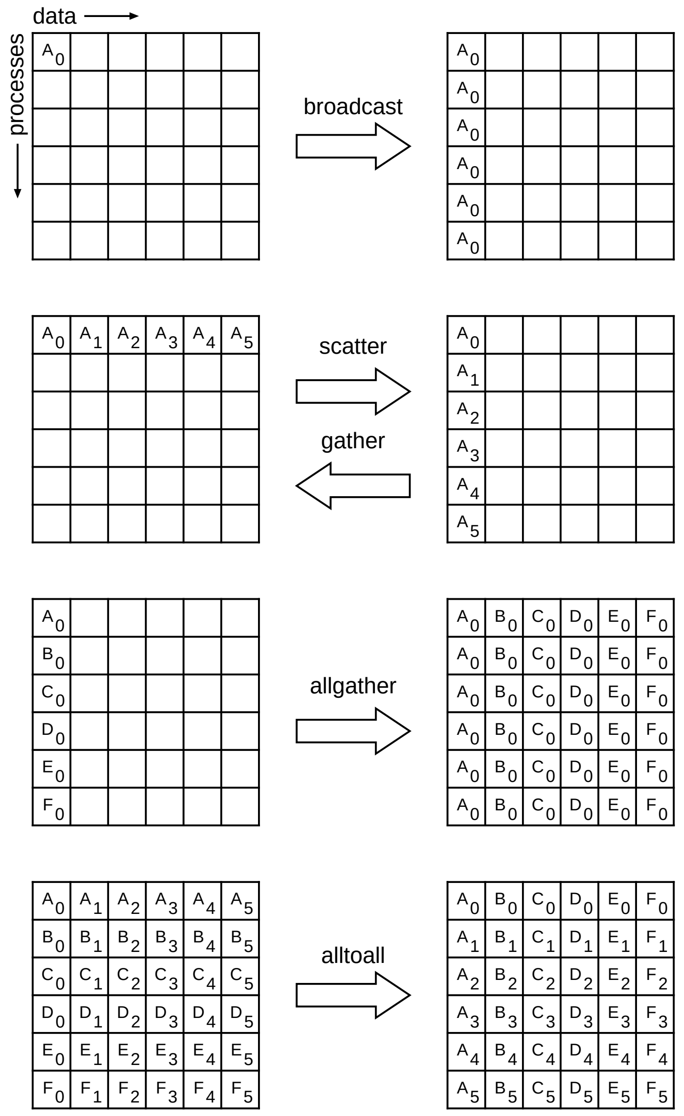


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

and gather have a single originating or receiving process. Such a process is called the *root*. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root. The reader is referred to Chapter 4 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 6 for information on how to define groups and create communicators.

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 4.1) between sender and receiver are still allowed.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. A more detailed discussion of correct use of collective routines is found in Section 5.12.

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

The collective operations do not accept a message tag argument. If future revisions of MPI define non-blocking collective functions, then tags (or a similar mechanism) might need to be added so as to allow the dis-ambiguation of multiple, pending, collective operations. (*End of rationale.*)

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 5.12. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden,

5.1. INTRODUCTION AND OVERVIEW 131

and gather have a single originating or receiving process. Such a process is called the *root*. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root. The reader is referred to Chapter 4 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 6 for information on how to define groups and create communicators.

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 4.1) between sender and receiver are still allowed.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. A more detailed discussion of correct use of collective routines is found in Section 5.12.

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

The collective operations do not accept a message tag argument. If future revisions of MPI define non-blocking collective functions, then tags (or a similar mechanism) might need to be added so as to allow the dis-ambiguation of multiple, pending, collective operations. (*End of rationale.*)

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 5.12. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden,

special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.12. (*End of advice to implementors.*)

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are *not* correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intracommunicator can be thought of as an indentifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

5.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine with matching arguments.

In many cases, collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

Rationale. The "in place" operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits aliasing of arguments; the approach of using a special value to denote "in place" operation eliminates that difficulty. (*End of rationale.*)

Advice to users. By allowing the "in place" option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has.

Some intracommunicator collective operations do not support the "in place" option (e.g., MPI_ALLTOALLV). (*End of advice to users.*)

132 CHAPTER 5. COLLECTIVE COMMUNICATION

special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.12. (*End of advice to implementors.*)

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are *not* correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intracommunicator can be thought of as an indentifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

5.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine with matching arguments.

In many cases, collective communication can occur "in place" for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

Rationale. The "in place" operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits aliasing of arguments; the approach of using a special value to denote "in place" operation eliminates that difficulty. (*End of rationale.*)

Advice to users. By allowing the "in place" option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has.

Some intracommunicator collective operations do not support the "in place" option (e.g., MPI_ALLTOALLV). (*End of advice to users.*)

5.2.2 Applying Collective Operations to Intercommunicators

To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [43]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_ALLGATHERV
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
- MPI_ALLREDUCE, MPI_REDUCE_SCATTER

All-To-One All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_GATHERV
- MPI_REDUCE

One-To-All One process contributes to the result. All processes receive the result.

- MPI_BCAST
- MPI_SCATTER, MPI_SCATTERV

Other Collective operations that do not fit into one of the above categories.

- MPI_SCAN, MPI_EXSCAN
- MPI_BARRIER

The MPI_BARRIER operation does not fit into this classification since no data is being moved (other than the implicit fact that a barrier has been called). The data movement patterns of MPI_SCAN and MPI_EXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:

- MPI_BARRIER,
- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,

5.2. COMMUNICATOR ARGUMENT 133

5.2.2 Applying Collective Operations to Intercommunicators

To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [43]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_ALLGATHERV
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
- MPI_ALLREDUCE, MPI_REDUCE_SCATTER

All-To-One All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_GATHERV
- MPI_REDUCE

One-To-All One process contributes to the result. All processes receive the result.

- MPI_BCAST
- MPI_SCATTER, MPI_SCATTERV

Other Collective operations that do not fit into one of the above categories.

- MPI_SCAN, MPI_EXSCAN
- MPI_BARRIER

The MPI_BARRIER operation does not fit into this classification since no data is being moved (other than the implicit fact that a barrier has been called). The data movement patterns of MPI_SCAN and MPI_EXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:

- MPI_BARRIER,
- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,

- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW,
- MPI_ALLREDUCE, MPI_REDUCE,
- MPI_REDUCE_SCATTER.

In C++, the bindings for these functions are in the `MPI::Comm` class. However, since the collective operations do not make sense on a C++ `MPI::Comm` (as it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual.

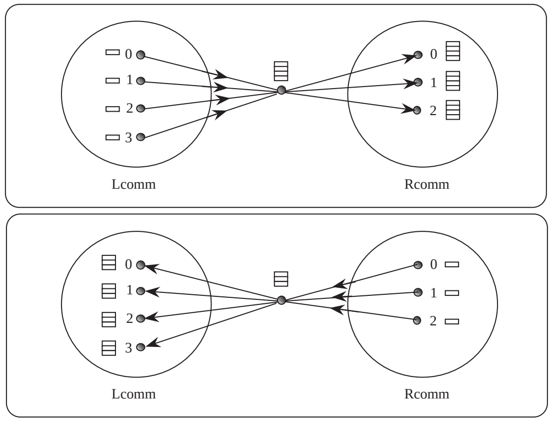


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine. In addition, processes in the same group must call the routine with matching arguments.

Note that the "in place" option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process to itself.

For intercommunicator collective communication, if the operation is rooted (e.g., broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, for the group containing the root process, all processes in the group must call the routine using a special argument for the root. For this, the root process uses the special root value MPI_ROOT; all other processes in the same group as the root use MPI_PROC_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is unrooted (e.g., alltoall), then the transfer is bidirectional.

134 CHAPTER 5. COLLECTIVE COMMUNICATION

- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW,
- MPI_ALLREDUCE, MPI_REDUCE,
- MPI_REDUCE_SCATTER.

In C++, the bindings for these functions are in the `MPI::Comm` class. However, since the collective operations do not make sense on a C++ `MPI::Comm` (as it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual.

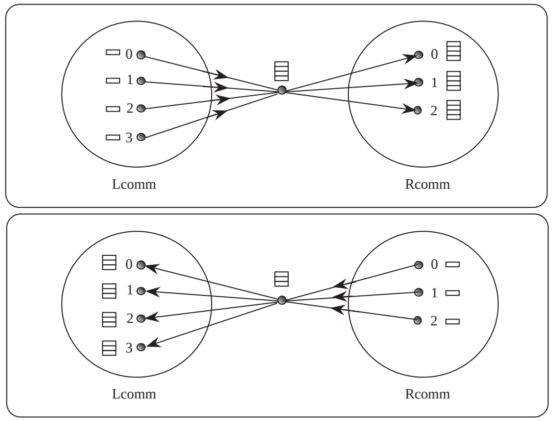


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine. In addition, processes in the same group must call the routine with matching arguments.

Note that the "in place" option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process to itself.

For intercommunicator collective communication, if the operation is rooted (e.g., broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, for the group containing the root process, all processes in the group must call the routine using a special argument for the root. For this, the root process uses the special root value MPI_ROOT; all other processes in the same group as the root use MPI_PROC_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is unrooted (e.g., alltoall), then the transfer is bidirectional.

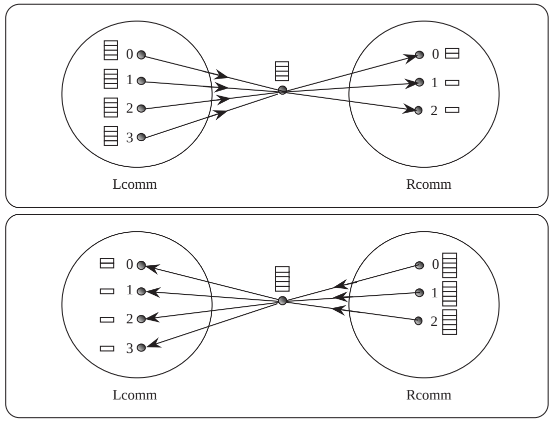


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

Rationale. Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

5.3 Barrier Synchronization

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
int MPI_Barrier(MPI_Comm comm )
```

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

```
void MPI::Comm::Barrier() const = 0
```

If comm is an intracommunicator, MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in one group (group A) of the intercommunicator may exit the barrier when all of the processes in the other group (group B) have entered the barrier.

5.3. BARRIER SYNCHRONIZATION 135

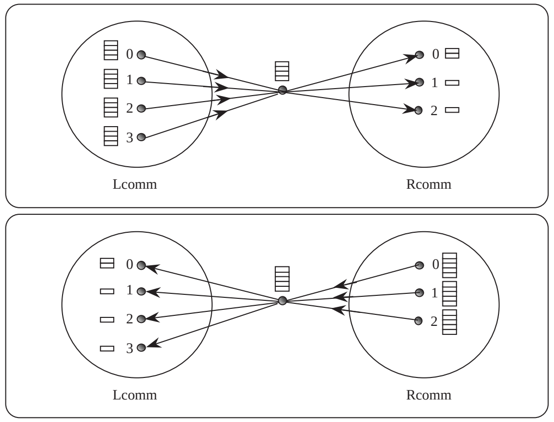


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

Rationale. Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

5.3 Barrier Synchronization

MPI_BARRIER(comm)

IN comm communicator (handle)

int MPI_Barrier(MPI_Comm comm)

MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR

void MPI::Comm::Barrier() const = 0

If comm is an intracommunicator, MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in one group (group A) of the intercommunicator may exit the barrier when all of the processes in the other group (group B) have entered the barrier.

5.4 Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm )
```

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

```
void MPI::Comm::Bcast(void* buffer, int count,
             const MPI::Datatype& datatype, int root) const = 0
```

If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root's buffer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1 Broadcast 100 ints from process 0 to every process in the group.

```
    MPI_Comm comm;
    int array[100];
```

136 CHAPTER 5. COLLECTIVE COMMUNICATION

5.4 Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT buffer starting address of buffer (choice)
IN count number of entries in buffer (non-negative integer)
IN datatype data type of buffer (handle)
IN root rank of broadcast root (integer)
IN comm communicator (handle)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm)

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

void MPI::Comm::Bcast(void* buffer, int count,
             const MPI::Datatype& datatype, int root) const = 0

If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root's buffer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1 Broadcast 100 ints from process 0 to every process in the group.

    MPI_Comm comm;
    int array[100];

```
    int root=0;
    ...
    MPI_Bcast( array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as `comm` in the above) have been assigned appropriate values.

5.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)
```

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

```
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf, int recvcount,
              const MPI::Datatype& recvtype, int root) const = 0
```

If comm is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is *as if* each of the `n` processes in the group (including the root process) had executed a call to

$$\mathtt{MPI\_Send}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{root}, ...),$$

and the root had executed `n` calls to

$$\mathtt{MPI\_Recv}(\mathtt{recvbuf} + \mathtt{i} \cdot \mathtt{recvcount} \cdot \mathtt{extent}(\mathtt{recvtype}), \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{i}, ...),$$

5.5. GATHER 137

```
    int root=0;
    ...
    MPI_Bcast( array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as `comm` in the above) have been assigned appropriate values.

5.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice, significant only at root)
IN recvcount number of elements for any single receive (non-negative integer, significant only at root)
IN recvtype data type of recv buffer elements (significant only at root) (handle)
IN root rank of receiving process (integer)
IN comm communicator (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)
```

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

```
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf, int recvcount,
              const MPI::Datatype& recvtype, int root) const = 0
```

If comm is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is *as if* each of the `n` processes in the group (including the root process) had executed a call to

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

and the root had executed `n` calls to

MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...),

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

An alternative description is that the `n` messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount·n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on each process must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the recvcount argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)

138 CHAPTER 5. COLLECTIVE COMMUNICATION

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

An alternative description is that the `n` messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount·n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on each process must be equal to the type signature of recvcount, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the recvcount argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice, significant only at root)

IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int *recvcounts, int *displs,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
```

```
void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs.

If comm is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

$$\mathtt{MPI\_Send}(\mathsf{sendbuf}, \mathsf{sendcount}, \mathsf{sendtype}, \mathsf{root}, ...),$$

and the root executes n receives,

$$\mathtt{MPI\_Recv}(\mathsf{recvbuf} + \mathsf{displs}[\mathsf{j}] \cdot \mathsf{extent}(\mathsf{recvtype}), \mathsf{recvcounts}[\mathsf{j}], \mathsf{recvtype}, \mathtt{i}, ...).$$

The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype).

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

5.5. GATHER 139

IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int *recvcounts, int *displs,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
```

```
void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs.

If comm is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

$$\mathtt{MPI\_Send}(\mathsf{sendbuf}, \mathsf{sendcount}, \mathsf{sendtype}, \mathsf{root}, ...),$$

and the root executes n receives,

$$\mathtt{MPI\_Recv}(\mathsf{recvbuf} + \mathsf{displs}[\mathsf{j}] \cdot \mathsf{extent}(\mathsf{recvtype}), \mathsf{recvcounts}[\mathsf{j}], \mathsf{recvtype}, \mathtt{i}, ...).$$

The data received from process j is placed into recvbuf of the root process beginning at offset displs[j] elements (in terms of the recvtype).

The receive buffer is ignored for all non-root processes.

The type signature implied by sendcount, sendtype on process i must be equal to the type signature implied by recvcounts[i], recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

All arguments to the function are significant on process root, while on other processes, only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

The “in place” option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

Example 5.2 Gather 100 `int`s from every process in group to root. See figure 5.4.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int root, *rbuf;
    ...
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
    MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.3 Previous example modified – only the root allocates memory for the receive buffer.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int root, myrank, *rbuf;
    ...
    MPI_Comm_rank( comm, &myrank);
    if ( myrank == root) {
       MPI_Comm_size( comm, &gsize);
       rbuf = (int *)malloc(gsize*100*sizeof(int));
    }
    MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.4 Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of `gsize*100 int`s since type matching is defined pairwise between the root and each process in the gather.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
```

140 CHAPTER 5. COLLECTIVE COMMUNICATION

The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

The “in place” option for intracommunicators is specified by passing MPI_IN_PLACE as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

Example 5.2 Gather 100 `int`s from every process in group to root. See figure 5.4.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int root, *rbuf;
    ...
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
    MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.3 Previous example modified – only the root allocates memory for the receive buffer.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int root, myrank, *rbuf;
    ...
    MPI_Comm_rank( comm, &myrank);
    if ( myrank == root) {
       MPI_Comm_size( comm, &gsize);
       rbuf = (int *)malloc(gsize*100*sizeof(int));
    }
    MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.4 Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of `gsize*100 int`s since type matching is defined pairwise between the root and each process in the gather.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
```

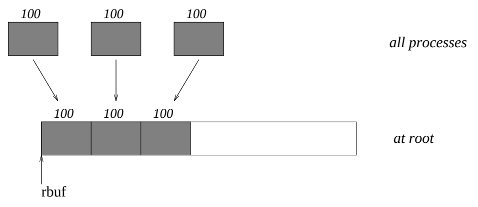


Figure 5.4: The root process gathers 100 `int`s from each process in the group.

```
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 5.5 Now have each process send 100 `int`s to root, but place each set (of 100) `stride int`s apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume $stride \geq 100$. See Figure 5.5.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

Note that the program is erroneous if $stride < 100$.

Example 5.6 Same as Example 5.5 on the receiving side, but send the 100 `int`s from the 0th column of a 100×150 `int` array, in C. See Figure 5.6.

```
MPI_Comm comm;
```

5.5. GATHER 141

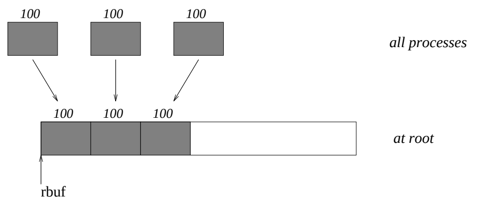


Figure 5.4: The root process gathers 100 `int`s from each process in the group.

```
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 5.5 Now have each process send 100 `int`s to root, but place each set (of 100) `stride int`s apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume $stride \geq 100$. See Figure 5.5.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

Note that the program is erroneous if $stride < 100$.

Example 5.6 Same as Example 5.5 on the receiving side, but send the 100 `int`s from the 0th column of a 100×150 `int` array, in C. See Figure 5.6.

```
MPI_Comm comm;
```

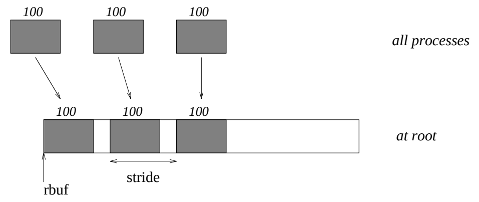


Figure 5.5: The root process gathers 100 `int`s from each process in the group, each set is placed `stride` `int`s apart.

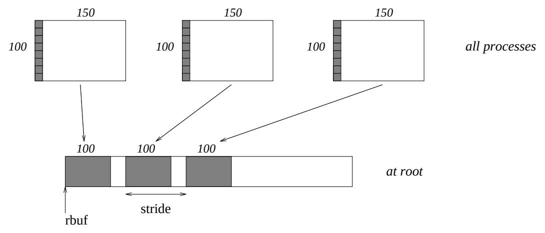


Figure 5.6: The root process gathers column `0` of a 100×150 C array, and each set is placed `stride` `int`s apart.

```
    int gsize,sendarray[100][150];
    int root, *rbuf, stride;
    MPI_Datatype stype;
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
    /* Create datatype for 1 column of array
     */
    MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

Example 5.7 Process `i` sends `(100-i)` `int`s from the `i`-th column of a 100 × 150 `int`

142 CHAPTER 5. COLLECTIVE COMMUNICATION

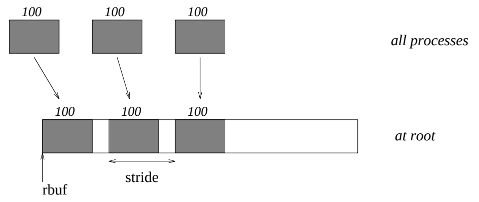


Figure 5.5: The root process gathers 100 `int`s from each process in the group, each set is placed `stride` `int`s apart.

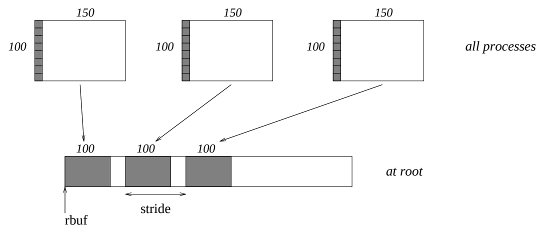


Figure 5.6: The root process gathers column `0` of a 100×150 C array, and each set is placed `stride` `int`s apart.

```
    int gsize,sendarray[100][150];
    int root, *rbuf, stride;
    MPI_Datatype stype;
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
    /* Create datatype for 1 column of array
     */
    MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

Example 5.7 Process `i` sends `(100-i)` `int`s from the `i`-th column of a 100 × 150 `int`

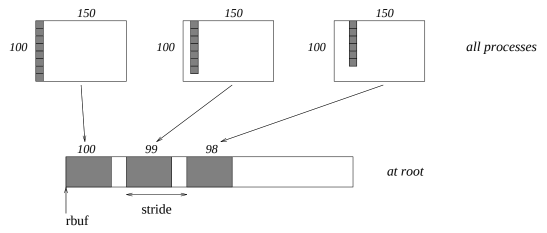


Figure 5.7: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride` `int`s apart.

array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;     /* note change from previous example */
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

Note that a different amount of data is received from each process.

Example 5.8 Same as Example 5.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 4.16, Section 4.1.14.

5.5. GATHER 143

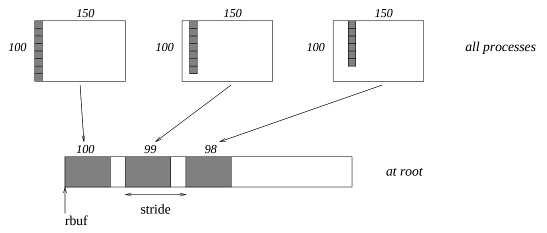


Figure 5.7: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride` `int`s apart.

array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;     /* note change from previous example */
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

Note that a different amount of data is received from each process.

Example 5.8 Same as Example 5.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 4.16, Section 4.1.14.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank, disp[2], blocklen[2];
    MPI_Datatype stype,type[2];
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;
    }
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                             root, comm);
```

Example 5.9 Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, *stride, myrank, bufsize;
    MPI_Datatype stype;
    int *displs,i,*rcounts,offset;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    stride = (int *)malloc(gsize*sizeof(int));
    ...
    /* stride[i] for i = 0 to gsize-1 is set somehow
     */
```

144 CHAPTER 5. COLLECTIVE COMMUNICATION

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank, disp[2], blocklen[2];
    MPI_Datatype stype,type[2];
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;
    }
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                             root, comm);
```

Example 5.9 Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, *stride, myrank, bufsize;
    MPI_Datatype stype;
    int *displs,i,*rcounts,offset;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    stride = (int *)malloc(gsize*sizeof(int));
    ...
    /* stride[i] for i = 0 to gsize-1 is set somehow
     */
```

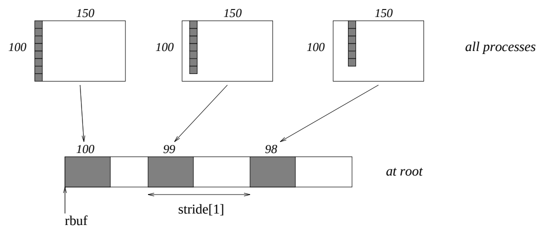


Figure 5.8: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride[i]` `int`s apart (a varying stride).

```
    /* set up displs and rcounts vectors first
     */
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    offset = 0;
    for (i=0; i<gsize; ++i) {
        displs[i] = offset;
        offset += stride[i];
        rcounts[i] = 100-i;
    }
    /* the required buffer size for rbuf is now easily obtained
     */
    bufsize = displs[gsize-1]+rcounts[gsize-1];
    rbuf = (int *)malloc(bufsize*sizeof(int));
    /* Create datatype for the column we are sending
     */
    MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                        root, comm);
```

Example 5.10 Process `i` sends `num` `int`s from the `i`-th column of a 100 × 150 `int` array, in C. The complicating factor is that the various values of `num` are not known to `root`, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank, disp[2], blocklen[2];
    MPI_Datatype stype,types[2];
    int *displs,i,*rcounts,num;

    ...
```

5.5. GATHER 145

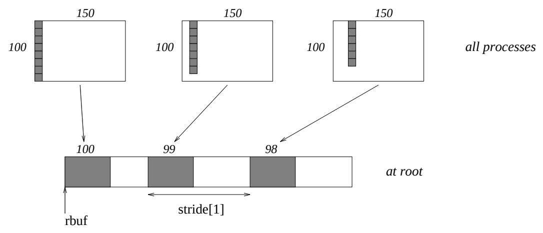


Figure 5.8: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride[i]` `int`s apart (a varying stride).

```
    /* set up displs and rcounts vectors first
     */
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    offset = 0;
    for (i=0; i<gsize; ++i) {
        displs[i] = offset;
        offset += stride[i];
        rcounts[i] = 100-i;
    }
    /* the required buffer size for rbuf is now easily obtained
     */
    bufsize = displs[gsize-1]+rcounts[gsize-1];
    rbuf = (int *)malloc(bufsize*sizeof(int));
    /* Create datatype for the column we are sending
     */
    MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                        root, comm);
```

Example 5.10 Process `i` sends `num` `int`s from the `i`-th column of a 100 × 150 `int` array, in C. The complicating factor is that the various values of `num` are not known to `root`, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank, disp[2], blocklen[2];
    MPI_Datatype stype,types[2];
    int *displs,i,*rcounts,num;

    ...
    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    /* First, gather nums to root
     */
    rcounts = (int *)malloc(gsize*sizeof(int));
    MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
    /* root now has correct rcounts, using these we set displs[] so
     * that data is placed contiguously (or concatenated) at receive end
     */
    displs = (int *)malloc(gsize*sizeof(int));
    displs[0] = 0;
    for (i=1; i<gsize; ++i) {
        displs[i] = displs[i-1]+rcounts[i-1];
    }
    /* And, create receive buffer
     */
    rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                                                          *sizeof(int));
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

146 CHAPTER 5. COLLECTIVE COMMUNICATION

```
    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    /* First, gather nums to root
     */
    rcounts = (int *)malloc(gsize*sizeof(int));
    MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
    /* root now has correct rcounts, using these we set displs[] so
     * that data is placed contiguously (or concatenated) at receive end
     */
    displs = (int *)malloc(gsize*sizeof(int));
    displs[0] = 0;
    for (i=1; i<gsize; ++i) {
        displs[i] = displs[i-1]+rcounts[i-1];
    }
    /* And, create receive buffer
     */
    rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                                                          *sizeof(int));
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                            root, comm);
```

5.6 Scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm)
```

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

```
void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

If comm is an intracommunicator, the outcome is *as if* the root executed n send operations,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount·n, sendtype, ...). This message is split into n equal segments, the i-th segment is sent to the i-th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

5.6. SCATTER 147

5.6 Scatter

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN sendbuf address of send buffer (choice, significant only at root)
IN sendcount number of elements sent to each process (non-negative integer, significant only at root)
IN sendtype data type of send buffer elements (significant only at root) (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements in receive buffer (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN root rank of sending process (integer)
IN comm communicator (handle)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0

MPI_SCATTER is the inverse operation to MPI_GATHER.

If comm is an intracommunicator, the outcome is *as if* the root executed n send operations,

MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i, ...),

and each process executed a receive,

MPI_Recv(recvbuf, recvcount, recvtype, i, ...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount·n, sendtype, ...). This message is split into n equal segments, the i-th segment is sent to the i-th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

> *Rationale.* Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to sendbuf from which to take the outgoing data to process `i`
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
```

148 CHAPTER 5. COLLECTIVE COMMUNICATION

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN sendbuf — address of send buffer (choice, significant only at root)
IN sendcounts — non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN displs — integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
IN sendtype — data type of send buffer elements (handle)
OUT recvbuf — address of receive buffer (choice)
IN recvcount — number of elements in receive buffer (non-negative integer)
IN recvtype — data type of receive buffer elements (handle)
IN root — rank of sending process (integer)
IN comm — communicator (handle)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

```
void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], const MPI::Datatype& sendtype,
             void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
             int root) const = 0
```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, displs.

If comm is an intracommunicator, the outcome is as if the root executed n send operations,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

The examples in this section use intracommunicators.

Example 5.11 The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.

5.6. SCATTER 149

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

```
void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], const MPI::Datatype& sendtype,
             void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
             int root) const = 0
```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, displs.

If comm is an intracommunicator, the outcome is as if the root executed n send operations,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts, types, and displacements should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

The examples in this section use intracommunicators.

Example 5.11 The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.

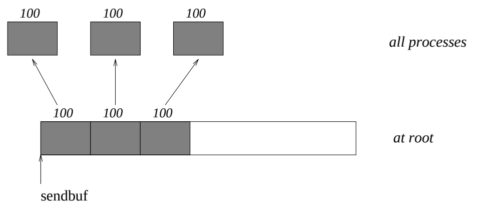


Figure 5.9: The root process scatters sets of 100 `int`s to each process in the group.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.12 The reverse of Example 5.5. The root process scatters sets of 100 `int`s to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI_SCATTERV. Assume *stride* $\geq$ 100. See Figure 5.10.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                                                           root, comm);
```

Example 5.13 The reverse of Example 5.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the `i`-th column of a 100×150 C array. See Figure 5.11.

```
MPI_Comm comm;
```

150 CHAPTER 5. COLLECTIVE COMMUNICATION

Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

Example 5.12 The reverse of Example 5.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are stride ints apart in the sending buffer. Requires use of MPI_SCATTERV. Assume stride ≥ 100. See Figure 5.10.

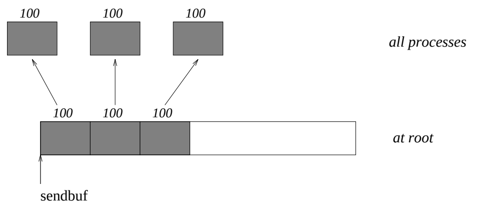
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                                                           root, comm);

Example 5.13 The reverse of Example 5.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the i-th column of a 100×150 C array. See Figure 5.11.

MPI_Comm comm;



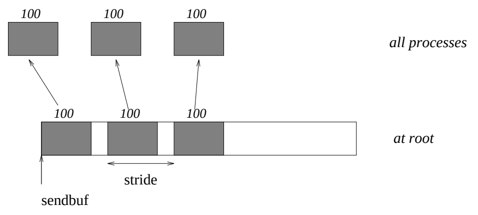


Figure 5.10: The root process scatters sets of 100 `int`s, moving by `stride` `int`s from send to send in the scatter.

```
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                          root, comm);
```

5.6. SCATTER 151

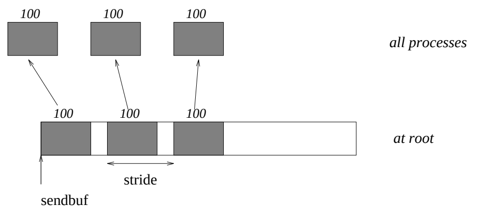


Figure 5.10: The root process scatters sets of 100 `int`s, moving by `stride` `int`s from send to send in the scatter.

```
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                          root, comm);
```

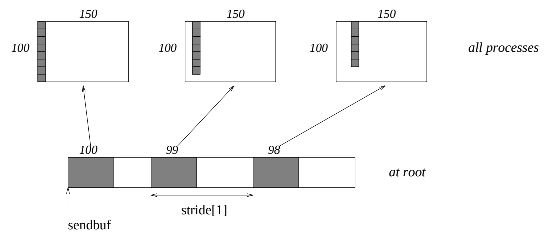


Figure 5.11: The root scatters blocks of `100-i` `int`s into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` `int`s apart.

5.7 Gather-to-all

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)
```

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

```
void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the `j`-th process is received by every process and placed in the `j`-th block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.

152 CHAPTER 5. COLLECTIVE COMMUNICATION

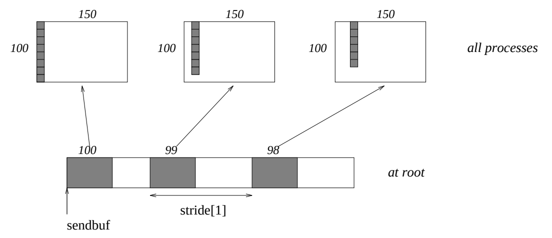


Figure 5.11: The root scatters blocks of `100-i` `int`s into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` `int`s apart.

5.7 Gather-to-all

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements received from any process (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.

If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to

```
MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                                             recvtype,root,comm),
```

for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount $= 0$ for the communication in the reverse direction.

(*End of advice to users.*)

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
```

5.7. GATHER-TO-ALL 153

If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to

```
MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                                             recvtype,root,comm),
```

for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount $= 0$ for the communication in the reverse direction.

(*End of advice to users.*)

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
              RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

```
void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf,
              const int recvcounts[], const int displs[],
              const MPI::Datatype& recvtype) const = 0
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

```
    MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                               recvtype,root,comm),
```

for `root = 0 , ..., n-1`. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV

The examples in this section use intracommunicators.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int *rbuf;
    ...
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
    MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.

154 CHAPTER 5. COLLECTIVE COMMUNICATION

```
              MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
              RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

```
void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf,
              const int recvcounts[], const int displs[],
              const MPI::Datatype& recvtype) const = 0
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

```
    MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                               recvtype,root,comm),
```

for `root = 0 , ..., n-1`. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV

The examples in this section use intracommunicators.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```
    MPI_Comm comm;
    int gsize,sendarray[100];
    int *rbuf;
    ...
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
    MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.

5.8 All-to-All Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)
```

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

```
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

$$\mathtt{MPI\_Send}(\mathtt{sendbuf} + \mathtt{i} \cdot \mathtt{sendcount} \cdot \mathtt{extent}(\mathtt{sendtype}), \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{i}, ...),$$

and a receive from every other process with a call to,

$$\mathtt{MPI\_Recv}(\mathtt{recvbuf} + \mathtt{i} \cdot \mathtt{recvcount} \cdot \mathtt{extent}(\mathtt{recvtype}), \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{i}, ...).$$

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process

5.8. ALL-TO-ALL SCATTER/GATHER 155

5.8 All-to-All Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements sent to each process (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements received from any process (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i, ...),

and a receive from every other process with a call to,

MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...).

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process

i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

> *Advice to users.* When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction.
>
> (*End of advice to users.*)

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
             MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
             int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

```
void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype& sendtype,
             void* recvbuf, const int recvcounts[], const int rdispls[],
             const MPI::Datatype& recvtype) const = 0
```

156 CHAPTER 5. COLLECTIVE COMMUNICATION

i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

> *Advice to users.* When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying sendcount = 0 in the reverse direction.
>
> (*End of advice to users.*)

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array equal to the group size specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array equal to the group size specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
             MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
             int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

```
void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype& sendtype,
             void* recvbuf, const int recvcounts[], const int rdispls[],
             const MPI::Datatype& recvtype) const = 0
```

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcount[j], sendtype at process i must be equal to the type signature associated with recvcount[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

and received a message from every other process with a call to

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\texttt{i}], \texttt{recvtype}, \texttt{i}, ...).$$

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No "in place" option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

> *Rationale.* The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

> *Advice to implementors.* Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

5.8. ALL-TO-ALL SCATTER/GATHER 157

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcount[j], sendtype at process i must be equal to the type signature associated with recvcount[i], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

and received a message from every other process with a call to

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\texttt{i}], \texttt{recvtype}, \texttt{i}, ...).$$

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No "in place" option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor (array of non-negative integers)
IN	sdispls	integer array (of length group size). Entry `j` specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process `j` (array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry `j` specifies the type of data to send to process `j` (array of handles)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array equal to the group size specifying the number of elements that can be received from each processor (array of non-negative integers)
IN	rdispls	integer array (of length group size). Entry `i` specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process `i` (array of integers)
IN	recvtypes	array of datatypes (of length group size). Entry `i` specifies the type of data received from process `i` (array of handles)
IN	comm	communicator (handle)

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
             MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
             int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR
```

```
void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype sendtypes[], void*
             recvbuf, const int recvcounts[], const int rdispls[], const
             MPI::Datatype recvtypes[]) const = 0
```

MPI_ALLTOALLW is the most general form of `All-to-all`. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

158 CHAPTER 5. COLLECTIVE COMMUNICATION

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcounts integer array equal to the group size specifying the number of elements to send to each processor (array of non-negative integers)
IN sdispls integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
IN sendtypes array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
OUT recvbuf address of receive buffer (choice)
IN recvcounts integer array equal to the group size specifying the number of elements that can be received from each processor (array of non-negative integers)
IN rdispls integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
IN recvtypes array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
IN comm communicator (handle)

int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
             MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
             int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR

void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype sendtypes[], void*
             recvbuf, const int recvcounts[], const int rdispls[], const
             MPI::Datatype recvtypes[]) const = 0

MPI_ALLTOALLW is the most general form of All-to-all. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{sdispls}[\texttt{i}], \texttt{sendcounts}[\texttt{i}], \texttt{sendtypes}[\texttt{i}], \texttt{i}, ...),$$

and received a message from every other process with a call to

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{rdispls}[\texttt{i}], \texttt{recvcounts}[\texttt{i}], \texttt{recvtypes}[\texttt{i}], \texttt{i}, ...).$$

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have `sendcounts[i] = 0`, this achieves an `MPI_SCATTERW` function. (*End of rationale.*)

5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

5.9. GLOBAL REDUCTION OPERATIONS 159

If comm is an intracommunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with

MPI_Send(sendbuf + sdispls[i], sendcounts[i], sendtypes[i], i, ...),

and received a message from every other process with a call to

MPI_Recv(recvbuf + rdispls[i], recvcounts[i], recvtypes[i], i, ...).

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (End of rationale.)

5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

5.9.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

```
void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op, int root)
             const = 0
```

If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then $\mathsf{recvbuf}(1) = \text{global}\max(\mathsf{sendbuf}(1))$ and $\mathsf{recvbuf}(2) = \text{global}\max(\mathsf{sendbuf}(2))$.

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes each operation can be applied to. In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation.

160 CHAPTER 5. COLLECTIVE COMMUNICATION

5.9.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN sendbuf address of send buffer (choice)
OUT recvbuf address of receive buffer (choice, significant only at root)
IN count number of elements in send buffer (non-negative integer)
IN datatype data type of elements of send buffer (handle)
IN op reduce operation (handle)
IN root rank of root process (integer)
IN comm communicator (handle)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op, int root)
             const = 0

If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then $\mathsf{recvbuf}(1) = \text{global}\max(\mathsf{sendbuf}(1))$ and $\mathsf{recvbuf}(2) = \text{global}\max(\mathsf{sendbuf}(2))$.

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes each operation can be applied to. In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation.

This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

> *Advice to implementors.* It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (*End of advice to implementors.*)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

> *Advice to users.* Users should make no assumptions about how MPI_REDUCE is implemented. Safest is to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and

5.9. GLOBAL REDUCTION OPERATIONS 161

This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (*End of advice to implementors.*)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. Safest is to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

Name Meaning
MPI_MAX maximum
MPI_MIN minimum
MPI_SUM sum
MPI_PROD product
MPI_LAND logical and
MPI_BAND bit-wise and

MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR
Fortran integer:	MPI_INTEGER
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

Now, the valid datatypes for each option is specified below.

Op	Allowed Types
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

The following examples use intracommunicators.

Example 5.15 A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m)       ! local slice of array
REAL c                ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr

! local sum
sum = 0.0
```

162 CHAPTER 5. COLLECTIVE COMMUNICATION

MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR
Fortran integer:	MPI_INTEGER
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

Now, the valid datatypes for each option is specified below.

Op	Allowed Types
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

The following examples use intracommunicators.

Example 5.15 A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m)       ! local slice of array
REAL c                ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr

! local sum
sum = 0.0
```

```
DO i = 1, m
   sum = sum + a(i)*b(i)
END DO

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN
```

Example 5.16 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! return result at node zero (and garbage at the other nodes)
RETURN
```

5.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR (which represents printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

> *Advice to users.* The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

5.9. GLOBAL REDUCTION OPERATIONS 163

```
DO i = 1, m
   sum = sum + a(i)*b(i)
END DO

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN
```

Example 5.16 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at node zero.

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! return result at node zero (and garbage at the other nodes)
RETURN
```

5.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR (which represents printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

5.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. Thus, if each process supplies a value and its rank within the group, then a reduce operation with op = MPI_MAXLOC will return the maximum value and the rank of the first process with that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered according to the first component of each pair, and ties are resolved according to the second component.

The reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, for Fortran, by having the MPI-provided type consist of a pair of the same type as value, and coercing the index to this type also. In C, the MPI-provided pair type has distinct types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide a datatype argument that represents a pair (value and index). MPI provides nine such

164 CHAPTER 5. COLLECTIVE COMMUNICATION

5.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$, then the value returned is (u, r), where $u = \max_i u_i$ and r is the index of the first global maximum in the sequence. Thus, if each process supplies a value and its rank within the group, then a reduce operation with op = MPI_MAXLOC will return the maximum value and the rank of the first process with that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered according to the first component of each pair, and ties are resolved according to the second component.

The reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, for Fortran, by having the MPI-provided type consist of a pair of the same type as value, and coercing the index to this type also. In C, the MPI-provided pair type has distinct types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide a datatype argument that represents a pair (value and index). MPI provides nine such

predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with each of the following datatypes.

Fortran:

Name	Description
MPI_2REAL	pair of REALs
MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables
MPI_2INTEGER	pair of INTEGERs

C:

Name	Description
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

The datatype MPI_2REAL is *as if* defined by the following (see Section 4.1).

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.

The datatype MPI_FLOAT_INT is *as if* defined by the following sequence of instructions.

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

Example 5.17 Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
    ...
    /* each process has an array of 30 double: ain[30]
     */
    double ain[30], aout[30];
    int  ind[30];
    struct {
        double val;
        int   rank;
    } in[30], out[30];
    int i, myrank, root;
```

5.9. GLOBAL REDUCTION OPERATIONS 165

predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with each of the following datatypes.

Fortran:

Name	Description
MPI_2REAL	pair of REALs
MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables
MPI_2INTEGER	pair of INTEGERs

C:

Name	Description
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

The datatype MPI_2REAL is *as if* defined by the following (see Section 4.1).

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.

The datatype MPI_FLOAT_INT is *as if* defined by the following sequence of instructions.

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

Example 5.17 Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
    ...
    /* each process has an array of 30 double: ain[30]
     */
    double ain[30], aout[30];
    int  ind[30];
    struct {
        double val;
        int   rank;
    } in[30], out[30];
    int i, myrank, root;
    MPI_Comm_rank(comm, &myrank);
    for (i=0; i<30; ++i) {
        in[i].val = ain[i];
        in[i].rank = myrank;
    }
    MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
    /* At this point, the answer resides on process root
     */
    if (myrank == root) {
        /* read ranks out
         */
        for (i=0; i<30; ++i) {
            aout[i] = out[i].val;
            ind[i] = out[i].rank;
        }
    }
```

Example 5.18 Same example, in Fortran.

```
    ...
    ! each process has an array of 30 double: ain(30)

    DOUBLE PRECISION ain(30), aout(30)
    INTEGER ind(30)
    DOUBLE PRECISION in(2,30), out(2,30)
    INTEGER i, myrank, root, ierr

    CALL MPI_COMM_RANK(comm, myrank, ierr)
    DO I=1, 30
        in(1,i) = ain(i)
        in(2,i) = myrank    ! myrank is coerced to a double
    END DO

    CALL MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                              comm, ierr )
    ! At this point, the answer resides on process root

    IF (myrank .EQ. root) THEN
        ! read ranks out
        DO I= 1, 30
            aout(i) = out(1,i)
            ind(i) = out(2,i)  ! rank is coerced back to an integer
        END DO
    END IF
```

Example 5.19 Each process has a non-empty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.

166 CHAPTER 5. COLLECTIVE COMMUNICATION

```
    MPI_Comm_rank(comm, &myrank);
    for (i=0; i<30; ++i) {
        in[i].val = ain[i];
        in[i].rank = myrank;
    }
    MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
    /* At this point, the answer resides on process root
     */
    if (myrank == root) {
        /* read ranks out
         */
        for (i=0; i<30; ++i) {
            aout[i] = out[i].val;
            ind[i] = out[i].rank;
        }
    }
```

Example 5.18 Same example, in Fortran.

```
    ...
    ! each process has an array of 30 double: ain(30)

    DOUBLE PRECISION ain(30), aout(30)
    INTEGER ind(30)
    DOUBLE PRECISION in(2,30), out(2,30)
    INTEGER i, myrank, root, ierr

    CALL MPI_COMM_RANK(comm, myrank, ierr)
    DO I=1, 30
        in(1,i) = ain(i)
        in(2,i) = myrank    ! myrank is coerced to a double
    END DO

    CALL MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                              comm, ierr )
    ! At this point, the answer resides on process root

    IF (myrank .EQ. root) THEN
        ! read ranks out
        DO I= 1, 30
            aout(i) = out(1,i)
            ind(i) = out(2,i)  ! rank is coerced back to an integer
        END DO
    END IF
```

Example 5.19 Each process has a non-empty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.

```
#define  LEN   1000

float val[LEN];        /* local array of values */
int count;             /* local number of values */
int myrank, minrank, minindex;
float minval;

struct {
    float value;
    int   index;
} in, out;

    /* local minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
    /* At this point, the answer resides on process root
     */
if (myrank == root) {
    /* read answer out
     */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

5.9. GLOBAL REDUCTION OPERATIONS 167

```
#define  LEN   1000

float val[LEN];        /* local array of values */
int count;             /* local number of values */
int myrank, minrank, minindex;
float minval;

struct {
    float value;
    int   index;
} in, out;

    /* local minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
    /* At this point, the answer resides on process root
     */
if (myrank == root) {
    /* read answer out
     */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(function, commute, op)

IN	function	user defined function (function)
IN	commute	`true` if commutative; `false` otherwise.
OUT	op	operation (handle)

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
```

```
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

```
void MPI::Op::Init(MPI::User_function* function, bool commute)
```

MPI_OP_CREATE binds a user-defined global operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative. If commute = `true`, then the operation should be both commutative and associative. If commute = `false`, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = `true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

function is the user-defined function, which must have the following four arguments: invec, inoutvec, len and datatype.

The ISO C prototype for the function is the following.

```
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
             MPI_Datatype *datatype);
```

The Fortran declaration of the user-defined function appears below.

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, TYPE
```

The C++ declaration of the user-defined function appears below.

```
typedef void MPI::User_function(const void* invec, void *inoutvec, int len,
             const Datatype& datatype);
```

The datatype argument is a handle to the data type that was passed into the call to MPI_REDUCE. The user reduce function should be written such that the following holds: Let u[0], ... , u[len-1] be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let v[0], ... , v[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0], ... , w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then w[i] = u[i]∘v[i], for i=0 , ... , len-1, where ∘ is the reduce operation that the function computes.

168 CHAPTER 5. COLLECTIVE COMMUNICATION

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(function, commute, op)

IN function user defined function (function)
IN commute true if commutative; false otherwise.
OUT op operation (handle)

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR

void MPI::Op::Init(MPI::User_function* function, bool commute)

MPI_OP_CREATE binds a user-defined global operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

function is the user-defined function, which must have the following four arguments: invec, inoutvec, len and datatype.

The ISO C prototype for the function is the following.
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
             MPI_Datatype *datatype);

The Fortran declaration of the user-defined function appears below.
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, TYPE

The C++ declaration of the user-defined function appears below.
typedef void MPI::User_function(const void* invec, void *inoutvec, int len,
             const Datatype& datatype);

The datatype argument is a handle to the data type that was passed into the call to MPI_REDUCE. The user reduce function should be written such that the following holds: Let u[0], ... , u[len-1] be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let v[0], ... , v[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0], ... , w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then w[i] = u[i]∘v[i], for i=0 , ... , len-1, where ∘ is the reduce operation that the function computes.

Informally, we can think of invec and inoutvec as arrays of len elements that function is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: I.e, the function returns in inoutvec[i] the value invec[i] $\circ$ inoutvec[i], for $\mathsf{i} = 0, \ldots, \mathsf{count} - 1$, where $\circ$ is the combining operation computed by the function.

Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE not supporting the "in place" option.

```
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
```

5.9. GLOBAL REDUCTION OPERATIONS 169

Informally, we can think of invec and inoutvec as arrays of len elements that function is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: I.e, the function returns in inoutvec[i] the value invec[i] $\circ$ inoutvec[i], for $\mathsf{i} = 0, \ldots, \mathsf{count} - 1$, where $\circ$ is the combining operation computed by the function.

Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

Advice to implementors. We outline below a naive and inefficient implementation of MPI_REDUCE not supporting the "in place" option.

```
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
```

```
     */
    if (rank == root) {
        MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
    }
    if (rank == groupsize-1) {
        MPI_Send(sendbuf, count, datatype, root, ...);
    }
    if (rank == root) {
        MPI_Wait(&req, &status);
    }
```

The reduction computation proceeds, sequentially, from process `0` to process `groupsize-1`. This order is chosen so as to respect the order of a possibly non-commutative operator defined by the function `User_reduce()`. A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size `len` <`count`.

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPI_REDUCE handles these functions as a special case. (*End of advice to implementors.*)

MPI_OP_FREE(op)

INOUT	op	operation (handle)

```
int MPI_op_free( MPI_Op *op)
```

```
MPI_OP_FREE( OP, IERROR)
    INTEGER OP, IERROR
```

```
void MPI::Op::Free()
```

Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.

Example of User-defined Reduce

It is time for an example of user-defined reduction. The example in this section uses an intracommunicator.

Example 5.20 Compute the product of an array of complex numbers, in C.

```
typedef struct {
    double real,imag;
} Complex;

/* the user-defined function
```

170 CHAPTER 5. COLLECTIVE COMMUNICATION

```
     */
    if (rank == root) {
        MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
    }
    if (rank == groupsize-1) {
        MPI_Send(sendbuf, count, datatype, root, ...);
    }
    if (rank == root) {
        MPI_Wait(&req, &status);
    }
```

The reduction computation proceeds, sequentially, from process `0` to process `groupsize-1`. This order is chosen so as to respect the order of a possibly non-commutative operator defined by the function `User_reduce()`. A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size `len` <`count`.

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPI_REDUCE handles these functions as a special case. (*End of advice to implementors.*)

MPI_OP_FREE(op)

INOUT op operation (handle)

```
int MPI_op_free( MPI_Op *op)
```

```
MPI_OP_FREE( OP, IERROR)
    INTEGER OP, IERROR
```

```
void MPI::Op::Free()
```

Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.

Example of User-defined Reduce

It is time for an example of user-defined reduction. The example in this section uses an intracommunicator.

Example 5.20 Compute the product of an array of complex numbers, in C.

```
typedef struct {
    double real,imag;
} Complex;

/* the user-defined function
```

```
 */
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
                   inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                   inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* explain to MPI how type Complex is defined
     */
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
    MPI_Type_commit( &ctype );
    /* create the complex-product user-op
     */
    MPI_Op_create( myProd, True, &myOp );

    MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

    /* At this point, the answer, which consists of 100 Complexes,
     * resides on process root
     */
```

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

5.9. GLOBAL REDUCTION OPERATIONS 171

```
 */
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
                   inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                   inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* explain to MPI how type Complex is defined
     */
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
    MPI_Type_commit( &ctype );
    /* create the complex-product user-op
     */
    MPI_Op_create( myProd, True, &myOp );

    MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

    /* At this point, the answer, which consists of 100 Complexes,
     * resides on process root
     */
```

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const = 0
```

If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as MPI_REDUCE except that the result appears in the receive buffer of all the group members.

Advice to implementors. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 5.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
```

172 CHAPTER 5. COLLECTIVE COMMUNICATION

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const = 0
```

If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as MPI_REDUCE except that the result appears in the receive buffer of all the group members.

Advice to implementors. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 5.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
```

```
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN
```

5.10 Reduce-Scatter

MPI includes a variant of the reduce operations where the result is scattered to all processes in a group on return.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
              IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

```
void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
              int recvcounts[], const MPI::Datatype& datatype,
              const MPI::Op& op) const = 0
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first does an element-wise reduction on vector of count $= \sum_i$ recvcounts[i] elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector of results is split into `n` disjoint segments, where `n` is the number of members in the group. Segment `i` contains recvcounts[i] elements. The `i`-th segment is sent to process `i` and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of

5.10. REDUCE-SCATTER 173

    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN

5.10 Reduce-Scatter

MPI includes a variant of the reduce operations where the result is scattered to all processes in a group on return.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN sendbuf starting address of send buffer (choice)
OUT recvbuf starting address of receive buffer (choice)
IN recvcounts non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN datatype data type of elements of input buffer (handle)
IN op operation (handle)
IN comm communicator (handle)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
              IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
              int recvcounts[], const MPI::Datatype& datatype,
              const MPI::Op& op) const = 0

If comm is an intracommunicator, MPI_REDUCE_SCATTER first does an element-wise reduction on vector of count $= \sum_i$ recvcounts[i] elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector of results is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcounts[i] elements. The i-th segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of

recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is scattered among processes in group B, and vice versa. Within each group, all processes provide the same recvcounts argument, and the sum of the recvcounts entries should be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

5.11 Scan

5.11.1 Inclusive Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

```
void MPI::Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const
```

If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i` (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.

174 CHAPTER 5. COLLECTIVE COMMUNICATION

recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the top of the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is scattered among processes in group B, and vice versa. Within each group, all processes provide the same recvcounts argument, and the sum of the recvcounts entries should be the same for the two groups.

Rationale. The last restriction is needed so that the length of the send buffer can be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

5.11 Scan

5.11.1 Inclusive Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN sendbuf starting address of send buffer (choice)
OUT recvbuf starting address of receive buffer (choice)
IN count number of elements in input buffer (non-negative integer)
IN datatype data type of elements of input buffer (handle)
IN op operation (handle)
IN comm communicator (handle)

int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

void MPI::Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const

If comm is an intracommunicator, MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,...,i (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.

This operation is invalid for intercommunicators.

5.11.2 Exclusive Scan

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	intracommunicator (handle)

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

```
void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const
```

If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not signficant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank $i > 1$, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

No “in place” option is supported.

This operation is invalid for intercommunicators.

Advice to users. As for MPI_SCAN, MPI does not specify which processes may call the operation, only that the result be correctly computed. In particular, note that the process with rank 1 need not call the MPI_Op, since all it needs to do is to receive the value from the process with rank 0. However, all processes, even the processes with ranks zero and one, must provide the same op. (*End of advice to users.*)

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan.

No in-place version is specified for MPI_EXSCAN because it is not clear what this means for the process with rank zero. (*End of rationale.*)

5.11. SCAN 175

This operation is invalid for intercommunicators.

5.11.2 Exclusive Scan

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN sendbuf starting address of send buffer (choice)
OUT recvbuf starting address of receive buffer (choice)
IN count number of elements in input buffer (non-negative integer)
IN datatype data type of elements of input buffer (handle)
IN op operation (handle)
IN comm intracommunicator (handle)

int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
const MPI::Datatype& datatype, const MPI::Op& op) const

If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not signficant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank $i > 1$, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

No “in place” option is supported.

This operation is invalid for intercommunicators.

Advice to users. As for MPI_SCAN, MPI does not specify which processes may call the operation, only that the result be correctly computed. In particular, note that the process with rank 1 need not call the MPI_Op, since all it needs to do is to receive the value from the process with rank 0. However, all processes, even the processes with ranks zero and one, must provide the same op. (End of advice to users.)

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan.

No in-place version is specified for MPI_EXSCAN because it is not clear what this means for the process with rank zero. (End of rationale.)

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.22 This example uses a user-defined operation to produce a *segmented scan*. A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

$$\begin{array}{lcccccccc} \textit{values} & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\ \textit{logicals} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ \textit{result} & v_1 & v_1+v_2 & v_3 & v_3+v_4 & v_3+v_4+v_5 & v_6 & v_6+v_7 & v_8 \end{array}$$

The operator that produces this effect is,

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ j \end{pmatrix},$$

where,

$$w = \begin{cases} u+v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases} .$$

Note that this is a non-commutative operator. C code that implements it is given below.

```
typedef struct {
    double val;
    int log;
} SegScanPair;

/* the user-defined function
 */
void segScan( SegScanPair *in, SegScanPair *inout, int *len,
                                                MPI_Datatype *dptr )
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if ( in->log == inout->log )
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

Note that the inout argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

176 CHAPTER 5. COLLECTIVE COMMUNICATION

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.22 This example uses a user-defined operation to produce a segmented scan. A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

values v1 v2 v3 v4 v5 v6 v7 v8
logicals 0 0 1 1 1 0 0 1
result v1 v1+v2 v3 v3+v4 v3+v4+v5 v6 v6+v7 v8

The operator that produces this effect is,

(u i) ∘ (v j) = (w j),

where,

w = u+v if i = j; v if i ≠ j.

Note that this is a non-commutative operator. C code that implements it is given below.

typedef struct {
    double val;
    int log;
} SegScanPair;

/* the user-defined function
 */
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
                                                MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if (in->log == inout->log)
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}

Note that the inout argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
    int i,base;
    SeqScanPair  a, answer;
    MPI_Op       myOp;
    MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
    MPI_Aint     disp[2];
    int          blocklen[2] = { 1, 1};
    MPI_Datatype sspair;

    /* explain to MPI how type SegScanPair is defined
     */
    MPI_Address( a, disp);
    MPI_Address( a.log, disp+1);
    base = disp[0];
    for (i=0; i<2; ++i) disp[i] -= base;
    MPI_Type_struct( 2, blocklen, disp, type, &sspair );
    MPI_Type_commit( &sspair );
    /* create the segmented-scan user-op
     */
    MPI_Op_create( segScan, 0, &myOp );
    ...
    MPI_Scan( a, answer, 1, sspair, myOp, comm );
```

5.12 Correctness

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators.

Example 5.23 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

We assume that the group of comm is {0,1}. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.24 The following is erroneous.

5.12. CORRECTNESS 177

```
    int i,base;
    SeqScanPair  a, answer;
    MPI_Op       myOp;
    MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
    MPI_Aint     disp[2];
    int          blocklen[2] = { 1, 1};
    MPI_Datatype sspair;

    /* explain to MPI how type SegScanPair is defined
     */
    MPI_Address( a, disp);
    MPI_Address( a.log, disp+1);
    base = disp[0];
    for (i=0; i<2; ++i) disp[i] -= base;
    MPI_Type_struct( 2, blocklen, disp, type, &sspair );
    MPI_Type_commit( &sspair );
    /* create the segmented-scan user-op
     */
    MPI_Op_create( segScan, 0, &myOp );
    ...
    MPI_Scan( a, answer, 1, sspair, myOp, comm );
```

5.12 Correctness

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators.

Example 5.23 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

We assume that the group of comm is {0,1}. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

Example 5.24 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

Assume that the group of comm0 is {0,1}, of comm1 is {1, 2} and of comm2 is {2,0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 5.25 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 5.26 An unsafe, non-deterministic program.

```
switch(rank) {
    case 0:
```

178 CHAPTER 5. COLLECTIVE COMMUNICATION

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

Assume that the group of comm0 is {0,1}, of comm1 is {1, 2} and of comm2 is {2,0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 5.25 The following is erroneous.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 5.26 An unsafe, non-deterministic program.

```
switch(rank) {
    case 0:
```

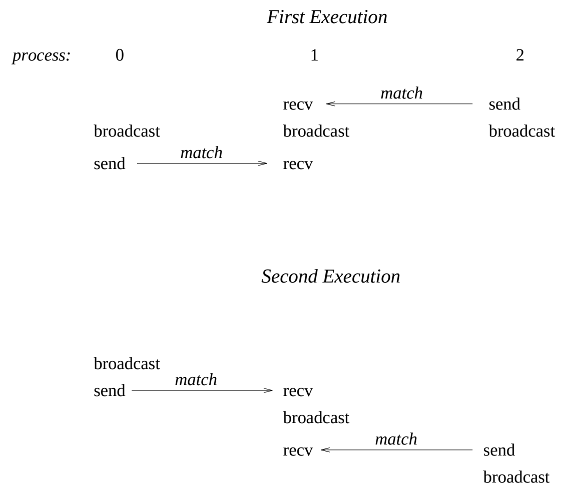


Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

```
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's re-

5.12. CORRECTNESS 179

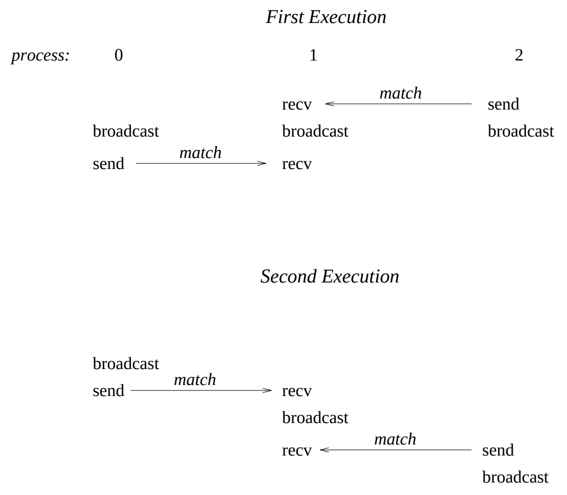


Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

```
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's re-

sponsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then, messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved.

It is the implementor's responsibility to ensure that point-to-point messages are not confused with collective messages. One way to accomplish this is, whenever a communicator is created, to also create a "hidden communicator" for collective communication. One could achieve a similar effect more cheaply, for example, by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication. (*End of advice to implementors.*)

180 CHAPTER 5. COLLECTIVE COMMUNICATION

sponsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then, messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved.

It is the implementor's responsibility to ensure that point-to-point messages are not confused with collective messages. One way to accomplish this is, whenever a communicator is created, to also create a "hidden communicator" for collective communication. One could achieve a similar effect more cheaply, for example, by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication. (*End of advice to implementors.*)

Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a “higher level” of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [42] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to “adorn” a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a “higher level” of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [42] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to “adorn” a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

181

6.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- **Groups** of processes,
- **Virtual topologies**,
- **Attribute caching**,
- **Communicators**.

Communicators (see [19, 40, 45]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

Caching. Communicators (see below) provide a "caching" mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. This can be used by advanced users to adorn communicators further, and by MPI to implement some communicator functions. For example, the virtual-topology functions described in Chapter 7 are likely to be supported this way.

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

Intra-communicators. The most commonly used means for message passing in MPI is via intra-communicators. Intra-communicators contain an instance of a group, contexts of communication for both point-to-point and collective communication, and the ability to include virtual topology and other attributes. These features work as follows:

- **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.
- **Groups** define the participants in the communication (see above) of a communicator.

182 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

6.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- **Groups** of processes,
- **Virtual topologies**,
- **Attribute caching**,
- **Communicators**.

Communicators (see [19, 40, 45]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

Caching. Communicators (see below) provide a "caching" mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. This can be used by advanced users to adorn communicators further, and by MPI to implement some communicator functions. For example, the virtual-topology functions described in Chapter 7 are likely to be supported this way.

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

Intra-communicators. The most commonly used means for message passing in MPI is via intra-communicators. Intra-communicators contain an instance of a group, contexts of communication for both point-to-point and collective communication, and the ability to include virtual topology and other attributes. These features work as follows:

- **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.
- **Groups** define the participants in the communication (see above) of a communicator.

- A **virtual topology** defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- **Attributes** define the local information that the user or library has added to a communicator for later reference.

  Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. *Users who are satisfied with this practice can plug in* MPI_COMM_WORLD *wherever a communicator argument is required, and can consequently disregard the rest of this chapter.* (*End of advice to users.*)

Inter-communicators. The discussion has dealt so far with **intra-communication**: communication within a group. MPI also supports **inter-communication**: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- **Contexts** provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely

6.1. INTRODUCTION 183

- A **virtual topology** defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- **Attributes** define the local information that the user or library has added to a communicator for later reference.

  Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. *Users who are satisfied with this practice can plug in* MPI_COMM_WORLD *wherever a communicator argument is required, and can consequently disregard the rest of this chapter.* (*End of advice to users.*)

Inter-communicators. The discussion has dealt so far with **intra-communication**: communication within a group. MPI also supports **inter-communication**: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- **Contexts** provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely

ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

6.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

6.2.1 Groups

A **group** is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque **group objects**, and hence cannot be directly transferred from one process to another. A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants (thus giving them unique names within that "universe" of communication).

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no members. The predefined constant MPI_GROUP_NULL is the value used for invalid group handles.

> *Advice to users.* MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

> *Advice to implementors.* A group may be represented by a virtual-to-real process-address-translation table. Each communicator object (see below) would have a pointer to such a table.
>
> Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (*End of advice to implementors.*)

6.2.2 Contexts

A **context** is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

> *Advice to implementors.* Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

184 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

6.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

6.2.1 Groups

A **group** is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque **group objects**, and hence cannot be directly transferred from one process to another. A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants (thus giving them unique names within that "universe" of communication).

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no members. The predefined constant MPI_GROUP_NULL is the value used for invalid group handles.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (*End of advice to users.*)

Advice to implementors. A group may be represented by a virtual-to-real process-address-translation table. Each communicator object (see below) would have a pointer to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (*End of advice to implementors.*)

6.2.2 Contexts

A **context** is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicator-generating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also "cache" additional information (see Section 6.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups in different processes.

All MPI implementations are required to provide the MPI_COMM_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to

6.2. BASIC CONCEPTS 185

A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicator-generating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also "cache" additional information (see Section 6.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups in different processes.

All MPI implementations are required to provide the MPI_COMM_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to

this communicator does not appear as a pre-defined constant, but it may be accessed using MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither does MPI specify the function of the host process, if any. Other implementation-dependent, predefined communicators may also be provided.

6.3 Group Management

This section describes the manipulation of process groups in MPI. These operations are local and their execution does not require interprocess communication.

6.3.1 Group Accessors

MPI_GROUP_SIZE(group, size)

IN	group	group (handle)
OUT	size	number of processes in the group (integer)

```
int MPI_Group_size(MPI_Group group, int *size)
```

```
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
```

```
int MPI::Group::Get_size() const
```

MPI_GROUP_RANK(group, rank)

IN	group	group (handle)
OUT	rank	rank of the calling process in group, or MPI_UNDEFINED if the process is not a member (integer)

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
```

```
int MPI::Group::Get_rank() const
```

186 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

this communicator does not appear as a pre-defined constant, but it may be accessed using MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither does MPI specify the function of the host process, if any. Other implementation-dependent, predefined communicators may also be provided.

6.3 Group Management

This section describes the manipulation of process groups in MPI. These operations are local and their execution does not require interprocess communication.

6.3.1 Group Accessors

MPI_GROUP_SIZE(group, size)
IN group group (handle)
OUT size number of processes in the group (integer)

int MPI_Group_size(MPI_Group group, int *size)
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
int MPI::Group::Get_size() const

MPI_GROUP_RANK(group, rank)
IN group group (handle)
OUT rank rank of the calling process in group, or MPI_UNDEFINED if the process is not a member (integer)

int MPI_Group_rank(MPI_Group group, int *rank)
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
int MPI::Group::Get_rank() const

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)
```

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

```
static void MPI::Group::Translate_ranks (const MPI::Group& group1, int n,
              const int ranks1[], const MPI::Group& group2, int ranks2[])
```

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.

MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.

MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
```

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

```
static int MPI::Group::Compare(const MPI::Group& group1,
              const MPI::Group& group2)
```

MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

6.3.2 Group Constructors

Group constructors are used to subset and superset existing groups. These constructors construct new groups from existing groups. These are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator-building functions. MPI does not provide a mechanism to build a group

6.3. GROUP MANAGEMENT 187

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
              MPI_Group group2, int *ranks2)
```

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

```
static void MPI::Group::Translate_ranks (const MPI::Group& group1, int n,
              const int ranks1[], const MPI::Group& group2, int ranks2[])
```

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.

MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.

MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
```

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

```
static int MPI::Group::Compare(const MPI::Group& group1,
              const MPI::Group& group2)
```

MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

6.3.2 Group Constructors

Group constructors are used to subset and superset existing groups. These constructors construct new groups from existing groups. These are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator-building functions. MPI does not provide a mechanism to build a group

from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).

> *Rationale.* In what follows, there is no group duplication function analogous to MPI_COMM_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

> *Advice to implementors.* Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

MPI_COMM_GROUP(comm, group)

IN	comm	communicator (handle)
OUT	group	group corresponding to comm (handle)

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

```
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

```
MPI::Group MPI::Comm::Get_group() const
```

MPI_COMM_GROUP returns in group a handle to the group of comm.

MPI_GROUP_UNION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	union group (handle)

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)
```

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
static MPI::Group MPI::Group::Union(const MPI::Group& group1,
              const MPI::Group& group2)
```

188 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).

Rationale. In what follows, there is no group duplication function analogous to MPI_COMM_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

MPI_COMM_GROUP(comm, group)

IN comm communicator (handle)
OUT group group corresponding to comm (handle)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI::Group MPI::Comm::Get_group() const

MPI_COMM_GROUP returns in group a handle to the group of comm.

MPI_GROUP_UNION(group1, group2, newgroup)

IN group1 first group (handle)
IN group2 second group (handle)
OUT newgroup union group (handle)

int MPI_Group_union(MPI_Group group1, MPI_Group group2,
              MPI_Group *newgroup)

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Union(const MPI::Group& group1,
              const MPI::Group& group2)

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	intersection group (handle)

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
static MPI::Group MPI::Group::Intersect(const MPI::Group& group1,
             const MPI::Group& group2)
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	difference group (handle)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
             const MPI::Group& group2)
```

The set-like operations are defined as follows:

union All elements of the first group (group1), followed by all elements of second group (group2) not in first.

intersect all elements of the first group that are also in the second group, ordered as in first group.

difference all elements of the first group that are not in the second group, ordered as in the first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to MPI_GROUP_EMPTY.

6.3. GROUP MANAGEMENT 189

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	intersection group (handle)

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
static MPI::Group MPI::Group::Intersect(const MPI::Group& group1,
             const MPI::Group& group2)
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	difference group (handle)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
             const MPI::Group& group2)
```

The set-like operations are defined as follows:

union All elements of the first group (group1), followed by all elements of second group (group2) not in first.

intersect all elements of the first group that are also in the second group, ordered as in first group.

difference all elements of the first group that are not in the second group, ordered as in the first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to MPI_GROUP_EMPTY.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (and size of newgroup) (integer)
IN	ranks	ranks of processes in group to appear in newgroup (array of integers)
OUT	newgroup	new group derived from above, in the order defined by ranks (handle)

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

```
MPI::Group MPI::Group::Incl(int n, const int ranks[]) const
```

The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks rank[0],..., rank[n-1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (integer)
IN	ranks	array of integer ranks in group not to appear in newgroup
OUT	newgroup	new group derived from above, preserving the order defined by group (handle)

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

```
MPI::Group MPI::Group::Excl(int n, const int ranks[]) const
```

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0] ,... ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If n = 0, then newgroup is identical to group.

190 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN group — group (handle)
IN n — number of elements in array ranks (and size of newgroup) (integer)
IN ranks — ranks of processes in group to appear in newgroup (array of integers)
OUT newgroup — new group derived from above, in the order defined by ranks (handle)

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI::Group MPI::Group::Incl(int n, const int ranks[]) const

The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks rank[0],..., rank[n-1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI_GROUP_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI_GROUP_COMPARE.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN group — group (handle)
IN n — number of elements in array ranks (integer)
IN ranks — array of integer ranks in group not to appear in newgroup
OUT newgroup — new group derived from above, preserving the order defined by group (handle)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI::Group MPI::Group::Excl(int n, const int ranks[]) const

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0] ,... ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If n = 0, then newgroup is identical to group.

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN	group	group (handle)
IN	n	number of triplets in array ranges (integer)
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup
OUT	newgroup	new group derived from above, in the order defined by ranges (handle)

```
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

```
MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3]) const
```

If ranges consist of the triplets

$$(first_1, last_1, stride_1), ..., (first_n, last_n, stride_n)$$

then newgroup consists of the sequence of processes in group with ranks

$$first_1, first_1 + stride_1, ..., first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, ...$$

$$first_n, first_n + stride_n, ..., first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have $first_i > last_i$, and $stride_i$ may be negative, but cannot be zero.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranges (integer)
IN	ranges	a one-dimensional array of integer triplets of the form (first rank, last rank, stride), indicating the ranks in group of processes to be excluded from the output group newgroup.
OUT	newgroup	new group derived from above, preserving the order in group (handle)

6.3. GROUP MANAGEMENT 191

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN group group (handle)
IN n number of triplets in array ranges (integer)
IN ranges a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup
OUT newgroup new group derived from above, in the order defined by ranges (handle)

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
              MPI_Group *newgroup)

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3]) const

If ranges consist of the triplets

$(first_1, last_1, stride_1), ..., (first_n, last_n, stride_n)$

then newgroup consists of the sequence of processes in group with ranks

$first_1, first_1 + stride_1, ..., first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, ...$

$first_n, first_n + stride_n, ..., first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have $first_i > last_i$, and $stride_i$ may be negative, but cannot be zero.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN group group (handle)
IN n number of elements in array ranges (integer)
IN ranges a one-dimensional array of integer triplets of the form (first rank, last rank, stride), indicating the ranks in group of processes to be excluded from the output group newgroup.
OUT newgroup new group derived from above, preserving the order in group (handle)

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

```
MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const
```

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (*End of advice to implementors.*)

6.3.3 Group Destructors

MPI_GROUP_FREE(group)

INOUT	group	group (handle)

```
int MPI_Group_free(MPI_Group *group)
```

```
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

```
void MPI::Group::Free()
```

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE and MPI_COMM_DUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

192 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (*End of advice to implementors.*)

6.3.3 Group Destructors

MPI_GROUP_FREE(group)

INOUT group group (handle)

int MPI_Group_free(MPI_Group *group)

MPI_GROUP_FREE(GROUP, IERROR)
INTEGER GROUP, IERROR

void MPI::Group::Free()

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE and MPI_COMM_DUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

6.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

6.4.1 Communicator Accessors

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

```
int MPI::Comm::Get_size() const
```

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_COMM_RANK indicates the rank of the process that calls it in the range from $0 \ldots \mathsf{size}-1$, where size is the return value of MPI_COMM_SIZE.(*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of the calling process in group of comm (integer)

6.4. COMMUNICATOR MANAGEMENT 193

6.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

6.4.1 Communicator Accessors

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN comm communicator (handle)
OUT size number of processes in the group of comm (integer)

int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

int MPI::Comm::Get_size() const

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the size using MPI_GROUP_SIZE, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_COMM_RANK indicates the rank of the process that calls it in the range from $0 \ldots \mathsf{size}-1$, where size is the return value of MPI_COMM_SIZE.(*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN comm communicator (handle)
OUT rank rank of the calling process in group of comm (integer)

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

int MPI::Comm::Get_rank() const
```

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function gives the rank of the process in the particular communicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.

Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)
IN	comm2	second communicator (handle)
OUT	result	result (integer)

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

static int MPI::Comm::Compare(const MPI::Comm& comm1,
             const MPI::Comm& comm2)
```

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm.

Rationale. Note that there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. The base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. This model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

194 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

int MPI::Comm::Get_rank() const

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function gives the rank of the process in the particular communicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.

Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)

IN comm1 first communicator (handle)
IN comm2 second communicator (handle)
OUT result result (integer)

int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

static int MPI::Comm::Compare(const MPI::Comm& comm1,
             const MPI::Comm& comm2)

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm.

Rationale. Note that there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. The base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. This model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

The MPI interface provides four communicator construction routines that apply to both intracommunicators and intercommunicators. The construction routine MPI_INTERCOMM_CREATE (discussed later) applies only to intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local* group; the other group (relative to that process) is the *remote* group. The left and right group labels give us a way to describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are).

MPI_COMM_DUP(comm, newcomm)

IN	comm	communicator (handle)
OUT	newcomm	copy of comm (handle)

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

```
MPI::Intracomm MPI::Intracomm::Dup() const

MPI::Intercomm MPI::Intercomm::Dup() const

MPI::Cartcomm MPI::Cartcomm::Dup() const

MPI::Graphcomm MPI::Graphcomm::Dup() const

MPI::Comm& MPI::Comm::Clone() const = 0

MPI::Intracomm& MPI::Intracomm::Clone() const

MPI::Intercomm& MPI::Intercomm::Clone() const

MPI::Cartcomm& MPI::Cartcomm::Clone() const

MPI::Graphcomm& MPI::Graphcomm::Clone() const
```

MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 455 for further discussion about the C++ bindings for Dup() and Clone().

Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of

6.4. COMMUNICATOR MANAGEMENT 195

The MPI interface provides four communicator construction routines that apply to both intracommunicators and intercommunicators. The construction routine MPI_INTERCOMM_CREATE (discussed later) applies only to intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intercommunicator are called the *left* and *right* groups. A process in an intercommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the *local* group; the other group (relative to that process) is the *remote* group. The left and right group labels give us a way to describe the two groups in an intercommunicator that is not relative to any particular process (as the local and remote groups are).

MPI_COMM_DUP(comm, newcomm)

IN comm communicator (handle)
OUT newcomm copy of comm (handle)

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

MPI::Intracomm MPI::Intracomm::Dup() const

MPI::Intercomm MPI::Intercomm::Dup() const

MPI::Cartcomm MPI::Cartcomm::Dup() const

MPI::Graphcomm MPI::Graphcomm::Dup() const

MPI::Comm& MPI::Comm::Clone() const = 0

MPI::Intracomm& MPI::Intracomm::Clone() const

MPI::Intercomm& MPI::Intercomm::Clone() const

MPI::Cartcomm& MPI::Cartcomm::Clone() const

MPI::Graphcomm& MPI::Graphcomm::Clone() const

MPI_COMM_DUP Duplicates the existing communicator comm with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new communicator. Returns in newcomm a new communicator with the same group or groups, any copied cached information, but a new context (see Section 6.7.1). Please see Section 16.1.7 on page 455 for further discussion about the C++ bindings for Dup() and Clone().

Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of

the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicator management are also possible.

This call applies to both intra- and inter-communicators. (*End of advice to users.*)

Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(*End of advice to implementors.*)

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	communicator (handle)
IN	group	Group, which is a subset of the group of comm (handle)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

```
MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const
```

```
MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const
```

If comm is an intra-communicator, this function creates a new communicator newcomm with communication group defined by group and a new context. No cached information propagates from comm to newcomm. The function returns MPI_COMM_NULL to processes that are not in group. The call is erroneous if not all group arguments have the same value, or if group is not a subset of the group associated with comm. Note that the call is to be executed by all processes in comm, even if they do not belong to the new group.

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations sometimes to avoid communication related to context creation.

(*End of rationale.*)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

196 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end of the call. Other models of communicator management are also possible.

This call applies to both intra- and inter-communicators. (*End of advice to users.*)

Advice to implementors. One need not actually copy the group information, but only add a new reference and increment the reference count. Copy on write can be used for the cached information.(*End of advice to implementors.*)

MPI_COMM_CREATE(comm, group, newcomm)

IN comm communicator (handle)
IN group Group, which is a subset of the group of comm (handle)
OUT newcomm new communicator (handle)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR

MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const

MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const

If comm is an intra-communicator, this function creates a new communicator newcomm with communication group defined by group and a new context. No cached information propagates from comm to newcomm. The function returns MPI_COMM_NULL to processes that are not in group. The call is erroneous if not all group arguments have the same value, or if group is not a subset of the group associated with comm. Note that the call is to be executed by all processes in comm, even if they do not belong to the new group.

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations sometimes to avoid communication related to context creation.

(*End of rationale.*)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) further to subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

Advice to implementors. Since all processes calling MPI_COMM_DUP or MPI_COMM_CREATE provide the same group argument, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

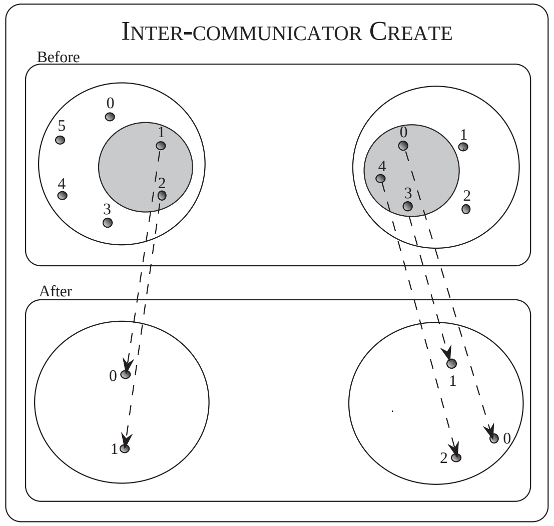


Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

6.4. COMMUNICATOR MANAGEMENT 197

Advice to implementors. Since all processes calling MPI_COMM_DUP or MPI_COMM_CREATE provide the same group argument, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (*End of rationale.*)

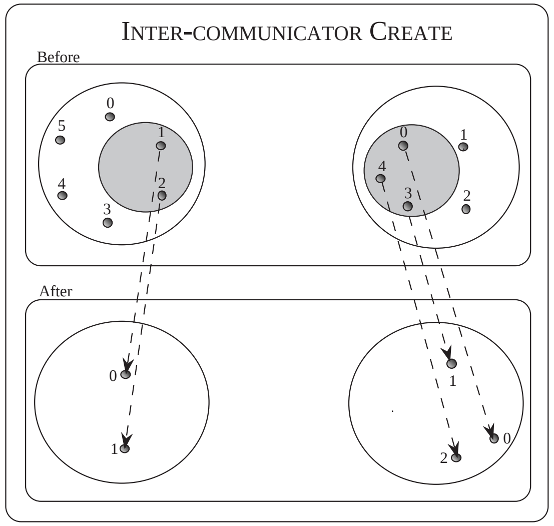


Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

Example 6.1 The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

```
        MPI_Comm  inter_comm, new_inter_comm;
        MPI_Group local_group, group;
        int       rank = 0; /* rank on left side to include in
                               new inter-comm */

        /* Construct the original intercommunicator: "inter_comm" */
        ...

        /* Construct the group of processes to be in new
           intercommunicator */
        if (/* I'm on the left side of the intercommunicator */) {
          MPI_Comm_group ( inter_comm, &local_group );
          MPI_Group_incl ( local_group, 1, &rank, &group );
          MPI_Group_free ( &local_group );
        }
        else
          MPI_Comm_group ( inter_comm, &group );

        MPI_Comm_create ( inter_comm, group, &new_inter_comm );
        MPI_Group_free( &group );
```

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assigment (integer)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

```
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const
```

```
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

198 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

Example 6.1 The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

```
        MPI_Comm  inter_comm, new_inter_comm;
        MPI_Group local_group, group;
        int       rank = 0; /* rank on left side to include in
                               new inter-comm */

        /* Construct the original intercommunicator: "inter_comm" */
        ...

        /* Construct the group of processes to be in new
           intercommunicator */
        if (/* I'm on the left side of the intercommunicator */) {
          MPI_Comm_group ( inter_comm, &local_group );
          MPI_Group_incl ( local_group, 1, &rank, &group );
          MPI_Group_free ( &local_group );
        }
        else
          MPI_Comm_group ( inter_comm, &group );

        MPI_Comm_create ( inter_comm, group, &new_inter_comm );
        MPI_Group_free( &group );
```

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assigment (integer)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

```
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const
```

```
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

A call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where all members of group provide color = 0 and key = rank in group, and all processes that are not members of group provide color = MPI_UNDEFINED. The function MPI_COMM_SPLIT allows more general partitioning of a group into one or more subgroups with optional reordering.

The value of color must be nonnegative.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the color and key in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.

Essentially, making the key value zero for all processes of a given color means that one doesn't really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

Rationale. color is restricted to be nonnegative, so as not to confict with the value assigned to MPI_UNDEFINED. (*End of rationale.*)

The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

Example 6.2 (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create intercommunicator with clients and servers:
   multiple_server_comm */
...
```

6.4. COMMUNICATOR MANAGEMENT 199

A call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where all members of group provide color = 0 and key = rank in group, and all processes that are not members of group provide color = MPI_UNDEFINED. The function MPI_COMM_SPLIT allows more general partitioning of a group into one or more subgroups with optional reordering.

The value of color must be nonnegative.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the color and key in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.

Essentially, making the key value zero for all processes of a given color means that one doesn't really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

Rationale. color is restricted to be nonnegative, so as not to confict with the value assigned to MPI_UNDEFINED. (*End of rationale.*)

The result of MPI_COMM_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, MPI_COMM_NULL is returned. MPI_COMM_NULL is also returned to those processes that specify MPI_UNDEFINED as the color.

Example 6.2 (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create intercommunicator with clients and servers:
   multiple_server_comm */
...
```

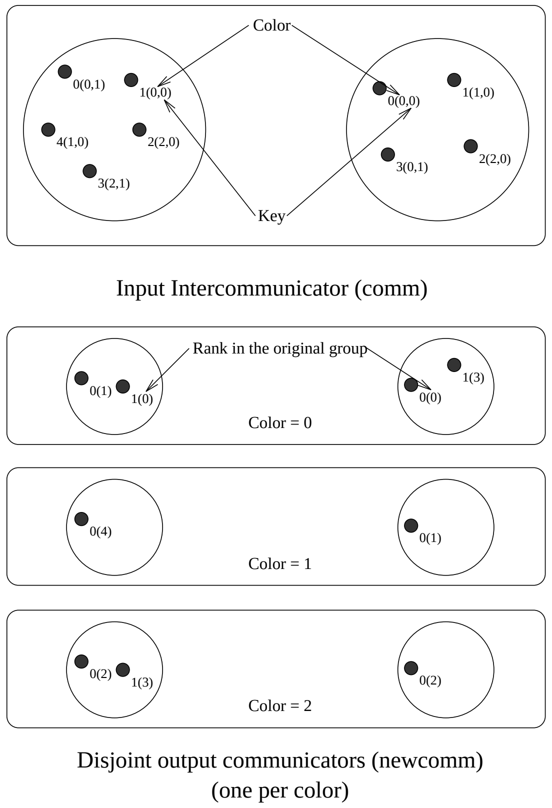


Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.

200 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.

```
        /* Find out the number of servers available */
        MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

        /* Determine my color */
        MPI_Comm_rank ( multiple_server_comm, &rank );
        color = rank % num_servers;

        /* Split the intercommunicator */
        MPI_Comm_split ( multiple_server_comm, color, rank,
                         &single_server_comm );
```

The following is the corresponding server code:

```
        /* Server code */
        MPI_Comm  multiple_client_comm;
        MPI_Comm  single_server_comm;
        int       rank;

        /* Create intercommunicator with clients and servers:
           multiple_client_comm */
        ...

        /* Split the intercommunicator for a single server per group
           of clients */
        MPI_Comm_rank ( multiple_client_comm, &rank );
        MPI_Comm_split ( multiple_client_comm, rank, 0,
                         &single_server_comm );
```

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT	comm	communicator to be destroyed (handle)

```
int MPI_Comm_free(MPI_Comm *comm)
```

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

```
void MPI::Comm::Free()
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

6.4. COMMUNICATOR MANAGEMENT 201

```
        /* Find out the number of servers available */
        MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

        /* Determine my color */
        MPI_Comm_rank ( multiple_server_comm, &rank );
        color = rank % num_servers;

        /* Split the intercommunicator */
        MPI_Comm_split ( multiple_server_comm, color, rank,
                         &single_server_comm );
```

The following is the corresponding server code:

```
        /* Server code */
        MPI_Comm  multiple_client_comm;
        MPI_Comm  single_server_comm;
        int       rank;

        /* Create intercommunicator with clients and servers:
           multiple_client_comm */
        ...

        /* Split the intercommunicator for a single server per group
           of clients */
        MPI_Comm_rank ( multiple_client_comm, &rank );
        MPI_Comm_split ( multiple_client_comm, rank, 0,
                         &single_server_comm );
```

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT comm communicator to be destroyed (handle)

```
int MPI_Comm_free(MPI_Comm *comm)
```

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

```
void MPI::Comm::Free()
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to MPI_COMM_DUP, and decremented by each call to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

6.5 Motivating Examples

6.5.1 Current Practice #1

Example #1a:

```
main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the "all" communicator, and prints a message. It terminates itself legally too. This example does not imply that MPI supports `printf`-like communication itself.
Example #1b (supposing that `size` is even):

```
main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
      /* send unless highest-numbered process */
      if((me + 1) < size)
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
   }
```

202 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to MPI_COMM_DUP, and decremented by each call to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

6.5 Motivating Examples

6.5.1 Current Practice #1

Example #1a:

```
main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the "all" communicator, and prints a message. It terminates itself legally too. This example does not imply that MPI supports `printf`-like communication itself.
Example #1b (supposing that `size` is even):

```
main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
      /* send unless highest-numbered process */
      if((me + 1) < size)
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
   }
```

```
      else
         MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);

      ...
      MPI_Finalize();
   }
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

6.5.2 Current Practice #2

```
main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ``data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

6.5.3 (Approximate) Current Practice #3

```
main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */
```

6.5. MOTIVATING EXAMPLES 203

```
      else
         MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);

      ...
      MPI_Finalize();
   }
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

6.5.2 Current Practice #2

```
main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ``data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

6.5.3 (Approximate) Current Practice #3

```
main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */
```

```
    MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* local */
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

    if(me != 0)
    {
      /* compute on slave */
      ...
      MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
      ...
      MPI_Comm_free(&commslave);
    }
    /* zero falls through immediately to this reduce, others do later... */
    MPI_Reduce(send_buf2, recv_buff2, count2,
               MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&grprem);
    MPI_Finalize();
  }
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

6.5.4 Example #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT       50

main(int argc, char **argv)
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
```

204 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
    MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* local */
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

    if(me != 0)
    {
      /* compute on slave */
      ...
      MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
      ...
      MPI_Comm_free(&commslave);
    }
    /* zero falls through immediately to this reduce, others do later... */
    MPI_Reduce(send_buf2, recv_buff2, count2,
               MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&grprem);
    MPI_Finalize();
  }
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

6.5.4 Example #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT       50

main(int argc, char **argv)
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
  MPI_Group_rank(subgroup, &me);     /* local */

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED)
  {
      MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                        the_comm, request);
      MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                        the_comm, request+1);
      for(i = 0; i < SOME_COUNT, i++)
        MPI_Reduce(..., the_comm);
      MPI_Waitall(2, request, status);

      MPI_Comm_free(&the_comm);
  }

  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&subgroup);
  MPI_Finalize();
}
```

6.5.5 Library Example #1

The main program:

```
main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
  ...
  user_start_op(libh_a, dataset1);
  user_start_op(libh_b, dataset2);
  ...
  while(!done)
  {
     /* work */
     ...
     MPI_Reduce(..., MPI_COMM_WORLD);
```

6.5. MOTIVATING EXAMPLES 205

```
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
  MPI_Group_rank(subgroup, &me);     /* local */

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED)
  {
      MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                        the_comm, request);
      MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                        the_comm, request+1);
      for(i = 0; i < SOME_COUNT, i++)
        MPI_Reduce(..., the_comm);
      MPI_Waitall(2, request, status);

      MPI_Comm_free(&the_comm);
  }

  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&subgroup);
  MPI_Finalize();
}
```

6.5.5 Library Example #1

The main program:

```
main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
  ...
  user_start_op(libh_a, dataset1);
  user_start_op(libh_b, dataset2);
  ...
  while(!done)
  {
     /* work */
     ...
     MPI_Reduce(..., MPI_COMM_WORLD);
```

```
        ...
        /* see if done */
        ...
     }
     user_end_op(libh_a);
     user_end_op(libh_b);

     uninit_user_lib(libh_a);
     uninit_user_lib(libh_b);
     MPI_Finalize();
   }
```

The user library initialization code:

```
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
   {
     user_lib_t *save;

     user_lib_initsave(&save); /* local */
     MPI_Comm_dup(comm, &(save -> comm));

     /* other inits */
     ...

     *handle = save;
   }
```

User start-up code:

```
   void user_start_op(user_lib_t *handle, void *data)
   {
     MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
     MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
   }
```

User communication clean-up code:

```
   void user_end_op(user_lib_t *handle)
   {
     MPI_Status status;
     MPI_Wait(handle -> isend_handle, &status);
     MPI_Wait(handle -> irecv_handle, &status);
   }
```

User object clean-up code:

```
   void uninit_user_lib(user_lib_t *handle)
   {
     MPI_Comm_free(&(handle -> comm));
     free(handle);
   }
```

206 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
        ...
        /* see if done */
        ...
     }
     user_end_op(libh_a);
     user_end_op(libh_b);

     uninit_user_lib(libh_a);
     uninit_user_lib(libh_b);
     MPI_Finalize();
   }
```

The user library initialization code:

```
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
   {
     user_lib_t *save;

     user_lib_initsave(&save); /* local */
     MPI_Comm_dup(comm, &(save -> comm));

     /* other inits */
     ...

     *handle = save;
   }
```

User start-up code:

```
   void user_start_op(user_lib_t *handle, void *data)
   {
     MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
     MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
   }
```

User communication clean-up code:

```
   void user_end_op(user_lib_t *handle)
   {
     MPI_Status status;
     MPI_Wait(handle -> isend_handle, &status);
     MPI_Wait(handle -> irecv_handle, &status);
   }
```

User object clean-up code:

```
   void uninit_user_lib(user_lib_t *handle)
   {
     MPI_Comm_free(&(handle -> comm));
     free(handle);
   }
```

6.5.6 Library Example #2

The main program:

```
   main(int argc, char **argv)
   {
     int ma, mb;
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
     MPI_Comm comm_a, comm_b;

     static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
     static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
     static int list_b[] = {0, 2};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);

     ...
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
```

6.5. MOTIVATING EXAMPLES 207

6.5.6 Library Example #2

The main program:

```
   main(int argc, char **argv)
   {
     int ma, mb;
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
     MPI_Comm comm_a, comm_b;

     static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
     static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
     static int list_b[] = {0, 2};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);

     ...
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
```

```
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Finalize();
  }
```

The library:

```
  void lib_call(MPI_Comm comm)
  {
    int me, done = 0;
    MPI_Status status;
    MPI_Comm_rank(comm, &me);
    if(me == 0)
       while(!done)
       {
          MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
          ...
       }
    else
    {
      /* work */
      MPI_Send(..., 0, ARBITRARY_TAG, comm);
      ....
    }
#ifdef EXAMPLE_2C
    /* include (resp, exclude) for safety (resp, no safety): */
    MPI_Barrier(comm);
#endif
  }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [45]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-to-point operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 6.9.

208 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Finalize();
  }
```

The library:

```
  void lib_call(MPI_Comm comm)
  {
    int me, done = 0;
    MPI_Status status;
    MPI_Comm_rank(comm, &me);
    if(me == 0)
       while(!done)
       {
          MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
          ...
       }
    else
    {
      /* work */
      MPI_Send(..., 0, ARBITRARY_TAG, comm);
      ....
    }
#ifdef EXAMPLE_2C
    /* include (resp, exclude) for safety (resp, no safety): */
    MPI_Barrier(comm);
#endif
  }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [45]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-to-point operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 6.9.

6.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called "inter-communication" and the communicator used is called an "inter-communicator," as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint.

> *Advice to users.* The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

6.6. INTER-COMMUNICATION 209

6.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called "inter-communication" and the communicator used is called an "inter-communicator," as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint.

> *Advice to users.* The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_COMM_CREATE).

> *Advice to implementors.* For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:
>
> **group**
> **send_context**
> **receive_context**
> **source**
>
> For inter-communicators, **group** describes the remote group, and **source** is the rank of the process in the local group. For intra-communicators, **group** is the communicator group (remote=local), **source** is the rank of the process in this group, and **send context** and **receive context** are identical. A group can be represented by a rank-to-absolute-address translation table.
>
> The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process **P** in group $\mathcal{P}$, which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$, and a process **Q** in group $\mathcal{Q}$, which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then
>
> - $\mathbf{C}_{\mathcal{P}}$**.group** describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$**.group** describes the group $\mathcal{P}$.
> - $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathrm{C}_{\mathcal{Q}}$**.receive_context** and the context is unique in $\mathcal{Q}$;
> $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** and this context is unique in $\mathcal{P}$.
> - $\mathbf{C}_{\mathcal{P}}$**.source** is rank of **P** in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$**.source** is rank of **Q** in $\mathcal{Q}$.
>
> Assume that **P** sends a message to **Q** using the inter-communicator. Then **P** uses the **group** table to find the absolute address of **Q**; **source** and **send_context** are appended to the message.
>
> Assume that **Q** posts a receive with an explicit source argument using the inter-communicator. Then **Q** matches **receive_context** to the message context and source argument to the message source.
>
> The same algorithm is appropriate for intra-communicators as well.
>
> In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

6.6.1 Inter-communicator Accessors

MPI_COMM_TEST_INTER(comm, flag)

IN	comm	communicator (handle)
OUT	flag	(logical)

210 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_COMM_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group
send_context
receive_context
source

For inter-communicators, **group** describes the remote group, and **source** is the rank of the process in the local group. For intra-communicators, **group** is the communicator group (remote=local), **source** is the rank of the process in this group, and **send context** and **receive context** are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process **P** in group $\mathcal{P}$, which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$, and a process **Q** in group $\mathcal{Q}$, which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $\mathbf{C}_{\mathcal{P}}$**.group** describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$**.group** describes the group $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathrm{C}_{\mathcal{Q}}$**.receive_context** and the context is unique in $\mathcal{Q}$;
  $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** and this context is unique in $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.source** is rank of **P** in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$**.source** is rank of **Q** in $\mathcal{Q}$.

Assume that **P** sends a message to **Q** using the inter-communicator. Then **P** uses the **group** table to find the absolute address of **Q**; **source** and **send_context** are appended to the message.

Assume that **Q** posts a receive with an explicit source argument using the inter-communicator. Then **Q** matches **receive_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

6.6.1 Inter-communicator Accessors

MPI_COMM_TEST_INTER(comm, flag)

IN comm communicator (handle)
OUT flag (logical)

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

```
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
```

```
bool MPI::Comm::Is_inter() const
```

This local routine allows the calling process to determine if a communicator is an inter-communicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

MPI_COMM_SIZE	returns the size of the local group.
MPI_COMM_GROUP	returns the local group.
MPI_COMM_RANK	returns the rank in the local group

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT and MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an inter-communicator:

The following are all local operations.

MPI_COMM_REMOTE_SIZE(comm, size)

IN	comm	inter-communicator (handle)
OUT	size	number of processes in the remote group of comm (integer)

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

```
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

```
int MPI::Intercomm::Get_remote_size() const
```

MPI_COMM_REMOTE_GROUP(comm, group)

IN	comm	inter-communicator (handle)
OUT	group	remote group corresponding to comm (handle)

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

6.6. INTER-COMMUNICATION 211

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
INTEGER COMM, IERROR
LOGICAL FLAG

bool MPI::Comm::Is_inter() const

This local routine allows the calling process to determine if a communicator is an inter-communicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

MPI_COMM_SIZE returns the size of the local group.
MPI_COMM_GROUP returns the local group.
MPI_COMM_RANK returns the rank in the local group

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT and MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an inter-communicator:

The following are all local operations.

MPI_COMM_REMOTE_SIZE(comm, size)
IN comm inter-communicator (handle)
OUT size number of processes in the remote group of comm (integer)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR

int MPI::Intercomm::Get_remote_size() const

MPI_COMM_REMOTE_GROUP(comm, group)
IN comm inter-communicator (handle)
OUT group remote group corresponding to comm (handle)

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

```
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

```
MPI::Group MPI::Intercomm::Get_remote_group() const
```

Rationale. Symmetric access to both the local and remote groups of an inter-communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

6.6.2 Inter-communicator Operations

This section introduces four blocking inter-communicator operations. MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

In standard MPI implementations (with static process allocation at initialization), the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that have used spawn or join, it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 7 do not apply to inter-communicators. Users that require this capability should utilize MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topology-oriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

212 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI::Group MPI::Intercomm::Get_remote_group() const

Rationale. Symmetric access to both the local and remote groups of an inter-communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

6.6.2 Inter-communicator Operations

This section introduces four blocking inter-communicator operations. MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

In standard MPI implementations (with static process allocation at initialization), the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that have used spawn or join, it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 7 do not apply to inter-communicators. Users that require this capability should utilize MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topology-oriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN	local_comm	local intra-communicator (handle)
IN	local_leader	rank of local group leader in local_comm (integer)
IN	peer_comm	"peer" communicator; significant only at the local_leader (handle)
IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)
IN	tag	"safe" tag (integer)
OUT	newintercomm	new inter-communicator (handle)

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
              MPI_Comm peer_comm, int remote_leader, int tag,
              MPI_Comm *newintercomm)
```

```
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
              TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR
```

```
MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const
              MPI::Comm& peer_comm, int remote_leader, int tag) const
```

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.

This call uses point-to-point communication with communicator peer_comm, and with tag tag between the leaders. Thus, care must be taken that there be no pending communication on peer_comm that could interfere with this communication.

Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (*End of advice to users.*)

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN	intercomm	Inter-Communicator (handle)
IN	high	(logical)
OUT	newintracomm	new intra-communicator (handle)

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
              MPI_Comm *newintracomm)
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
```

6.6. INTER-COMMUNICATION 213

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN local_comm local intra-communicator (handle)
IN local_leader rank of local group leader in local_comm (integer)
IN peer_comm "peer" communicator; significant only at the local_leader (handle)
IN remote_leader rank of remote group leader in peer_comm; significant only at the local_leader (integer)
IN tag "safe" tag (integer)
OUT newintercomm new inter-communicator (handle)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
              MPI_Comm peer_comm, int remote_leader, int tag,
              MPI_Comm *newintercomm)

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
              TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR

MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const
              MPI::Comm& peer_comm, int remote_leader, int tag) const

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.

This call uses point-to-point communication with communicator peer_comm, and with tag tag between the leaders. Thus, care must be taken that there be no pending communication on peer_comm that could interfere with this communication.

Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (*End of advice to users.*)

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN intercomm Inter-Communicator (handle)
IN high (logical)
OUT newintracomm new intra-communicator (handle)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
              MPI_Comm *newintracomm)

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH

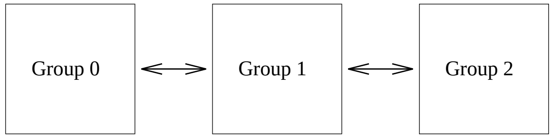


Figure 6.3: Three-group pipeline.

```
MPI::Intracomm MPI::Intercomm::Merge(bool high) const
```

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the “low” group before the “high” group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

> *Advice to implementors.* The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

6.6.3 Inter-Communication Examples

Example 1: Three-Group “Pipeline”

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```
   main(int argc, char **argv)
   {
     MPI_Comm   myComm;       /* intra-communicator of local sub-group */
     MPI_Comm   myFirstComm;  /* inter-communicator */
     MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only) */
     int membershipKey;
     int rank;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     /* User code must generate membershipKey in the range [0, 1, 2] */
     membershipKey = rank % 3;
```

214 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

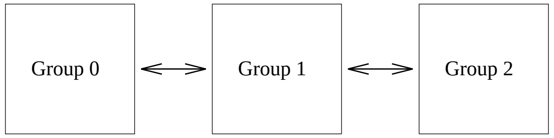


Figure 6.3: Three-group pipeline.

MPI::Intracomm MPI::Intercomm::Merge(bool high) const

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the “low” group before the “high” group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

Advice to implementors. The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

6.6.3 Inter-Communication Examples

Example 1: Three-Group “Pipeline”

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

   main(int argc, char **argv)
   {
     MPI_Comm   myComm;   /* intra-communicator of local sub-group */
     MPI_Comm   myFirstComm; /* inter-communicator */
     MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only) */
     int membershipKey;
     int rank;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     /* User code must generate membershipKey in the range [0, 1, 2] */
     membershipKey = rank % 3;

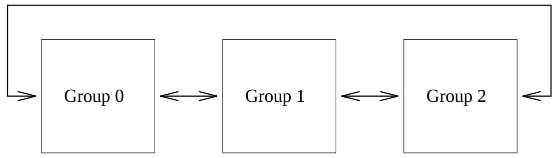


Figure 6.4: Three-group ring.

```
   /* Build intra-communicator for local sub-group */
   MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

   /* Build inter-communicators.  Tags are hard-coded. */
   if (membershipKey == 0)
   {                     /* Group 0 communicates with group 1. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
   }
   else if (membershipKey == 1)
   {              /* Group 1 communicates with groups 0 and 2. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
   }
   else if (membershipKey == 2)
   {                     /* Group 2 communicates with group 1. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm);
   }

   /* Do work ... */

   switch(membershipKey)  /* free communicators appropriately */
   {
   case 1:
      MPI_Comm_free(&mySecondComm);
   case 0:
   case 2:
      MPI_Comm_free(&myFirstComm);
      break;
   }

   MPI_Finalize();
 }
```

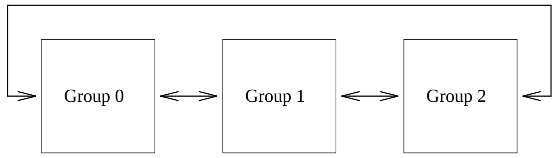


Figure 6.4: Three-group ring.

```
   /* Build intra-communicator for local sub-group */
   MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

   /* Build inter-communicators.  Tags are hard-coded. */
   if (membershipKey == 0)
   {                     /* Group 0 communicates with group 1. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
   }
   else if (membershipKey == 1)
   {              /* Group 1 communicates with groups 0 and 2. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
   }
   else if (membershipKey == 2)
   {                     /* Group 2 communicates with group 1. */
     MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm);
   }

   /* Do work ... */

   switch(membershipKey)  /* free communicators appropriately */
   {
   case 1:
      MPI_Comm_free(&mySecondComm);
   case 0:
   case 2:
      MPI_Comm_free(&myFirstComm);
      break;
   }

   MPI_Finalize();
 }
```

Example 2: Three-Group "Ring"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
main(int argc, char **argv)
{
  MPI_Comm   myComm;      /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm; /* inter-communicators */
  MPI_Comm   mySecondComm;
  MPI_Status status;
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* Build inter-communicators.  Tags are hard-coded. */
  if (membershipKey == 0)
  {             /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          2, &mySecondComm);
  }
  else if (membershipKey == 1)
  {         /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
  }
  else if (membershipKey == 2)
  {        /* Group 2 communicates with groups 0 and 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          2, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &mySecondComm);
  }

  /* Do some work ... */
```

216 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

Example 2: Three-Group "Ring"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
main(int argc, char **argv)
{
  MPI_Comm   myComm;      /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm; /* inter-communicators */
  MPI_Comm   mySecondComm;
  MPI_Status status;
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* Build inter-communicators.  Tags are hard-coded. */
  if (membershipKey == 0)
  {             /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          2, &mySecondComm);
  }
  else if (membershipKey == 1)
  {         /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
  }
  else if (membershipKey == 2)
  {        /* Group 2 communicates with groups 0 and 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          2, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &mySecondComm);
  }

  /* Do some work ... */
```

```
    /* Then free communicators before terminating... */
    MPI_Comm_free(&myFirstComm);
    MPI_Comm_free(&mySecondComm);
    MPI_Comm_free(&myComm);
    MPI_Finalize();
  }
```

Example 3: Building Name Service for Intercommunication

The following procedures exemplify the process by which a user could create name service for building intercommunicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute MPI_INIT, every process calls the example function, Init_server(), defined below. Then, if the new_world returned is NULL, the process getting NULL is required to implement a server function, in a reactive loop, Do_server(). Everyone else just does their prescribed computation, using new_world as the new effective "global" communicator. One designated process calls Undo_Server() to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

```
#define INIT_SERVER_TAG_1   666
#define UNDO_SERVER_TAG_1   777

static int server_key_val;

/* for attribute management for server_comm,  copy callback: */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state,
void *attribute_val_in, void **attribute_val_out, int *flag)
{
   /* copy the handle */
   *attribute_val_out = attribute_val_in;
   *flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world;    /* new effective world, sans server */
{
    MPI_Comm temp_comm, lone_comm;
    MPI_Group peer_group, temp_group;
    int rank_in_peer_comm, size, color, key = 0;
```

6.6. INTER-COMMUNICATION 217

```
    /* Then free communicators before terminating... */
    MPI_Comm_free(&myFirstComm);
    MPI_Comm_free(&mySecondComm);
    MPI_Comm_free(&myComm);
    MPI_Finalize();
  }
```

Example 3: Building Name Service for Intercommunication

The following procedures exemplify the process by which a user could create name service for building intercommunicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute MPI_INIT, every process calls the example function, Init_server(), defined below. Then, if the new_world returned is NULL, the process getting NULL is required to implement a server function, in a reactive loop, Do_server(). Everyone else just does their prescribed computation, using new_world as the new effective "global" communicator. One designated process calls Undo_Server() to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

```
#define INIT_SERVER_TAG_1   666
#define UNDO_SERVER_TAG_1   777

static int server_key_val;

/* for attribute management for server_comm,  copy callback: */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state,
void *attribute_val_in, void **attribute_val_out, int *flag)
{
   /* copy the handle */
   *attribute_val_out = attribute_val_in;
   *flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world;    /* new effective world, sans server */
{
    MPI_Comm temp_comm, lone_comm;
    MPI_Group peer_group, temp_group;
    int rank_in_peer_comm, size, color, key = 0;
    int peer_leader, peer_leader_rank_in_temp_comm;

    MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
    MPI_Comm_size(peer_comm, &size);

    if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
        return (MPI_ERR_OTHER);

    /* create two communicators, by splitting peer_comm
       into the server process, and everyone else */

    peer_leader = (rank_of_server + 1) % size;  /* arbitrary choice */

    if ((color = (rank_in_peer_comm == rank_of_server)))
    {
        MPI_Comm_split(peer_comm, color, key, &lone_comm);

        MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
                           INIT_SERVER_TAG_1, server_comm);

        MPI_Comm_free(&lone_comm);
        *new_world = MPI_COMM_NULL;
    }
    else
    {
        MPI_Comm_Split(peer_comm, color, key, &temp_comm);

        MPI_Comm_group(peer_comm, &peer_group);
        MPI_Comm_group(temp_comm, &temp_group);
        MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
  temp_group, &peer_leader_rank_in_temp_comm);

        MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
                           peer_comm, rank_of_server,
                           INIT_SERVER_TAG_1, server_comm);

        /* attach new_world communication attribute to server_comm: */

        /* CRITICAL SECTION FOR MULTITHREADING */
        if(server_keyval == MPI_KEYVAL_INVALID)
        {
            /* acquire the process-local name for the server keyval */
            MPI_keyval_create(handle_copy_fn, NULL,
                                                &server_keyval, NULL);
        }

        *new_world = temp_comm;
```

218 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
    int peer_leader, peer_leader_rank_in_temp_comm;

    MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
    MPI_Comm_size(peer_comm, &size);

    if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
        return (MPI_ERR_OTHER);

    /* create two communicators, by splitting peer_comm
       into the server process, and everyone else */

    peer_leader = (rank_of_server + 1) % size;  /* arbitrary choice */

    if ((color = (rank_in_peer_comm == rank_of_server)))
    {
        MPI_Comm_split(peer_comm, color, key, &lone_comm);

        MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
                           INIT_SERVER_TAG_1, server_comm);

        MPI_Comm_free(&lone_comm);
        *new_world = MPI_COMM_NULL;
    }
    else
    {
        MPI_Comm_Split(peer_comm, color, key, &temp_comm);

        MPI_Comm_group(peer_comm, &peer_group);
        MPI_Comm_group(temp_comm, &temp_group);
        MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
  temp_group, &peer_leader_rank_in_temp_comm);

        MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
                           peer_comm, rank_of_server,
                           INIT_SERVER_TAG_1, server_comm);

        /* attach new_world communication attribute to server_comm: */

        /* CRITICAL SECTION FOR MULTITHREADING */
        if(server_keyval == MPI_KEYVAL_INVALID)
        {
            /* acquire the process-local name for the server keyval */
            MPI_keyval_create(handle_copy_fn, NULL,
                                                &server_keyval, NULL);
        }

        *new_world = temp_comm;
        /* Cache handle of intra-communicator on inter-communicator: */
        MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
    }

    return (MPI_SUCCESS);
}
```

The actual server process would commit to running the following code:

```
int Do_server(server_comm)
MPI_Comm server_comm;
{
    void init_queue();
    int en_queue(), de_queue(); /* keep triplets of integers
                               for later matching (fns not shown) */

    MPI_Comm comm;
    MPI_Status status;
    int client_tag, client_source;
    int client_rank_in_new_world, pairs_rank_in_new_world;
    int buffer[10], count = 1;

    void *queue;
    init_queue(&queue);


    for (;;)
    {
        MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 server_comm, &status); /* accept from any client */

        /* determine client: */
        client_tag = status.MPI_TAG;
        client_source = status.MPI_SOURCE;
        client_rank_in_new_world = buffer[0];

        if (client_tag == UNDO_SERVER_TAG_1)       /* client that
                                                    terminates server */
        {
            while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world,
                            &pairs_rank_in_server))
                ;

            MPI_Comm_free(&server_comm);
            break;
        }

        if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
```

6.6. INTER-COMMUNICATION 219

```
        /* Cache handle of intra-communicator on inter-communicator: */
        MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
    }

    return (MPI_SUCCESS);
}
```

The actual server process would commit to running the following code:

```
int Do_server(server_comm)
MPI_Comm server_comm;
{
    void init_queue();
    int en_queue(), de_queue(); /* keep triplets of integers
                               for later matching (fns not shown) */

    MPI_Comm comm;
    MPI_Status status;
    int client_tag, client_source;
    int client_rank_in_new_world, pairs_rank_in_new_world;
    int buffer[10], count = 1;

    void *queue;
    init_queue(&queue);


    for (;;)
    {
        MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 server_comm, &status); /* accept from any client */

        /* determine client: */
        client_tag = status.MPI_TAG;
        client_source = status.MPI_SOURCE;
        client_rank_in_new_world = buffer[0];

        if (client_tag == UNDO_SERVER_TAG_1)       /* client that
                                                    terminates server */
        {
            while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world,
                            &pairs_rank_in_server))
                ;

            MPI_Comm_free(&server_comm);
            break;
        }

        if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
                        &pairs_rank_in_server))
        {
            /* matched pair with same tag, tell them
               about each other! */
            buffer[0] = pairs_rank_in_new_world;
            MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
                                                       server_comm);

            buffer[0] = client_rank_in_new_world;
            MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
                     server_comm);
        }
        else
            en_queue(queue, client_tag, client_source,
                                        client_rank_in_new_world);

    }
}
```

A particular process would be responsible for ending the server when it is no longer needed. Its call to Undo_server would terminate server function.

```
int Undo_server(server_comm)     /* example client that ends server */
MPI_Comm *server_comm;
{
    int buffer = 0;
    MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
    MPI_Comm_free(server_comm);
}
```

The following is a blocking name-service for inter-communication, with same semantic restrictions as MPI_Intercomm_create, but simplified syntax. It uses the functionality just defined to create the name service.

```
int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
{
    int error;
    int found;   /* attribute acquisition mgmt for new_world */
                 /* comm in server_comm */
    void *val;

    MPI_Comm new_world;

    int buffer[10], rank;
    int local_leader = 0;
```

220 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
                        &pairs_rank_in_server))
        {
            /* matched pair with same tag, tell them
               about each other! */
            buffer[0] = pairs_rank_in_new_world;
            MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
                                                       server_comm);

            buffer[0] = client_rank_in_new_world;
            MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
                     server_comm);
        }
        else
            en_queue(queue, client_tag, client_source,
                                        client_rank_in_new_world);

    }
}
```

A particular process would be responsible for ending the server when it is no longer needed. Its call to Undo_server would terminate server function.

```
int Undo_server(server_comm)     /* example client that ends server */
MPI_Comm *server_comm;
{
    int buffer = 0;
    MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
    MPI_Comm_free(server_comm);
}
```

The following is a blocking name-service for inter-communication, with same semantic restrictions as MPI_Intercomm_create, but simplified syntax. It uses the functionality just defined to create the name service.

```
int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
{
    int error;
    int found;   /* attribute acquisition mgmt for new_world */
                 /* comm in server_comm */
    void *val;

    MPI_Comm new_world;

    int buffer[10], rank;
    int local_leader = 0;
    MPI_Attr_get(server_comm, server_keyval, &val, &found);
    new_world = (MPI_Comm)val; /* retrieve cached handle */

    MPI_Comm_rank(server_comm, &rank);  /* rank in local group */

    if (rank == local_leader)
    {
        buffer[0] = rank;
        MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
        MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
    }

    error = MPI_Intercomm_create(local_comm, local_leader, new_world,
                                 buffer[0], tag, comm);

    return(error);
}
```

6.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attributing-caching mechanism. (*End of advice to users.*)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

6.7. CACHING 221

```
    MPI_Attr_get(server_comm, server_keyval, &val, &found);
    new_world = (MPI_Comm)val; /* retrieve cached handle */

    MPI_Comm_rank(server_comm, &rank);  /* rank in local group */

    if (rank == local_leader)
    {
        buffer[0] = rank;
        MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
        MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
    }

    error = MPI_Intercomm_create(local_comm, local_leader, new_world,
                                 buffer[0], tag, comm);

    return(error);
}
```

6.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attributing-caching mechanism. (*End of advice to users.*)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. To overcome this problem with attribute caching on communicators, functions are also given for this case. The functions to cache on datatypes and windows also address this issue. For a general discussion of the address size problem, see Section 16.3.6.

> *Advice to implementors.* High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

> *Advice to users.* Attributes in C are of type void *. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

> *Advice to implementors.* Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

> *Advice to implementors.* Caching and callback functions are only called synchronously, in response to explicit application requests. This avoid problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)
>
> The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.
>
> A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid

222 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

One difficulty is the potential for size differences between Fortran integers and C pointers. To overcome this problem with attribute caching on communicators, functions are also given for this case. The functions to cache on datatypes and windows also address this issue. For a general discussion of the address size problem, see Section 16.3.6.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI_COMM_DUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void *. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoid problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid

access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

IN	comm_copy_attr_fn	copy callback function for comm_keyval (function)
IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
             MPI_Comm_delete_attr_function *comm_delete_attr_fn,
             int *comm_keyval, void *extra_state)
```

```
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function*
             comm_copy_attr_fn,
             MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
             void* extra_state)
```

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
             void *attribute_val, void *extra_state);
```

6.7. CACHING 223

access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

IN comm_copy_attr_fn copy callback function for comm_keyval (function)
IN comm_delete_attr_fn delete callback function for comm_keyval (function)
OUT comm_keyval key value for future access (integer)
IN extra_state extra state for callback functions

int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state)

MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
INTEGER COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function* comm_copy_attr_fn, MPI::Comm::Delete_attr_function* comm_delete_attr_fn, void* extra_state)

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);

and

typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state);

which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.

The Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callbacks are:

```
typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
             int comm_keyval, void* extra_state, void* attribute_val_in,
             void* attribute_val_out, bool& flag);
```

and

```
typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
             int comm_keyval, void* attribute_val, void* extra_state);
```

The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns flag = 0, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1), the new attribute value is set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP will fail).

The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C, C++, or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void *, their usage differs. The C copy function is passed by MPI in attribute_val_in the *value* of the attribute, and in attribute_val_out the *address* of the attribute, so as to allow the function to return the (new) attribute value. The use of type void * for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

224 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.

The Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callbacks are:

```
typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
             int comm_keyval, void* extra_state, void* attribute_val_in,
             void* attribute_val_out, bool& flag);
```

and

```
typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
             int comm_keyval, void* attribute_val, void* extra_state);
```

The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns flag = 0, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1), the new attribute value is set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP will fail).

The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C, C++, or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void *, their usage differs. The C copy function is passed by MPI in attribute_val_in the *value* of the attribute, and in attribute_val_out the *address* of the attribute, so as to allow the function to return the (new) attribute value. The use of type void * for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT	comm_keyval	key value (integer)

```
int MPI_Comm_free_keyval(int *comm_keyval)
```

```
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR
```

```
static void MPI::Comm::Free_keyval(int& comm_keyval)
```

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explictly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the new communicator-specific creation function. The use of MPI_KEYVAL_FREE is deprecated.

6.7. CACHING 225

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT comm_keyval key value (integer)

int MPI_Comm_free_keyval(int *comm_keyval)

MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR

static void MPI::Comm::Free_keyval(int& comm_keyval)

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explictly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the new communicator-specific creation function. The use of MPI_KEYVAL_FREE is deprecated.

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT	comm	communicator from which attribute will be attached (handle)
IN	comm_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
```

```
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

```
void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const
```

This function stores the stipulated attribute value attribute_val for subsequent retrieval by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback function comm_delete_attr_fn was executed), and a new value was next stored. The call is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error code other than MPI_SUCCESS.

This function replaces MPI_ATTR_PUT, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)

IN	comm	communicator to which the attribute is attached (handle)
IN	comm_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
              int *flag)
```

```
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

```
bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const
```

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns `flag = false`. In particular MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the *value* of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the *address* of the

226 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT comm communicator from which attribute will be attached (handle)
IN comm_keyval key value (integer)
IN attribute_val attribute value

int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)

MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const

This function stores the stipulated attribute value attribute_val for subsequent retrieval by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback function comm_delete_attr_fn was executed), and a new value was next stored. The call is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error code other than MPI_SUCCESS.

This function replaces MPI_ATTR_PUT, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)

IN comm communicator to which the attribute is attached (handle)
IN comm_keyval key value (integer)
OUT attribute_val attribute value, unless flag = false
OUT flag false if no attribute is associated with the key (logical)

int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
              int *flag)

MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the *value* of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the *address* of the

location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr will be of type void**. (*End of advice to users.*)

Rationale. The use of a formal parameter attribute_val or type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (*End of rationale.*)

This function replaces MPI_ATTR_GET, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_DELETE_ATTR(comm, comm_keyval)

INOUT	comm	communicator from which the attribute is deleted (handle)
IN	comm_keyval	key value (integer)

```
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
```

```
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
```

```
void MPI::Comm::Delete_attr(int comm_keyval)
```

Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.

Whenever a communicator is replicated using the function MPI_COMM_DUP, all callback copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

This function is the same as MPI_ATTR_DELETE but is needed to match the new communicator specific functions. The use of MPI_ATTR_DELETE is deprecated.

6.7.3 Windows

The new functions for caching on windows are:

6.7. CACHING 227

location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr will be of type void**. (*End of advice to users.*)

Rationale. The use of a formal parameter attribute_val or type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (*End of rationale.*)

This function replaces MPI_ATTR_GET, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_DELETE_ATTR(comm, comm_keyval)

INOUT comm communicator from which the attribute is deleted (handle)
IN comm_keyval key value (integer)

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR

void MPI::Comm::Delete_attr(int comm_keyval)

Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.

Whenever a communicator is replicated using the function MPI_COMM_DUP, all callback copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

This function is the same as MPI_ATTR_DELETE but is needed to match the new communicator specific functions. The use of MPI_ATTR_DELETE is deprecated.

6.7.3 Windows

The new functions for caching on windows are:

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for win_keyval (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
             MPI_Win_delete_attr_function *win_delete_attr_fn,
             int *win_keyval, void *extra_state)
```

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function*
             win_copy_attr_fn,
             MPI::Win::Delete_attr_function* win_delete_attr_fn,
             void* extra_state)
```

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C, C++, or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C, C++, or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);
```

The Fortran callback functions are:

```
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

228 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for win_keyval (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
             MPI_Win_delete_attr_function *win_delete_attr_fn,
             int *win_keyval, void *extra_state)
```

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function*
             win_copy_attr_fn,
             MPI::Win::Delete_attr_function* win_delete_attr_fn,
             void* extra_state)
```

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C, C++, or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C, C++, or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);
```

The Fortran callback functions are:

```
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
             IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callbacks are:

```
typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin,
             int win_keyval, void* extra_state, void* attribute_val_in,
             void* attribute_val_out, bool& flag);
```

and

```
typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
             void* attribute_val, void* extra_state);
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is erroneous.

MPI_WIN_FREE_KEYVAL(win_keyval)

INOUT	win_keyval	key value (integer)

```
int MPI_Win_free_keyval(int *win_keyval)
```

```
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
```

```
static void MPI::Win::Free_keyval(int& win_keyval)
```

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)

INOUT	win	window to which attribute will be attached (handle)
IN	win_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
```

```
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

```
void MPI::Win::Set_attr(int win_keyval, const void* attribute_val)
```

6.7. CACHING 229

SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

The C++ callbacks are:
typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin, int win_keyval, void* extra_state, void* attribute_val_in, void* attribute_val_out, bool& flag);

and
typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval, void* attribute_val, void* extra_state);

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is erroneous.

MPI_WIN_FREE_KEYVAL(win_keyval)
INOUT win_keyval key value (integer)

int MPI_Win_free_keyval(int *win_keyval)

MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
INTEGER WIN_KEYVAL, IERROR

static void MPI::Win::Free_keyval(int& win_keyval)

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
INOUT win window to which attribute will be attached (handle)
IN win_keyval key value (integer)
IN attribute_val attribute value

int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)

MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

void MPI::Win::Set_attr(int win_keyval, const void* attribute_val)

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

IN	win	window to which the attribute is attached (handle)
IN	win_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
             int *flag)
```

```
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

```
bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const
```

MPI_WIN_DELETE_ATTR(win, win_keyval)

INOUT	win	window from which the attribute is deleted (handle)
IN	win_keyval	key value (integer)

```
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
```

```
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
```

```
void MPI::Win::Delete_attr(int win_keyval)
```

6.7.4 Datatypes

The new functions for caching on datatypes are:

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

IN	type_copy_attr_fn	copy callback function for type_keyval (function)
IN	type_delete_attr_fn	delete callback function for type_keyval (function)
OUT	type_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
             MPI_Type_delete_attr_function *type_delete_attr_fn,
             int *type_keyval, void *extra_state)
```

```
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
             EXTRA_STATE, IERROR)
```

230 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

IN win window to which the attribute is attached (handle)
IN win_keyval key value (integer)
OUT attribute_val attribute value, unless flag = false
OUT flag false if no attribute is associated with the key (logical)

int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
             int *flag)

MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const

MPI_WIN_DELETE_ATTR(win, win_keyval)

INOUT win window from which the attribute is deleted (handle)
IN win_keyval key value (integer)

int MPI_Win_delete_attr(MPI_Win win, int win_keyval)

MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR

void MPI::Win::Delete_attr(int win_keyval)

6.7.4 Datatypes

The new functions for caching on datatypes are:

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

IN type_copy_attr_fn copy callback function for type_keyval (function)
IN type_delete_attr_fn delete callback function for type_keyval (function)
OUT type_keyval key value for future access (integer)
IN extra_state extra state for callback functions

int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
             MPI_Type_delete_attr_function *type_delete_attr_fn,
             int *type_keyval, void *extra_state)

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
             EXTRA_STATE, IERROR)

```
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
             type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
             type_delete_attr_fn, void* extra_state)
```

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either C, C++, or Fortran. MPI_TYPE_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN from either C, C++, or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
             int type_keyval, void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
             int type_keyval, void *attribute_val, void *extra_state);
```

The Fortran callback functions are:

```
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callbacks are:

```
typedef int MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
             int type_keyval, void* extra_state,
             const void* attribute_val_in, void* attribute_val_out,
             bool& flag);
```

and

```
typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& type,
             int type_keyval, void* attribute_val, void* extra_state);
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),

6.7. CACHING 231

```
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
             type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
             type_delete_attr_fn, void* extra_state)
```

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either C, C++, or Fortran. MPI_TYPE_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN from either C, C++, or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
             int type_keyval, void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
             int type_keyval, void *attribute_val, void *extra_state);
```

The Fortran callback functions are:

```
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

```
SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The C++ callbacks are:

```
typedef int MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
             int type_keyval, void* extra_state,
             const void* attribute_val_in, void* attribute_val_out,
             bool& flag);
```

and

```
typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& type,
             int type_keyval, void* attribute_val, void* extra_state);
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),

is erroneous.

MPI_TYPE_FREE_KEYVAL(type_keyval)

INOUT	type_keyval	key value (integer)

```
int MPI_Type_free_keyval(int *type_keyval)
```

```
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
```

```
static void MPI::Datatype::Free_keyval(int& type_keyval)
```

MPI_TYPE_SET_ATTR(type, type_keyval, attribute_val)

INOUT	type	datatype to which attribute will be attached (handle)
IN	type_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
              void *attribute_val)
```

```
MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

```
void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val)
```

MPI_TYPE_GET_ATTR(type, type_keyval, attribute_val, flag)

IN	type	datatype to which the attribute is attached (handle)
IN	type_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void
              *attribute_val, int *flag)
```

```
MPI_TYPE_GET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

```
bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const
```

232 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

is erroneous.

MPI_TYPE_FREE_KEYVAL(type_keyval)

INOUT type_keyval key value (integer)

int MPI_Type_free_keyval(int *type_keyval)

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR

static void MPI::Datatype::Free_keyval(int& type_keyval)

MPI_TYPE_SET_ATTR(type, type_keyval, attribute_val)

INOUT type datatype to which attribute will be attached (handle)
IN type_keyval key value (integer)
IN attribute_val attribute value

int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
              void *attribute_val)

MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val)

MPI_TYPE_GET_ATTR(type, type_keyval, attribute_val, flag)

IN type datatype to which the attribute is attached (handle)
IN type_keyval key value (integer)
OUT attribute_val attribute value, unless flag = false
OUT flag false if no attribute is associated with the key (logical)

int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void
              *attribute_val, int *flag)

MPI_TYPE_GET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const

MPI_TYPE_DELETE_ATTR(type, type_keyval)

INOUT	type	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)

```
int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)
```

```
MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
```

```
void MPI::Datatype::Delete_attr(int type_keyval)
```

6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last three are included because keyval is an argument to the copy and delete functions for attributes.

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses. (*End of advice to users.*)

```
   /* key for this module's stuff: */
   static int gop_key = MPI_KEYVAL_INVALID;

   typedef struct
   {
      int ref_count;          /* reference count */
      /* other stuff, whatever else we want */
   } gop_stuff_type;

   Efficient_Collective_Op (comm, ...)
   MPI_Comm comm;
   {
     gop_stuff_type *gop_stuff;
     MPI_Group       group;
     int             foundflag;

     MPI_Comm_group(comm, &group);
```

6.7. CACHING 233

MPI_TYPE_DELETE_ATTR(type, type_keyval)

INOUT type datatype from which the attribute is deleted (handle)
IN type_keyval key value (integer)

int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)

MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR

void MPI::Datatype::Delete_attr(int type_keyval)

6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last three are included because keyval is an argument to the copy and delete functions for attributes.

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses. (End of advice to users.)

/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
   int ref_count;   /* reference count */
   /* other stuff, whatever else we want */
} gop_stuff_type;

Efficient_Collective_Op (comm, ...)
MPI_Comm comm;
{
  gop_stuff_type *gop_stuff;
  MPI_Group   group;
  int   foundflag;

  MPI_Comm_group(comm, &group);

```
  if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
  {
    if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
                             gop_stuff_destructor,
                             &gop_key, (void *)0));
    /* get the key while assigning its copy and delete callback
       behavior. */

    MPI_Abort (comm, 99);
  }

  MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
  if (foundflag)
  { /* This module has executed in this group before.
       We will use the cached information */
  }
  else
  { /* This is a group that we have not yet cached anything in.
       We will now do so.
    */

    /* First, allocate storage for the stuff we want,
       and initialize the reference count */

    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }

    gop_stuff -> ref_count = 1;

    /* Second, fill in *gop_stuff with whatever we want.
       This part isn't shown here */

    /* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr ( comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}

/* The following routine is called by MPI when a group is freed */

gop_stuff_destructor (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
{
  if (keyval != gop_key) { /* abort -- programming error */ }
```

234 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
  if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
  {
    if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
                             gop_stuff_destructor,
                             &gop_key, (void *)0));
    /* get the key while assigning its copy and delete callback
       behavior. */

    MPI_Abort (comm, 99);
  }

  MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
  if (foundflag)
  { /* This module has executed in this group before.
       We will use the cached information */
  }
  else
  { /* This is a group that we have not yet cached anything in.
       We will now do so.
    */

    /* First, allocate storage for the stuff we want,
       and initialize the reference count */

    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }

    gop_stuff -> ref_count = 1;

    /* Second, fill in *gop_stuff with whatever we want.
       This part isn't shown here */

    /* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr ( comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}

/* The following routine is called by MPI when a group is freed */

gop_stuff_destructor (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
{
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff -> ref_count -= 1;

  /* If no references remain, then free the storage */
  if (gop_stuff -> ref_count == 0) {
    free((void *)gop_stuff);
  }
}

/* The following routine is called by MPI when a group is copied */
gop_stuff_copier (comm, keyval, extra, gop_stuff_in, gop_stuff_out, flag)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff_in, *gop_stuff_out;
void *extra;
{
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The new group adds one reference to this gop_stuff */
  gop_stuff -> ref_count += 1;
  gop_stuff_out = gop_stuff_in;
}
```

6.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

MPI_COMM_SET_NAME (comm, comm_name)

INOUT	comm	communicator whose identifier is to be set (handle)
IN	comm_name	the character string which is remembered as the name (string)

```
int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)

MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME

void MPI::Comm::Set_name(const char* comm_name)
```

MPI_COMM_SET_NAME allows a user to associate a name string with a communicator. The character string which is passed to MPI_COMM_SET_NAME will be saved inside the

6.8. NAMING OBJECTS 235

```
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff -> ref_count -= 1;

  /* If no references remain, then free the storage */
  if (gop_stuff -> ref_count == 0) {
    free((void *)gop_stuff);
  }
}

/* The following routine is called by MPI when a group is copied */
gop_stuff_copier (comm, keyval, extra, gop_stuff_in, gop_stuff_out, flag)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff_in, *gop_stuff_out;
void *extra;
{
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The new group adds one reference to this gop_stuff */
  gop_stuff -> ref_count += 1;
  gop_stuff_out = gop_stuff_in;
}
```

6.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

MPI_COMM_SET_NAME (comm, comm_name)

INOUT comm communicator whose identifier is to be set (handle)
IN comm_name the character string which is remembered as the name (string)

int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)

MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME

void MPI::Comm::Set_name(const char* comm_name)

MPI_COMM_SET_NAME allows a user to associate a name string with a communicator. The character string which is passed to MPI_COMM_SET_NAME will be saved inside the

MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in name are significant but trailing ones are not.

MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the name of the communicator as seen in the process which made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every process where it exists.

> *Advice to users.* Since MPI_COMM_SET_NAME is provided to help debug code, it is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion. (*End of advice to users.*)

The length of the name which can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. Attempts to put names longer than this will result in truncation of the name. MPI_MAX_OBJECT_NAME must have a value of at least 64.

> *Advice to users.* Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed. (*End of advice to users.*)

> *Advice to implementors.* Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_COMM_GET_NAME. (*End of advice to implementors.*)

MPI_COMM_GET_NAME (comm, comm_name, resultlen)

IN	comm	communicator whose name is to be returned (handle)
OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
```

```
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME
```

```
void MPI::Comm::Get_name(char* comm_name, int& resultlen) const
```

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and got from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters. MPI_COMM_GET_NAME returns a copy of the set name in name. In C, a null character is

236 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in name are significant but trailing ones are not.

MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the name of the communicator as seen in the process which made the MPI_COMM_SET_NAME call. There is no requirement that the same (or any) name be assigned to a communicator in every process where it exists.

Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion. (*End of advice to users.*)

The length of the name which can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. Attempts to put names longer than this will result in truncation of the name. MPI_MAX_OBJECT_NAME must have a value of at least 64.

Advice to users. Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed. (*End of advice to users.*)

Advice to implementors. Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_COMM_GET_NAME. (*End of advice to implementors.*)

MPI_COMM_GET_NAME (comm, comm_name, resultlen)

IN comm communicator whose name is to be returned (handle)
OUT comm_name the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT resultlen length of returned name (integer)

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME

void MPI::Comm::Get_name(char* comm_name, int& resultlen) const

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and got from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters. MPI_COMM_GET_NAME returns a copy of the set name in name. In C, a null character is

additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, `""` in C and C++). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call `strdup` is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(*End of rationale.*)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

The following functions are used for setting and getting names of datatypes.

MPI_TYPE_SET_NAME (type, type_name)

INOUT	type	datatype whose identifier is to be set (handle)
IN	type_name	the character string which is remembered as the name (string)

```
int MPI_Type_set_name(MPI_Datatype type, char *type_name)
```

6.8. NAMING OBJECTS 237

additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, `""` in C and C++). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call `strdup` is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(*End of rationale.*)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

The following functions are used for setting and getting names of datatypes.

MPI_TYPE_SET_NAME (type, type_name)

INOUT type datatype whose identifier is to be set (handle)
IN type_name the character string which is remembered as the name (string)

int MPI_Type_set_name(MPI_Datatype type, char *type_name)

```
MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
    INTEGER TYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```

```
void MPI::Datatype::Set_name(const char* type_name)
```

MPI_TYPE_GET_NAME (type, type_name, resultlen)

IN	type	datatype whose name is to be returned (handle)
OUT	type_name	the name previously stored on the datatype, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)
```

```
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER TYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
```

```
void MPI::Datatype::Get_name(char* type_name, int& resultlen) const
```

Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

The following functions are used for setting and getting names of windows.

MPI_WIN_SET_NAME (win, win_name)

INOUT	win	window whose identifier is to be set (handle)
IN	win_name	the character string which is remembered as the name (string)

```
int MPI_Win_set_name(MPI_Win win, char *win_name)
```

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

```
void MPI::Win::Set_name(const char* win_name)
```

MPI_WIN_GET_NAME (win, win_name, resultlen)

IN	win	window whose name is to be returned (handle)
OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
```

238 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
    INTEGER TYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```

```
void MPI::Datatype::Set_name(const char* type_name)
```

MPI_TYPE_GET_NAME (type, type_name, resultlen)

IN	type	datatype whose name is to be returned (handle)
OUT	type_name	the name previously stored on the datatype, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)
```

```
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER TYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
```

```
void MPI::Datatype::Get_name(char* type_name, int& resultlen) const
```

Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

The following functions are used for setting and getting names of windows.

MPI_WIN_SET_NAME (win, win_name)

INOUT	win	window whose identifier is to be set (handle)
IN	win_name	the character string which is remembered as the name (string)

```
int MPI_Win_set_name(MPI_Win win, char *win_name)
```

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

```
void MPI::Win::Set_name(const char* win_name)
```

MPI_WIN_GET_NAME (win, win_name, resultlen)

IN	win	window whose name is to be returned (handle)
OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
```

```
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME
```

```
void MPI::Win::Get_name(char* win_name, int& resultlen) const
```

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static communicator allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can

6.9. FORMALIZING THE LOOSELY SYNCHRONOUS MODEL 239

MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME

void MPI::Win::Get_name(char* win_name, int& resultlen) const

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static communicator allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can

be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic communicator allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, then communicator creation be properly coordinated.

240 CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic communicator allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, then communicator creation be properly coordinated.

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of `n` processes. Each process in the group is assigned a rank between `0` and `n-1`. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal only with machine-independent mapping.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [32]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [10, 11].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal only with machine-independent mapping.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [32]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [10, 11].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and

241

notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

Rationale. Experience with similar techniques in PARMACS [5, 9] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem then that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord (0,0):	rank 0
coord (0,1):	rank 1
coord (1,0):	rank 2
coord (1,1):	rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

242 CHAPTER 7. PROCESS TOPOLOGIES

notational power in message-passing programming. (*End of rationale.*)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

Rationale. Experience with similar techniques in PARMACS [5, 9] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem then that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

coord (0,0): rank 0
coord (0,1): rank 1
coord (1,0): rank 2
coord (1,1): rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. All input arguments must have identical values on all processes of the group of comm_old. A new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

Rationale. Similar functions are contained in EXPRESS [12] and PARMACS. (*End of rationale.*)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI_CART_SUB is collective over the input communicator's group; all other functions are local.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.7 outlines such an implementation.

7.4. OVERVIEW OF THE FUNCTIONS 243

7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. All input arguments must have identical values on all processes of the group of comm_old. A new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

Rationale. Similar functions are contained in EXPRESS [12] and PARMACS. (*End of rationale.*)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI_CART_SUB is collective over the input communicator's group; all other functions are local.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.7 outlines such an implementation.

7.5 Topology Constructors

7.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	input communicator (handle)
IN	ndims	number of dimensions of Cartesian grid (integer)
IN	dims	integer array of size ndims specifying the number of processes in each dimension
IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator with new Cartesian topology (handle)

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
             int reorder, MPI_Comm *comm_cart)
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

```
MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
             const bool periods[], bool reorder) const
```

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an n-dimensional topology.

244 CHAPTER 7. PROCESS TOPOLOGIES

7.5 Topology Constructors

7.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	input communicator (handle)
IN	ndims	number of dimensions of Cartesian grid (integer)
IN	dims	integer array of size ndims specifying the number of processes in each dimension
IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator with new Cartesian topology (handle)

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
             int reorder, MPI_Comm *comm_cart)
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

```
MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
             const bool periods[], bool reorder) const
```

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an n-dimensional topology.

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	number of nodes in a grid (integer)
IN	ndims	number of Cartesian dimensions (integer)
INOUT	dims	integer array of size `ndims` specifying the number of nodes in each dimension

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

```
void MPI::Compute_dims(int nnodes, int ndims, int dims[])
```

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array `dims`. If `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i] = 0` are modified by the call.

Negative input values of `dims[i]` are erroneous. An error will occur if `nnodes` is not a multiple of $\prod_{i, dims[i] \neq 0} dims[i]$.

For `dims[i]` set by the call, `dims[i]` will be ordered in non-increasing order. Array `dims` is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.

Example 7.1

`dims` before call	function call	`dims` on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

7.5. TOPOLOGY CONSTRUCTORS 245

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	number of nodes in a grid (integer)
IN	ndims	number of Cartesian dimensions (integer)
INOUT	dims	integer array of size `ndims` specifying the number of nodes in each dimension

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

```
void MPI::Compute_dims(int nnodes, int ndims, int dims[])
```

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array `dims`. If `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i] = 0` are modified by the call.

Negative input values of `dims[i]` are erroneous. An error will occur if `nnodes` is not a multiple of $\prod_{i, dims[i] \neq 0} dims[i]$.

For `dims[i]` set by the call, `dims[i]` will be ordered in non-increasing order. Array `dims` is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.

Example 7.1

`dims` before call	function call	`dims` on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

7.5.3 General (Graph) Constructor

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN	comm_old	input communicator (handle)
IN	nnodes	number of nodes in graph (integer)
IN	index	array of integers describing node degrees (see below)
IN	edges	array of integers describing graph edges (see below)
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_graph	communicator with graph topology added (handle)

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
             int reorder, MPI_Comm *comm_graph)
```

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
             IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
```

```
MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[],
             const int edges[], bool reorder) const
```

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes. If the size, nnodes, of the graph is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from `0` to `nnodes-1`. The `i`-th entry of array index stores the total number of neighbors of the first `i` graph nodes. The lists of neighbors of nodes `0, 1, ..., nnodes-1` are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments `nnodes`, `index`, and `edges` are illustrated with the following simple example.

Example 7.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

246 CHAPTER 7. PROCESS TOPOLOGIES

7.5.3 General (Graph) Constructor

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN comm_old input communicator (handle)
IN nnodes number of nodes in graph (integer)
IN index array of integers describing node degrees (see below)
IN edges array of integers describing graph edges (see below)
IN reorder ranking may be reordered (true) or not (false) (logical)
OUT comm_graph communicator with graph topology added (handle)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
             int reorder, MPI_Comm *comm_graph)

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
             IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER

MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[],
             const int edges[], bool reorder) const

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes. If the size, nnodes, of the graph is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 7.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes =	4
index =	2, 3, 4, 6
edges =	1, 3, 0, 3, 0, 2

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for $0 \le \texttt{j} \le \texttt{index}[0] - 1$ and the list of neighbors of node `i`, $\texttt{i} > 0$, is stored in `edges[j]`, $\texttt{index}[\texttt{i}-1] \le \texttt{j} \le \texttt{index}[\texttt{i}] - 1$.

In Fortran, `index(1)` is the degree of node zero, and `index(i+1) - index(i)` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges(j)`, for $1 \le \texttt{j} \le \texttt{index}(1)$ and the list of neighbors of node `i`, $\texttt{i} > 0$, is stored in `edges(j)`, $\texttt{index}(\texttt{i}) + 1 \le \texttt{j} \le \texttt{index}(\texttt{i} + 1)$.

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  1. `ndims` (number of dimensions),
  2. `dims` (numbers of processes per coordinate direction),
  3. `periods` (periodicity information),
  4. `own_position` (own position in grid, could also be computed from rank and dims)
- For a graph topology:
  1. `index`,
  2. `edges`,

  which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (*End of advice to implementors.*)

7.5. TOPOLOGY CONSTRUCTORS 247

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes =	4
index =	2, 3, 4, 6
edges =	1, 3, 0, 3, 0, 2

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for $0 \le \texttt{j} \le \texttt{index}[0] - 1$ and the list of neighbors of node `i`, $\texttt{i} > 0$, is stored in `edges[j]`, $\texttt{index}[\texttt{i}-1] \le \texttt{j} \le \texttt{index}[\texttt{i}] - 1$.

In Fortran, `index(1)` is the degree of node zero, and `index(i+1) - index(i)` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges(j)`, for $1 \le \texttt{j} \le \texttt{index}(1)$ and the list of neighbors of node `i`, $\texttt{i} > 0$, is stored in `edges(j)`, $\texttt{index}(\texttt{i}) + 1 \le \texttt{j} \le \texttt{index}(\texttt{i} + 1)$.

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  1. `ndims` (number of dimensions),
  2. `dims` (numbers of processes per coordinate direction),
  3. `periods` (periodicity information),
  4. `own_position` (own position in grid, could also be computed from rank and dims)
- For a graph topology:
  1. `index`,
  2. `edges`,

  which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (*End of advice to implementors.*)

7.5.4 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator `comm` (state)

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

```
int MPI::Comm::Get_topology() const
```

The function MPI_TOPO_TEST returns the type of topology that is assigned to a communicator.

The output value `status` is one of the following:

MPI_GRAPH	graph topology
MPI_CART	Cartesian topology
MPI_UNDEFINED	no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	comm	communicator for group with graph structure (handle)
OUT	nnodes	number of nodes in graph (integer) (same as number of processes in the group)
OUT	nedges	number of edges in graph (integer)

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

```
void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const
```

Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE.

The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors `index` and `edges` correctly for the following call to MPI_GRAPH_GET.

248 CHAPTER 7. PROCESS TOPOLOGIES

7.5.4 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

IN comm communicator (handle)
OUT status topology type of communicator comm (state)

int MPI_Topo_test(MPI_Comm comm, int *status)

MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR

int MPI::Comm::Get_topology() const

The function MPI_TOPO_TEST returns the type of topology that is assigned to a communicator.

The output value status is one of the following:

MPI_GRAPH graph topology
MPI_CART Cartesian topology
MPI_UNDEFINED no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN comm communicator for group with graph structure (handle)
OUT nnodes number of nodes in graph (integer) (same as number of processes in the group)
OUT nedges number of edges in graph (integer)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR

void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const

Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE.

The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors index and edges correctly for the following call to MPI_GRAPH_GET.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN	comm	communicator with graph structure (handle)
IN	maxindex	length of vector index in the calling program (integer)
IN	maxedges	length of vector edges in the calling program (integer)
OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)
OUT	edges	array of integers containing the graph structure

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
              int *edges)
```

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

```
void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
              int edges[]) const
```

MPI_CARTDIM_GET(comm, ndims)

IN	comm	communicator with Cartesian structure (handle)
OUT	ndims	number of dimensions of the Cartesian structure (integer)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

```
int MPI::Cartcomm::Get_dim() const
```

The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topology information that was associated with a communicator by MPI_CART_CREATE. If comm is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.

7.5. TOPOLOGY CONSTRUCTORS 249

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN	comm	communicator with graph structure (handle)
IN	maxindex	length of vector index in the calling program (integer)
IN	maxedges	length of vector edges in the calling program (integer)
OUT	index	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)
OUT	edges	array of integers containing the graph structure

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
              int *edges)
```

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

```
void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
              int edges[]) const
```

MPI_CARTDIM_GET(comm, ndims)

IN	comm	communicator with Cartesian structure (handle)
OUT	ndims	number of dimensions of the Cartesian structure (integer)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

```
int MPI::Cartcomm::Get_dim() const
```

The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topology information that was associated with a communicator by MPI_CART_CREATE. If comm is associated with a zero-dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN	comm	communicator with Cartesian structure (handle)
IN	maxdims	length of vectors `dims`, `periods`, and `coords` in the calling program (integer)
OUT	dims	number of processes for each Cartesian dimension (array of integer)
OUT	periods	periodicity (true/false) for each Cartesian dimension (array of logical)
OUT	coords	coordinates of calling process in Cartesian structure (array of integer)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)
```

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```

```
void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
              int coords[]) const
```

MPI_CART_RANK(comm, coords, rank)

IN	comm	communicator with Cartesian structure (handle)
IN	coords	integer array (of size `ndims`) specifying the Cartesian coordinates of a process
OUT	rank	rank of specified process (integer)

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
```

```
int MPI::Cartcomm::Get_cart_rank(const int coords[]) const
```

For a process group with Cartesian structure, the function MPI_CART_RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension `i` with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i)` <0 or `coords(i)` $\geq$ `dims(i)`, it is shifted back to the interval $0 \leq$ `coords(i)` $<$ `dims(i)` automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.

250 CHAPTER 7. PROCESS TOPOLOGIES

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN comm communicator with Cartesian structure (handle)
IN maxdims length of vectors dims, periods, and coords in the calling program (integer)
OUT dims number of processes for each Cartesian dimension (array of integer)
OUT periods periodicity (true/false) for each Cartesian dimension (array of logical)
OUT coords coordinates of calling process in Cartesian structure (array of integer)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)

void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
              int coords[]) const

MPI_CART_RANK(comm, coords, rank)

IN comm communicator with Cartesian structure (handle)
IN coords integer array (of size ndims) specifying the Cartesian coordinates of a process
OUT rank rank of specified process (integer)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR

int MPI::Cartcomm::Get_cart_rank(const int coords[]) const

For a process group with Cartesian structure, the function MPI_CART_RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0 or coords(i) ≥ dims(i), it is shifted back to the interval 0 ≤ coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN	comm	communicator with Cartesian structure (handle)
IN	rank	rank of a process within group of comm (integer)
IN	maxdims	length of vector `coords` in the calling program (integer)
OUT	coords	integer array (of size `ndims`) containing the Cartesian coordinates of specified process (array of integers)

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

```
void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const
```

The inverse mapping, rank-to-coordinates translation is provided by MPI_CART_COORDS.

If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
OUT	nneighbors	number of neighbors of specified process (integer)

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

```
int MPI::Graphcomm::Get_neighbors_count(int rank) const
```

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general graph topology.

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	comm	communicator with graph topology (handle)
IN	rank	rank of process in group of comm (integer)
IN	maxneighbors	size of array neighbors (integer)
OUT	neighbors	ranks of processes that are neighbors to specified process (array of integer)

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int *neighbors)
```

7.5. TOPOLOGY CONSTRUCTORS 251

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN comm communicator with Cartesian structure (handle)
IN rank rank of a process within group of comm (integer)
IN maxdims length of vector coords in the calling program (integer)
OUT coords integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers)

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR

void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const

The inverse mapping, rank-to-coordinates translation is provided by MPI_CART_COORDS.

If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN comm communicator with graph topology (handle)
IN rank rank of process in group of comm (integer)
OUT nneighbors number of neighbors of specified process (integer)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR

int MPI::Graphcomm::Get_neighbors_count(int rank) const

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a general graph topology.

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN comm communicator with graph topology (handle)
IN rank rank of process in group of comm (integer)
IN maxneighbors size of array neighbors (integer)
OUT neighbors ranks of processes that are neighbors to specified process (array of integer)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int *neighbors)

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

```
void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int
             neighbors[]) const
```

Example 7.3 Suppose that `comm` is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

node		exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator `comm` has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

7.5.5 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV.

252 CHAPTER 7. PROCESS TOPOLOGIES

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

```
void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int
             neighbors[]) const
```

Example 7.3 Suppose that `comm` is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

node		exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator `comm` has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

7.5.5 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV.

The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	comm	communicator with Cartesian structure (handle)
IN	direction	coordinate dimension of shift (integer)
IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT	rank_source	rank of source process (integer)
OUT	rank_dest	rank of destination process (integer)

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
             int *rank_source, int *rank_dest)
```

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

```
void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source,
             int& rank_dest) const
```

The direction argument indicates the dimension of the shift, i.e., the coordinate which value is modified by the shift. The coordinates are numbered from 0 to `ndims-1`, when `ndims` is the number of dimensions.

Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

Example 7.4 The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of `REAL`s is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr))
C find Cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

7.5. TOPOLOGY CONSTRUCTORS 253

The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	comm	communicator with Cartesian structure (handle)
IN	direction	coordinate dimension of shift (integer)
IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT	rank_source	rank of source process (integer)
OUT	rank_dest	rank of destination process (integer)

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
             int *rank_source, int *rank_dest)
```

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

```
void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source,
             int& rank_dest) const
```

The direction argument indicates the dimension of the shift, i.e., the coordinate which value is modified by the shift. The coordinates are numbered from 0 to `ndims-1`, when `ndims` is the number of dimensions.

Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

Example 7.4 The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of `REAL`s is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr))
C find Cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

7.5.6 Partitioning of Cartesian structures

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	communicator with Cartesian structure (handle)
IN	remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT	newcomm	communicator containing the subgrid that includes the calling process (handle)

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

```
MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const
```

If a Cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 7.5 Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to,

```
MPI_CART_SUB(comm, remain_dims, comm_new),
```

will create three communicators each with eight processes in a 2×4 Cartesian topology. If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

7.5.7 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

254 CHAPTER 7. PROCESS TOPOLOGIES

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

7.5.6 Partitioning of Cartesian structures

MPI_CART_SUB(comm, remain_dims, newcomm)

IN comm communicator with Cartesian structure (handle)
IN remain_dims the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT newcomm communicator containing the subgrid that includes the calling process (handle)

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)

MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const

If a Cartesian topology has been created with MPI_CART_CREATE, the function MPI_CART_SUB can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)

Example 7.5 Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to,

MPI_CART_SUB(comm, remain_dims, comm_new),

will create three communicators each with eight processes in a 2×4 Cartesian topology. If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims, comm_new) will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

7.5.7 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

IN	comm	input communicator (handle)
IN	ndims	number of dimensions of Cartesian structure (integer)
IN	dims	integer array of size `ndims` specifying the number of processes in each coordinate direction
IN	periods	logical array of size `ndims` specifying the periodicity specification in each coordinate direction
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
             int *newrank)
```

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
```

```
int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[])
             const
```

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

> *Advice to implementors.* The function MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart), with `reorder = true` can be implemented by calling MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_cart), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.
>
> The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number encoding of the lost dimensions as `color` and a single number encoding of the preserved dimensions as `key`.
>
> All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

The corresponding new function for general graph structures is as follows.

7.5. TOPOLOGY CONSTRUCTORS 255

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

IN comm input communicator (handle)
IN ndims number of dimensions of Cartesian structure (integer)
IN dims integer array of size ndims specifying the number of processes in each coordinate direction
IN periods logical array of size ndims specifying the periodicity specification in each coordinate direction
OUT newrank reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)

int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
             int *newrank)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)

int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[])
             const

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

Advice to implementors. The function MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart), with reorder = true can be implemented by calling MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank $\neq$ MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

The corresponding new function for general graph structures is as follows.

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

IN	comm	input communicator (handle)
IN	nnodes	number of graph nodes (integer)
IN	index	integer array specifying the graph structure, see MPI_GRAPH_CREATE
IN	edges	integer array specifying the graph structure
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (integer)

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
              int *newrank)
```

```
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

```
int MPI::Graphcomm::Map(int nnodes, const int index[], const int edges[])
              const
```

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with `reorder = true` can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

7.6 An Application Example

Example 7.6 The example in Figure 7.1 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine `relax`.

In each relaxation step each process computes new values for the solution grid function at all points owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor `(i-1,j)`.

256 CHAPTER 7. PROCESS TOPOLOGIES

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

IN comm input communicator (handle)
IN nnodes number of graph nodes (integer)
IN index integer array specifying the graph structure, see MPI_GRAPH_CREATE
IN edges integer array specifying the graph structure
OUT newrank reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (integer)

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
              int *newrank)

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

int MPI::Graphcomm::Map(int nnodes, const int index[], const int edges[])
              const

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with `reorder = true` can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

7.6 An Application Example

Example 7.6 The example in Figure 7.1 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine `relax`.

In each relaxation step each process computes new values for the solution grid function at all points owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor `(i-1,j)`.

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
      integer neigh_rank(num_neigh), own_position(ndims), i, j
      logical periods(ndims)
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_WORLD
C     Set process grid size and periodicity
      call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
      periods(1) = .TRUE.
      periods(2) = .TRUE.
C     Create a grid structure in WORLD group and inquire about own position
      call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
      call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C     Look up the ranks for the neighbors.  Own process coordinates are (i,j).
C     Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
      i = own_position(1)
      j = own_position(2)
      neigh_def(1) = i-1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
        call relax (u, f)
C     Exchange data with neighbor processes
        call exchange (u, comm_cart, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.

7.6. AN APPLICATION EXAMPLE 257

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
      integer neigh_rank(num_neigh), own_position(ndims), i, j
      logical periods(ndims)
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_WORLD
C     Set process grid size and periodicity
      call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
      periods(1) = .TRUE.
      periods(2) = .TRUE.
C     Create a grid structure in WORLD group and inquire about own position
      call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
      call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C     Look up the ranks for the neighbors.  Own process coordinates are (i,j).
C     Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
      i = own_position(1)
      j = own_position(2)
      neigh_def(1) = i-1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
        call relax (u, f)
C     Exchange data with neighbor processes
        call exchange (u, comm_cart, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.

258 CHAPTER 7. PROCESS TOPOLOGIES

Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and run-time ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C and C++,

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 1
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

```
int MPI_Get_version(int *version, int *subversion)
```

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

```
void MPI::Get_version(int& version, int& subversion)
```

Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and run-time ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C and C++,

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 1
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

```
int MPI_Get_version(int *version, int *subversion)
```

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

```
void MPI::Get_version(int& version, int& subversion)
```

259

MPI_GET_VERSION is one of the few functions that can be called before MPI_INIT and after MPI_FINALIZE. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (2,1), (2,0), and (1,2).

8.1.2 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET described in Chapter 6. It is erroneous to delete these attributes, free their keys, or change their values.

The list of predefined attribute keys include

MPI_TAG_UB Upper bound for tag value.

MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.

MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.

MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.

Vendors may add implementation specific parameters (such as node number, real memory size, virtual memory size, etc.)

These predefined attributes do not change value between MPI initialization (MPI_INIT and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.

Advice to users. Note that in the C binding, the value returned by these attributes is a *pointer* to an `int` containing the requested value. (*End of advice to users.*)

The required parameter values are discussed in more detail below:

Tag Values

Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be *at least* 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI_TAG_UB.

The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.

Host Rank

The value returned for MPI_HOST gets the rank of the `HOST` process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a `HOST`, nor does it requires that a `HOST` exists.

The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.

260 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_GET_VERSION is one of the few functions that can be called before MPI_INIT and after MPI_FINALIZE. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (2,1), (2,0), and (1,2).

8.1.2 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET described in Chapter 6. It is erroneous to delete these attributes, free their keys, or change their values.

The list of predefined attribute keys include

MPI_TAG_UB Upper bound for tag value.

MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.

MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.

MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.

Vendors may add implementation specific parameters (such as node number, real memory size, virtual memory size, etc.)

These predefined attributes do not change value between MPI initialization (MPI_INIT and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.

Advice to users. Note that in the C binding, the value returned by these attributes is a *pointer* to an `int` containing the requested value. (*End of advice to users.*)

The required parameter values are discussed in more detail below:

Tag Values

Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be *at least* 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI_TAG_UB.

The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.

Host Rank

The value returned for MPI_HOST gets the rank of the `HOST` process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a `HOST`, nor does it requires that a `HOST` exists.

The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.

IO Rank

The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., `OPEN`, `REWIND`, `WRITE`). For C and C++, this means that all of the ISO C and C++, I/O operations are supported (e.g., `fopen`, `fprintf`, `lseek`).

If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.

Advice to users. Note that input is not collective, and this attribute does *not* indicate which process can or does provide input. (*End of advice to users.*)

Clock Synchronization

The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD.

The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT	name	A unique specifier for the actual (as opposed to virtual) node.
OUT	resultlen	Length (in printable characters) of the result returned in `name`

```
int MPI_Get_processor_name(char *name, int *resultlen)

MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR

void MPI::Get_processor_name(char* name, int& resultlen)
```

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the

8.1. IMPLEMENTATION INFORMATION 261

IO Rank

The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C and C++, this means that all of the ISO C and C++, I/O operations are supported (e.g., fopen, fprintf, lseek).

If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.

Advice to users. Note that input is not collective, and this attribute does not indicate which process can or does provide input. (End of advice to users.)

Clock Synchronization

The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD.

The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT name A unique specifier for the actual (as opposed to virtual) node.
OUT resultlen Length (in printable characters) of the result returned in name

int MPI_Get_processor_name(char *name, int *resultlen)

MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR

void MPI::Get_processor_name(char* name, int& resultlen)

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the

running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead per message buffered by a call to MPI_BSEND (see Section 3.6.1).

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory (see Section 11.4.3.)

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (nonnegative integer)
IN	info	info argument (handle)
OUT	baseptr	pointer to beginning of memory segment allocated

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

```
void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)
```

The info argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL is always valid.

262 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead per message buffered by a call to MPI_BSEND (see Section 3.6.1).

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory (see Section 11.4.3.)

MPI_ALLOC_MEM(size, info, baseptr)

IN size size of memory segment in bytes (nonnegative integer)
IN info info argument (handle)
OUT baseptr pointer to beginning of memory segment allocated

int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)

The info argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL is always valid.

The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM to indicate it failed because memory is exhausted.

MPI_FREE_MEM(base)

IN	base	initial address of memory segment allocated by MPI_ALLOC_MEM (choice)

```
int MPI_Free_mem(void *base)
```

```
MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR
```

```
void MPI::Free_mem(void *base)
```

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

Rationale. The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the `malloc` and `free` C library calls: a call to MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (*End of rationale.*)

Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C `malloc` and `free` functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes `malloc`, and MPI_FREE_MEM invokes `free`.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALs, and assume that pointers are address-sized.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

8.2. MEMORY ALLOCATION 263

The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM to indicate it failed because memory is exhausted.

MPI_FREE_MEM(base)

IN base initial address of memory segment allocated by MPI_ALLOC_MEM (choice)

int MPI_Free_mem(void *base)

MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR

void MPI::Free_mem(void *base)

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

Rationale. The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the `malloc` and `free` C library calls: a call to MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (*End of rationale.*)

Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C `malloc` and `free` functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes `malloc`, and MPI_FREE_MEM invokes `free`.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALs, and assume that pointers are address-sized.

REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed

Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran compilers for Intel) do not support this code.

Example 8.2 Same example, in C

```
float  (* f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI **exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM-_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

264 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran compilers for Intel) do not support this code.

Example 8.2 Same example, in C

```
float  (* f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI **exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM-_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C and C++ have distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER(function, errhandler), where XXX is, respectively, COMM, WIN, or FILE.

An error handler is attached to a communicator, window, or file by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C++, the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

8.3.1 Error Handlers for Communicators

8.3. ERROR HANDLING 265

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C and C++ have distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER(function, errhandler), where XXX is, respectively, COMM, WIN, or FILE.

An error handler is attached to a communicator, window, or file by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C++, the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

8.3.1 Error Handlers for Communicators

MPI_COMM_CREATE_ERRHANDLER(function, errhandler)

IN	function	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function,
             MPI_Errhandler *errhandler)
```

```
MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

```
static MPI::Errhandler
             MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_fn*
             function)
```

Creates an error handler that can be attached to communicators. This function is identical to MPI_ERRHANDLER_CREATE, whose use is deprecated.

The user routine should be, in C, a function of type MPI_Comm_errhandler_fn, which is defined as

```
typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "`stdargs`" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. This typedef replaces MPI_Handler_function, whose use is deprecated.

In Fortran, the user routine should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...)
    INTEGER COMM, ERROR_CODE
```

Advice to users. Users are discouraged from using a Fortran {COMM|WIN|FILE}_ERRHANDLER_FN since the routine expects a variable number of arguments. Some Fortran systems may allow this but some may fail to give the correct result or compile/link this code. Thus, it will not, in general, be possible to create portable code with a Fortran {COMM|WIN|FILE}_ERRHANDLER_FN. (*End of advice to users.*)

In C++, the user routine should be of the form:

```
typedef void MPI::Comm::Errhandler_fn(MPI::Comm &, int *, ...);
```

Rationale. The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify

266 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_COMM_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)
OUT errhandler MPI error handler (handle)

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function,
             MPI_Errhandler *errhandler)

MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

static MPI::Errhandler
             MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_fn*
             function)

Creates an error handler that can be attached to communicators. This function is identical to MPI_ERRHANDLER_CREATE, whose use is deprecated.

The user routine should be, in C, a function of type MPI_Comm_errhandler_fn, which is defined as
typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "stdargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. This typedef replaces MPI_Handler_function, whose use is deprecated.

In Fortran, the user routine should be of the form:
SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...)
    INTEGER COMM, ERROR_CODE

Advice to users. Users are discouraged from using a Fortran {COMM|WIN|FILE}_ERRHANDLER_FN since the routine expects a variable number of arguments. Some Fortran systems may allow this but some may fail to give the correct result or compile/link this code. Thus, it will not, in general, be possible to create portable code with a Fortran {COMM|WIN|FILE}_ERRHANDLER_FN. (*End of advice to users.*)

In C++, the user routine should be of the form:
typedef void MPI::Comm::Errhandler_fn(MPI::Comm &, int *, ...);

Rationale. The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify

a “global” error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT	comm	communicator (handle)
IN	errhandler	new error handler for communicator (handle)

```
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

```
void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler)
```

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN	comm	communicator (handle)
OUT	errhandler	error handler currently associated with communicator (handle)

```
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

```
MPI::Errhandler MPI::Comm::Get_errhandler() const
```

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

IN	function	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

8.3. ERROR HANDLING 267

a “global” error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT comm communicator (handle)
IN errhandler new error handler for communicator (handle)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN comm communicator (handle)
OUT errhandler error handler currently associated with communicator (handle)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI::Errhandler MPI::Comm::Get_errhandler() const

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)
OUT errhandler MPI error handler (handle)

```
int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function,
             MPI_Errhandler *errhandler)
```

```
MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

```
static MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn*
             function)
```

Creates an error handler that can be attached to a window object. The user routine should be, in C, a function of type MPI_Win_errhandler_fn, which is defined as
`typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);`

The first argument is the window in use, the second is the error code to be returned.

In Fortran, the user routine should be of the form:

```
SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...)
    INTEGER WIN, ERROR_CODE
```

In C++, the user routine should be of the form:
`typedef void MPI::Win::Errhandler_fn(MPI::Win &, int *, ...);`

MPI_WIN_SET_ERRHANDLER(win, errhandler)

INOUT	win	window (handle)
IN	errhandler	new error handler for window (handle)

```
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

```
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

```
void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)
```

Attaches a new error handler to a window. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_WIN_CREATE_ERRHANDLER.

MPI_WIN_GET_ERRHANDLER(win, errhandler)

IN	win	window (handle)
OUT	errhandler	error handler currently associated with window (handle)

```
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
```

```
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

```
MPI::Errhandler MPI::Win::Get_errhandler() const
```

268 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function,
             MPI_Errhandler *errhandler)

MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

static MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn*
             function)

Creates an error handler that can be attached to a window object. The user routine should be, in C, a function of type MPI_Win_errhandler_fn, which is defined as
typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);

The first argument is the window in use, the second is the error code to be returned.

In Fortran, the user routine should be of the form:
SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...)
    INTEGER WIN, ERROR_CODE

In C++, the user routine should be of the form:
typedef void MPI::Win::Errhandler_fn(MPI::Win &, int *, ...);

MPI_WIN_SET_ERRHANDLER(win, errhandler)
INOUT win window (handle)
IN errhandler new error handler for window (handle)

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)

MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR

void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a window. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_WIN_CREATE_ERRHANDLER.

MPI_WIN_GET_ERRHANDLER(win, errhandler)
IN win window (handle)
OUT errhandler error handler currently associated with window (handle)

int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR

MPI::Errhandler MPI::Win::Get_errhandler() const

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)

IN	function	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_File_create_errhandler(MPI_File_errhandler_fn *function,
             MPI_Errhandler *errhandler)
```

```
MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

```
static MPI::Errhandler
             MPI::File::Create_errhandler(MPI::File::Errhandler_fn*
             function)
```

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_fn, which is defined as
`typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);`

The first argument is the file in use, the second is the error code to be returned.

In Fortran, the user routine should be of the form:

```
SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)
    INTEGER FILE, ERROR_CODE
```

In C++, the user routine should be of the form:
`typedef void MPI::File::Errhandler_fn(MPI::File &, int *, ...);`

MPI_FILE_SET_ERRHANDLER(file, errhandler)

INOUT	file	file (handle)
IN	errhandler	new error handler for file (handle)

```
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
```

```
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

```
void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)
```

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.

8.3. ERROR HANDLING 269

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)
IN function user defined error handling procedure (function)
OUT errhandler MPI error handler (handle)

int MPI_File_create_errhandler(MPI_File_errhandler_fn *function,
MPI_Errhandler *errhandler)

MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
EXTERNAL FUNCTION
INTEGER ERRHANDLER, IERROR

static MPI::Errhandler
MPI::File::Create_errhandler(MPI::File::Errhandler_fn*
function)

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_fn, which is defined as
typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);

The first argument is the file in use, the second is the error code to be returned.
In Fortran, the user routine should be of the form:
SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)
INTEGER FILE, ERROR_CODE

In C++, the user routine should be of the form:
typedef void MPI::File::Errhandler_fn(MPI::File &, int *, ...);

MPI_FILE_SET_ERRHANDLER(file, errhandler)
INOUT file file (handle)
IN errhandler new error handler for file (handle)

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
INTEGER FILE, ERRHANDLER, IERROR

void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.

MPI_FILE_GET_ERRHANDLER(file, errhandler)

IN	file	file (handle)
OUT	errhandler	error handler currently associated with file (handle)

```
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
```

```
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

```
MPI::Errhandler MPI::File::Get_errhandler() const
```

Retrieves the error handler currently associated with a file.

8.3.4 Freeing Errorhandlers and Retrieving Error Strings

MPI_ERRHANDLER_FREE(errhandler)

INOUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
```

```
void MPI::Errhandler::Free()
```

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

IN	errorcode	Error code returned by an MPI routine
OUT	string	Text that corresponds to the errorcode
OUT	resultlen	Length (in printable characters) of the result returned in string

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
```

```
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
```

```
void MPI::Get_error_string(int errorcode, char* name, int& resultlen)
```

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

270 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_FILE_GET_ERRHANDLER(file, errhandler)

IN file file (handle)
OUT errhandler error handler currently associated with file (handle)

int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)

MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR

MPI::Errhandler MPI::File::Get_errhandler() const

Retrieves the error handler currently associated with a file.

8.3.4 Freeing Errorhandlers and Retrieving Error Strings

MPI_ERRHANDLER_FREE(errhandler)

INOUT errhandler MPI error handler (handle)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR

void MPI::Errhandler::Free()

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

IN errorcode Error code returned by an MPI routine
OUT string Text that corresponds to the errorcode
OUT resultlen Length (in printable characters) of the result returned in string

int MPI_Error_string(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING

void MPI::Get_error_string(int errorcode, char* name, int& resultlen)

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

$$0 = \text{MPI\_SUCCESS} < \text{MPI\_ERR\_...} \leq \text{MPI\_ERR\_LASTCODE}.$$

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

MPI_ERROR_CLASS(errorcode, errorclass)

IN	errorcode	Error code returned by an MPI routine
OUT	errorclass	Error class associated with errorcode

```
int MPI_Error_class(int errorcode, int *errorclass)
```

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

```
int MPI::Get_error_class(int errorcode)
```

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.

8.4. ERROR CODES AND CLASSES 271

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

0 = MPI_SUCCESS < MPI_ERR_... ≤ MPI_ERR_LASTCODE.

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

MPI_ERROR_CLASS(errorcode, errorclass)

IN errorcode Error code returned by an MPI routine
OUT errorclass Error class associated with errorcode

int MPI_Error_class(int errorcode, int *errorclass)

MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR

int MPI::Get_error_class(int errorcode)

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_PENDING	Pending request
MPI_ERR_KEYVAL	Invalid keyval has been passed
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_SPAWN	Error in spawning processes
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_WIN	Invalid win argument
MPI_ERR_SIZE	Invalid size argument
MPI_ERR_DISP	Invalid disp argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_ASSERT	Invalid assert argument
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls

Table 8.1: Error classes (Part 1)

272 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_PENDING	Pending request
MPI_ERR_KEYVAL	Invalid keyval has been passed
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_SPAWN	Error in spawning processes
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_WIN	Invalid win argument
MPI_ERR_SIZE	Invalid size argument
MPI_ERR_DISP	Invalid disp argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_ASSERT	Invalid assert argument
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls

Table 8.1: Error classes (Part 1)

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.
MPI_ERR_IO	Other I/O error
MPI_ERR_LASTCODE	Last error code

Table 8.2: Error classes (Part 2)

8.5 Error Classes, Error Codes, and Error Handlers

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 13 on page 373. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
3. associate strings with these error codes, so that MPI_ERROR_STRING works.
4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes: it is not expected that an application will generate them in significant numbers.

8.5. ERROR CLASSES, ERROR CODES, AND ERROR HANDLERS 273

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.
MPI_ERR_IO	Other I/O error
MPI_ERR_LASTCODE	Last error code

Table 8.2: Error classes (Part 2)

8.5 Error Classes, Error Codes, and Error Handlers

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 13 on page 373. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
3. associate strings with these error codes, so that MPI_ERROR_STRING works.
4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes: it is not expected that an application will generate them in significant numbers.

MPI_ADD_ERROR_CLASS(errorclass)

OUT	errorclass	value for the new error class (integer)

```
int MPI_Add_error_class(int *errorclass)
```

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
```

```
int MPI::Add_error_class()
```

Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorclass in the same deterministic way on all processes. (*End of advice to implementors.*)

Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (*End of advice to users.*)

274 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_ADD_ERROR_CLASS(errorclass)

OUT errorclass value for the new error class (integer)

int MPI_Add_error_class(int *errorclass)

MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR

int MPI::Add_error_class()

Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorclass in the same deterministic way on all processes. (*End of advice to implementors.*)

Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (*End of advice to users.*)

MPI_ADD_ERROR_CODE(errorclass, errorcode)

IN	errorclass	error class (integer)
OUT	errorcode	new error code to associated with errorclass (integer)

```
int MPI_Add_error_code(int errorclass, int *errorcode)
```

```
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
```

```
int MPI::Add_error_code(int errorclass)
```

Creates new error code associated with errorclass and returns its value in errorcode.

Rationale. To avoid conflicts with existing error codes and classes, the value of the new error code is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorcode in the same deterministic way on all processes. (*End of advice to implementors.*)

MPI_ADD_ERROR_STRING(errorcode, string)

IN	errorcode	error code or class (integer)
IN	string	text corresponding to errorcode (string)

```
int MPI_Add_error_string(int errorcode, char *string)
```

```
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
```

```
void MPI::Add_error_string(int errorcode, const char* string)
```

Associates an error string with an error code or class. The string must be no more than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the calling language. The length of the string does not include the null terminator in C or C++. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that already has a string will replace the old string with the new string. It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value $\leq$ MPI_ERR_LASTCODE.

If MPI_ERROR_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C and C++).

Section 8.3 on page 264 describes the methods for creating and associating error handlers with communicators, files, and windows.

8.5. ERROR CLASSES, ERROR CODES, AND ERROR HANDLERS 275

MPI_ADD_ERROR_CODE(errorclass, errorcode)

IN	errorclass	error class (integer)
OUT	errorcode	new error code to associated with errorclass (integer)

```
int MPI_Add_error_code(int errorclass, int *errorcode)
```

```
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
```

```
int MPI::Add_error_code(int errorclass)
```

Creates new error code associated with errorclass and returns its value in errorcode.

Rationale. To avoid conflicts with existing error codes and classes, the value of the new error code is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorcode in the same deterministic way on all processes. (*End of advice to implementors.*)

MPI_ADD_ERROR_STRING(errorcode, string)

IN	errorcode	error code or class (integer)
IN	string	text corresponding to errorcode (string)

```
int MPI_Add_error_string(int errorcode, char *string)
```

```
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
```

```
void MPI::Add_error_string(int errorcode, const char* string)
```

Associates an error string with an error code or class. The string must be no more than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the calling language. The length of the string does not include the null terminator in C or C++. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that already has a string will replace the old string with the new string. It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value $\leq$ MPI_ERR_LASTCODE.

If MPI_ERROR_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C and C++).

Section 8.3 on page 264 describes the methods for creating and associating error handlers with communicators, files, and windows.

MPI_COMM_CALL_ERRHANDLER (comm, errorcode)

IN	comm	communicator with error handler (handle)
IN	errorcode	error code (integer)

```
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

void MPI::Comm::Call_errhandler(int errorcode) const
```

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users should note that the default error handler is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will abort the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (*End of advice to users.*)

MPI_WIN_CALL_ERRHANDLER (win, errorcode)

IN	win	window with error handler (handle)
IN	errorcode	error code (integer)

```
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

void MPI::Win::Call_errhandler(int errorcode) const
```

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. As with communicators, the default error handler for windows is MPI_ERRORS_ARE_FATAL. (*End of advice to users.*)

MPI_FILE_CALL_ERRHANDLER (fh, errorcode)

IN	fh	file with error handler (handle)
IN	errorcode	error code (integer)

276 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_COMM_CALL_ERRHANDLER (comm, errorcode)

IN comm communicator with error handler (handle)
IN errorcode error code (integer)

int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

void MPI::Comm::Call_errhandler(int errorcode) const

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users should note that the default error handler is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will abort the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (*End of advice to users.*)

MPI_WIN_CALL_ERRHANDLER (win, errorcode)

IN win window with error handler (handle)
IN errorcode error code (integer)

int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

void MPI::Win::Call_errhandler(int errorcode) const

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. As with communicators, the default error handler for windows is MPI_ERRORS_ARE_FATAL. (*End of advice to users.*)

MPI_FILE_CALL_ERRHANDLER (fh, errorcode)

IN fh file with error handler (handle)
IN errorcode error code (integer)

```
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
```

```
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR
```

```
void MPI::File::Call_errhandler(int errorcode) const
```

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Unlike errors on communicators and windows, the default behavior for files is to have MPI_ERRORS_RETURN. (*End of advice to users.*)

Advice to users. Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (*End of advice to users.*)

8.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section 2.6.5 on page 21.

MPI_WTIME()

```
double MPI_Wtime(void)
```

```
DOUBLE PRECISION MPI_WTIME()
```

```
double MPI::Wtime()
```

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

8.6. TIMERS AND SYNCHRONIZATION 277

int MPI_File_call_errhandler(MPI_File fh, int errorcode)

MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
INTEGER FH, ERRORCODE, IERROR

void MPI::File::Call_errhandler(int errorcode) const

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Unlike errors on communicators and windows, the default behavior for files is to have MPI_ERRORS_RETURN. (End of advice to users.)

Advice to users. Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)

8.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section 2.6.5 on page 21.

MPI_WTIME()

double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()

double MPI::Wtime()

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI_WTIME_IS_GLOBAL).

MPI_WTICK()

```
double MPI_Wtick(void)
```

```
DOUBLE PRECISION MPI_WTICK()
```

```
double MPI::Wtick()
```

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3}.

8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

```
void MPI::Init(int& argc, char**& argv)
```

```
void MPI::Init()
```

278 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI_WTIME_IS_GLOBAL).

MPI_WTICK()

```
double MPI_Wtick(void)
```

```
DOUBLE PRECISION MPI_WTICK()
```

```
double MPI::Wtick()
```

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3}.

8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

```
void MPI::Init(int& argc, char**& argv)
```

```
void MPI::Init()
```

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI_INITIALIZED).

All MPI programs must contain a call to MPI_INIT; this routine must be called before any other MPI routine (apart from MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED) is called. The version for ISO C accepts the argc and argv that are provided by the arguments to `main`:

```
int main(argc, argv)
int argc;
char **argv;
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C and C++. In C++, there is an alternative binding for MPI::Init that does not have these arguments at all.

> *Rationale.* In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by `mpiexec` can get that information from environment variables. (*End of rationale.*)

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

```
void MPI::Finalize()
```

This routine cleans up all MPI state. Each process must call MPI_FINALIZE before it exits. Unless there has been a call to MPI_ABORT, each process must ensure that all pending non-blocking communications are (locally) complete before calling MPI_FINALIZE. Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct:

```
        Process 0                Process 1
        ---------                ---------
        MPI_Init();              MPI_Init();
```

8.7. STARTUP 279

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI_INITIALIZED).

All MPI programs must contain a call to MPI_INIT; this routine must be called before any other MPI routine (apart from MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED) is called. The version for ISO C accepts the argc and argv that are provided by the arguments to `main`:

```
int main(argc, argv)
int argc;
char **argv;
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C and C++. In C++, there is an alternative binding for MPI::Init that does not have these arguments at all.

> *Rationale.* In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by `mpiexec` can get that information from environment variables. (*End of rationale.*)

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

```
void MPI::Finalize()
```

This routine cleans up all MPI state. Each process must call MPI_FINALIZE before it exits. Unless there has been a call to MPI_ABORT, each process must ensure that all pending non-blocking communications are (locally) complete before calling MPI_FINALIZE. Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct:

```
        Process 0                Process 1
        ---------                ---------
        MPI_Init();              MPI_Init();
        MPI_Send(dest=1);         MPI_Recv(src=0);
        MPI_Finalize();           MPI_Finalize();
```

Without the matching receive, the program is erroneous:

```
        Process 0                 Process 1
        -----------               -----------
        MPI_Init();               MPI_Init();
        MPI_Send (dest=1);
        MPI_Finalize();           MPI_Finalize();
```

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The MPI_ISEND is complete only when it is known by some means that a matching receive has completed. MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE combined with some other verification of completion).

Example 8.3 This program is correct:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Barrier();
MPI_Barrier();                  MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

Example 8.4 This program is erroneous and its behavior is undefined:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH.

Example 8.5 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached.

280 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

```
        MPI_Send(dest=1);         MPI_Recv(src=0);
        MPI_Finalize();           MPI_Finalize();
```

Without the matching receive, the program is erroneous:

```
        Process 0                 Process 1
        -----------               -----------
        MPI_Init();               MPI_Init();
        MPI_Send (dest=1);
        MPI_Finalize();           MPI_Finalize();
```

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The MPI_ISEND is complete only when it is known by some means that a matching receive has completed. MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE combined with some other verification of completion).

Example 8.3 This program is correct:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Barrier();
MPI_Barrier();                  MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

Example 8.4 This program is erroneous and its behavior is undefined:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH.

Example 8.5 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached.

```
rank 0                          rank 1
=====================================================
...                             ...
buffer = malloc(1000000);       MPI_Recv();
MPI_Buffer_attach();            MPI_Finalize();
MPI_Bsend();                    exit();
MPI_Finalize();
free(buffer);
exit();
```

Example 8.6 In this example, MPI_Iprobe() must return a FALSE flag. MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1.

The MPI_Iprobe() call is there to make sure the implementation knows that the "tag1" message exists at the destination, without being able to claim that the user knows about it.

```
rank 0                          rank 1
========================================================
MPI_Init();                     MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Iprobe(tag2);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Finalize();
                                exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE until all potential future message cancellations have been processed. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 318.

Advice to implementors. Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the

8.7. STARTUP 281

```
rank 0                          rank 1
=====================================================
...                             ...
buffer = malloc(1000000);       MPI_Recv();
MPI_Buffer_attach();            MPI_Finalize();
MPI_Bsend();                    exit();
MPI_Finalize();
free(buffer);
exit();
```

Example 8.6 In this example, MPI_Iprobe() must return a FALSE flag. MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1.

The MPI_Iprobe() call is there to make sure the implementation knows that the "tag1" message exists at the destination, without being able to claim that the user knows about it.

```
rank 0                          rank 1
========================================================
MPI_Init();                     MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Iprobe(tag2);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Finalize();
                                exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE until all potential future message cancellations have been processed. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 318.

Advice to implementors. Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the

underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.7 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile","w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

MPI_INITIALIZED(flag)

OUT	flag	Flag is true if MPI_INIT has been called and false otherwise.

```
int MPI_Initialized(int *flag)
```

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

```
bool MPI::Is_initialized()
```

This routine may be used to determine whether MPI_INIT has been called. MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one of the few routines that may be called before MPI_INIT is called.

282 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.7 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile","w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

MPI_INITIALIZED(flag)

OUT flag Flag is true if MPI_INIT has been called and false otherwise.

```
int MPI_Initialized(int *flag)
```

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

```
bool MPI::Is_initialized()
```

This routine may be used to determine whether MPI_INIT has been called. MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one of the few routines that may be called before MPI_INIT is called.

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

```
void MPI::Comm::Abort(int errorcode)
```

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. If no processes were spawned, accepted or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (*End of rationale.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., `mpiexec`), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. `mpiexec` or singleton init). (*End of advice to implementors.*)

8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in an arbitrary order. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

8.7. STARTUP 283

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

```
void MPI::Comm::Abort(int errorcode)
```

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. If no processes were spawned, accepted or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (*End of rationale.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., `mpiexec`), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. `mpiexec` or singleton init). (*End of advice to implementors.*)

8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in an arbitrary order. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. (*End of advice to implementors.*)

8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

MPI_FINALIZED(flag)

OUT	flag	true if MPI was finalized (logical)

```
int MPI_Finalized(int *flag)
```

```
MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

```
bool MPI::Is_finalized()
```

This routine returns true if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI_INIT *has* completed and MPI_FINALIZE *has not* completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invoked during MPI_FINALIZE. (*End of advice to users.*)

8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with

284 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. (*End of advice to implementors.*)

8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

MPI_FINALIZED(flag)

OUT flag true if MPI was finalized (logical)

int MPI_Finalized(int *flag)

MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

bool MPI::Is_finalized()

This routine returns true if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI_INIT *has* completed and MPI_FINALIZE *has not* completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invoked during MPI_FINALIZE. (*End of advice to users.*)

8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

mpirun <mpirun arguments> <program> <program arguments>

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with

existing practice, which is not standard and not portable among implementations, instead of `mpirun` MPI specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.3.4).

Analogous to MPI_COMM_SPAWN, we have

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

for the case where a single command line for the application program and its arguments will suffice. See Section 10.3.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the `-n x` argument is optional; the default is implementation dependent. It might be `1`, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the `info` argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

8.8. PORTABLE MPI PROCESS STARTUP 285

existing practice, which is not standard and not portable among implementations, instead of `mpirun` MPI specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.3.4).

Analogous to MPI_COMM_SPAWN, we have

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

for the case where a single command line for the application program and its arguments will suffice. See Section 10.3.4 for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the `-n x` argument is optional; the default is implementation dependent. It might be `1`, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the `info` argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

```
mpiexec -configfile <filename>
```

where the lines of <`filename`> are of the form separated by the colons in Form A. Lines beginning with '`#`' are comments, and lines may be continued by terminating the partial line with '\'.

Example 8.8 Start 16 instances of `myprog` on the current or default machine:

```
mpiexec -n 16 myprog
```

Example 8.9 Start 10 processes on the machine called `ferrari`:

```
mpiexec -n 10 -host ferrari myprog
```

Example 8.10 Start three copies of the same program with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 8.11 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's:

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 8.12 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's (Form B):

```
mpiexec -configfile myfile
```

where `myfile` contains

```
-n 5  -arch sun    ocean
-n 10 -arch rs6000 atmos
```

(*End of advice to implementors.*)

286 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

mpiexec -configfile <filename>

where the lines of <filename> are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with '\'.

Example 8.8 Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

Example 8.9 Start 10 processes on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

Example 8.10 Start three copies of the same program with different command-line arguments:

mpiexec myprog infile1 : myprog infile2 : myprog infile3

Example 8.11 Start the ocean program on five Suns and the atmos program on 10 RS/6000's:

mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 8.12 Start the ocean program on five Suns and the atmos program on 10 RS/6000's (Form B):

mpiexec -configfile myfile

where myfile contains

-n 5 -arch sun   ocean
-n 10 -arch rs6000 atmos

(*End of advice to implementors.*)

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C/C++ examples.

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C/C++ examples.

Attributes between languages, 487

C++ declarations in `mpi.h`, 460
C++ deriving from C++ MPI class, 451
C++ handle assignement and comparison operators, 456
C++ handle assignment operator, 452
C++ handle scope destruction, 452
C++ illegal communicator handle initialization, 456
C++ MPI class comparison operator, 451
C++ profiling example, 460, 461
C/C++ handle conversion, 481, 482
C/Fortran handle conversion, 480
Client-server code, 62, 63
Client-server code with blocking probe, 66
Client-server code with blocking probe, wrong, 66

Datatype - 3D array, 111
Datatype - absolute addresses, 117
Datatype - array of structures, 114
Datatype - elaborate example, 124, 125
Datatype - matching type, 101
Datatype - matrix transpose, 112, 113
Datatype - union, 118
Datatypes - matching, 35
Datatypes - not matching, 35
Datatypes - untyped, 35
Deadlock with MPI_Bcast, 177, 178
Deadlock, if not buffered, 44
Deadlock, wrong message exchange, 44

Fortran 90 copying and sequence problem, 463, 464
Fortran 90 derived types, 465
Fortran 90 heterogeneous communication, 476, 477
Fortran 90 illegal KIND, 472
Fortran 90 MPI_TYPE_MATCH_SIZE implementation, 476
Fortran 90 register optimization, 467

Intercommunicator, 198, 199
Interlanguage communication, 489
Intertwined matching pairs, 42

Message exchange, 43
MPI::Comm::Probe, 33
MPI_ACCUMULATE, 332
MPI_ADDRESS, 95
MPI_Address, 114, 117, 118, 124
MPI_Aint, 114
MPI_Allgather, 154
MPI_ALLOC_MEM, 263
MPI_Alloc_mem, 264
MPI_ALLREDUCE, 172
MPI_Barrier, 280, 281, 347
MPI_Bcast, 136, 177, 178
MPI_BSEND, 42
MPI_Buffer_attach, 46, 280
MPI_Buffer_detach, 46
MPI_BYTE, 35
MPI_Cancel, 281
MPI_CART_COORDS, 253
MPI_CART_GET, 256
MPI_CART_RANK, 256
MPI_CART_SHIFT, 253
MPI_CART_SUB, 254
MPI_CHARACTER, 36

571

572 *Examples Index*

MPI_Comm_create, 198
MPI_Comm_group, 198
MPI_Comm_remote_size, 199
MPI_COMM_SPAWN, 298
MPI_Comm_spawn, 298
MPI_COMM_SPAWN_MULTIPLE, 303
MPI_Comm_spawn_multiple, 303
MPI_Comm_split, 199
MPI_DIMS_CREATE, 245, 256
MPI_FILE_CLOSE, 394, 397
MPI_FILE_GET_AMODE, 381
MPI_FILE_IREAD, 397
MPI_FILE_OPEN, 394, 397
MPI_FILE_READ, 394
MPI_FILE_SET_ATOMICITY, 425
MPI_FILE_SET_VIEW, 394, 397
MPI_FILE_SYNC, 426
MPI_Finalize, 280–282
MPI_FREE_MEM, 263
MPI_Gather, 125, 140, 141, 145
MPI_Gatherv, 125, 142–145
MPI_GET, 328, 330
MPI_Get, 346, 347
MPI_GET_ADDRESS, 95, 484
MPI_Get_address, 114, 117, 118, 124
MPI_GET_COUNT, 103
MPI_GET_ELEMENTS, 103
MPI_GRAPH_CREATE, 246, 252
MPI_Grequest_complete, 361
MPI_Grequest_start, 361
MPI_Group_free, 198
MPI_Group_incl, 198
MPI_Iprobe, 281
MPI_IRECV, 54–56, 62, 63
MPI_ISEND, 54–56, 62, 63
MPI_Op_create, 170, 176
MPI_Pack, 124, 125
MPI_Pack_size, 125
MPI_PROBE, 66
MPI_Put, 339, 344, 346, 347
MPI_RECV, 35, 36, 42–44, 56, 66, 101
MPI_REDUCE, 162, 163, 166
MPI_Reduce, 165, 166, 170
MPI_REQUEST_FREE, 55
MPI_Request_free, 280
MPI_Scan, 176
MPI_Scatter, 149
MPI_Scatterv, 150
MPI_SEND, 35, 36, 43, 44, 56, 66, 101
MPI_Send, 114, 117, 118, 124
MPI_SENDRECV, 111–113
MPI_SENDRECV_REPLACE, 253
MPI_SSEND, 42, 56
MPI_Test_cancelled, 281
MPI_TYPE_COMMIT, 99, 111–113, 328
MPI_Type_commit, 114, 117, 118, 124, 141–145, 150, 176
MPI_TYPE_CONTIGUOUS, 79, 96, 101, 103
MPI_Type_contiguous, 141
MPI_TYPE_CREATE_DARRAY, 93
MPI_TYPE_CREATE_HVECTOR, 111, 112
MPI_Type_create_hvector, 114, 117
MPI_TYPE_CREATE_INDEXED_BLOCK, 328
MPI_TYPE_CREATE_STRUCT, 86, 96, 113
MPI_Type_create_struct, 114, 117, 118, 124, 143, 145, 176
MPI_TYPE_CREATE_SUBARRAY, 433
MPI_TYPE_EXTENT, 111–113, 328, 330, 332
MPI_Type_extent, 114
MPI_TYPE_FREE, 328
MPI_Type_get_contents, 119
MPI_Type_get_envelope, 119
MPI_TYPE_HVECTOR, 111, 112
MPI_Type_hvector, 114, 117
MPI_TYPE_INDEXED, 83, 112
MPI_Type_indexed, 114, 117
MPI_TYPE_STRUCT, 86, 96, 113
MPI_Type_struct, 114, 117, 118, 124, 143, 145, 176
MPI_TYPE_VECTOR, 80, 81, 111–113
MPI_Type_vector, 142, 144, 150
MPI_Unpack, 124, 125
MPI_WAIT, 54–56, 62, 63, 397
MPI_WAITANY, 62
MPI_WAITSOME, 63
MPI_Win_complete, 339, 347
MPI_WIN_CREATE, 328, 330, 332
MPI_WIN_FENCE, 328, 330, 332
MPI_Win_fence, 346
MPI_Win_lock, 344
MPI_Win_post, 347

Examples Index 573

MPI_Win_start, 339, 347
MPI_Win_unlock, 344
MPI_Win_wait, 347
mpiexec, 286

Non-deterministic program with MPI_Bcast, 178
Non-overtaking messages, 42
Nonblocking operations, 54, 55
Nonblocking operations - message ordering, 56
Nonblocking operations - progress, 56

Threads and MPI, 366
Typemap, 79–81, 83, 86, 93

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::*_NULL, 451
MPI::BOOL, 454, 495
MPI::BYTE, 453–455
MPI::CHAR, 454
MPI::CHARACTER, 454
MPI::COMM_NULL, 453, 456
MPI::COMPLEX, 454, 495
MPI::DOUBLE, 454
MPI::DOUBLE_COMPLEX, 454, 495
MPI::DOUBLE_INT, 455
MPI::DOUBLE_PRECISION, 454
MPI::ERRORS_ARE_FATAL, 19
MPI::ERRORS_RETURN, 19
MPI::ERRORS_THROW_EXCEPTIONS, 19, 23, 265
MPI::F_COMPLEX, 454
MPI::F_DOUBLE_COMPLEX, 455
MPI::FLOAT, 454
MPI::FLOAT_INT, 455
MPI::INT, 454
MPI::INTEGER, 453, 454
MPI::INTEGER1, 455
MPI::INTEGER2, 455
MPI::INTEGER4, 455
MPI::INTEGER8, 455
MPI::LOGICAL, 454
MPI::LOGICAL, MPI::BOOL, 455
MPI::LONG, 454
MPI::LONG_DOUBLE, 453, 454
MPI::LONG_DOUBLE_COMPLEX, 454, 455, 495
MPI::LONG_DOUBLE_INT, 455
MPI::LONG_INT, 455
MPI::LONG_LONG, 454
MPI::MAXLOC, 455
MPI::MINLOC, 455
MPI::PACKED, 453, 454
MPI::REAL, 454
MPI::REAL16, 455
MPI::REAL4, 455
MPI::REAL8, 455
MPI::SHORT, 454
MPI::SHORT_INT, 455
MPI::SIGNED_CHAR, 454
MPI::SIGNED_CHAR, MPI::UNSIGNED_CHAR, 453
MPI::TWODOUBLE_PRECISION, 455
MPI::TWOINT, 455
MPI::TWOINTEGER, 455
MPI::TWOREAL, 455
MPI::UNSIGNED, 454
MPI::UNSIGNED_CHAR, 454
MPI::UNSIGNED_LONG, 454
MPI::UNSIGNED_LONG_LONG, 454
MPI::UNSIGNED_SHORT, 454
MPI::WCHAR, 454
MPI_2DOUBLE_PRECISION, 165
MPI_2INT, 165
MPI_2INTEGER, 165
MPI_2REAL, 165
MPI_ADDRESS_KIND, 18, 106, 486, 487
MPI_ANY_SOURCE, 30, 42, 64–66, 71, 240, 261
MPI_ANY_TAG, 14, 30, 32, 64, 65, 67, 71, 75
MPI_APPNUM, 317, 318
MPI_ARGV_NULL, 298, 299, 462
MPI_ARGVS_NULL, 302, 462
MPI_BAND, 161, 162
MPI_BOR, 162

574

MPI Constant and Predefined Handle Index 575

MPI_BOTTOM, 10, 15, 17, 33, 94, 103, 104, 132, 300, 462, 465–467, 469, 484, 485, 489, 566
MPI_BSEND_OVERHEAD, 47, 262
MPI_BXOR, 162
MPI_BYTE, 27, 28, 34, 35, 37, 127, 162, 374, 412, 424, 453, 489, 562
MPI_CART, 248
MPI_CHAR, 28, 37, 86, 163
MPI_CHARACTER, 27, 36, 37, 163
MPI_COMBINER_CONTIGUOUS, 106, 109
MPI_COMBINER_DARRAY, 106, 111
MPI_COMBINER_DUP, 106, 109
MPI_COMBINER_F90_COMPLEX, 106, 111
MPI_COMBINER_F90_INTEGER, 106, 111
MPI_COMBINER_F90_REAL, 106, 111
MPI_COMBINER_HINDEXED, 106, 110
MPI_COMBINER_HINDEXED_INTEGER, 106, 110
MPI_COMBINER_HVECTOR, 106, 110
MPI_COMBINER_HVECTOR_INTEGER, 106, 110
MPI_COMBINER_INDEXED, 106, 110
MPI_COMBINER_INDEXED_BLOCK, 106, 110
MPI_COMBINER_NAMED, 105, 106, 109
MPI_COMBINER_RESIZED, 106, 111
MPI_COMBINER_STRUCT, 106, 110
MPI_COMBINER_STRUCT_INTEGER, 106, 110
MPI_COMBINER_SUBARRAY, 106, 110
MPI_COMBINER_VECTOR, 106, 109
MPI_COMM_NULL, 185, 196–199, 201, 237, 244, 246, 300, 319, 320, 563
MPI_COMM_PARENT, 237
MPI_COMM_SELF, 185, 221, 237, 283, 319, 375
MPI_COMM_WORLD, 14, 24, 29, 185–187, 193, 194, 212, 237, 244, 260, 261, 264, 267, 274, 281–283, 285, 293, 294, 296, 297, 301–303, 316–319, 369, 410, 429, 430, 479, 488, 564
MPI_COMPLEX, 27, 162, 414, 471
MPI_CONGRUENT, 194, 211
MPI_CONVERSION_FN_NULL, 419
MPI_DATATYPE, 19
MPI_DATATYPE_NULL, 100
MPI_DISPLACEMENT_CURRENT, 385, 565
MPI_DISTRIBUTE_BLOCK, 91
MPI_DISTRIBUTE_CYCLIC, 91
MPI_DISTRIBUTE_DFLT_DARG, 91
MPI_DISTRIBUTE_NONE, 91
MPI_DOUBLE, 28, 162, 470
MPI_DOUBLE_COMPLEX, 27, 414, 471
MPI_DOUBLE_INT, 165
MPI_DOUBLE_PRECISION, 27, 162, 471
MPI_ERR_ACCESS, 273, 378, 431
MPI_ERR_AMODE, 273, 377, 431
MPI_ERR_ARG, 272
MPI_ERR_ASSERT, 272, 349
MPI_ERR_BAD_FILE, 273, 431
MPI_ERR_BASE, 263, 272, 349
MPI_ERR_BUFFER, 272
MPI_ERR_COMM, 272
MPI_ERR_CONVERSION, 273, 419, 431
MPI_ERR_COUNT, 272
MPI_ERR_DIMS, 272
MPI_ERR_DISP, 272, 349
MPI_ERR_DUP_DATAREP, 273, 417, 431
MPI_ERR_FILE, 273, 431
MPI_ERR_FILE_EXISTS, 273, 431
MPI_ERR_FILE_IN_USE, 273, 378, 431
MPI_ERR_GROUP, 272
MPI_ERR_IN_STATUS, 31, 33, 52, 59, 61, 266, 272, 360, 390, 458
MPI_ERR_INFO, 272
MPI_ERR_INFO_KEY, 272, 288
MPI_ERR_INFO_NOKEY, 272, 289
MPI_ERR_INFO_VALUE, 272, 288
MPI_ERR_INTERN, 272
MPI_ERR_IO, 273, 431
MPI_ERR_KEYVAL, 233, 272
MPI_ERR_LASTCODE, 271, 273–275
MPI_ERR_LOCKTYPE, 272, 349
MPI_ERR_NAME, 272, 313
MPI_ERR_NO_MEM, 263, 272
MPI_ERR_NO_SPACE, 273, 431
MPI_ERR_NO_SUCH_FILE, 273, 378, 431
MPI_ERR_NOT_SAME, 273, 431
MPI_ERR_OP, 272
MPI_ERR_OTHER, 271, 272
MPI_ERR_PENDING, 59, 272

576 MPI Constant and Predefined Handle Index

MPI_ERR_PORT, 272, 310
MPI_ERR_QUOTA, 273, 431
MPI_ERR_RANK, 272
MPI_ERR_READ_ONLY, 273, 431
MPI_ERR_REQUEST, 272
MPI_ERR_RMA_CONFLICT, 272, 349
MPI_ERR_RMA_SYNC, 272, 349
MPI_ERR_ROOT, 272
MPI_ERR_SERVICE, 272, 312
MPI_ERR_SIZE, 272, 349
MPI_ERR_SPAWN, 272, 299, 300
MPI_ERR_TAG, 272
MPI_ERR_TOPOLOGY, 272
MPI_ERR_TRUNCATE, 272
MPI_ERR_TYPE, 272
MPI_ERR_UNKNOWN, 271, 272
MPI_ERR_UNSUPPORTED_DATAREP, 273, 431
MPI_ERR_UNSUPPORTED_OPERATION, 273, 431
MPI_ERR_WIN, 272, 349
MPI_ERRCODES_IGNORE, 17, 300, 462, 465
MPI_ERRHANDLER_NULL, 270
MPI_ERROR, 31, 52
MPI_ERROR_STRING, 271
MPI_ERRORS_ARE_FATAL, 264, 265, 276, 349, 429
MPI_ERRORS_RETURN, 264, 265, 277, 430, 488
MPI_F_STATUS_IGNORE, 483
MPI_F_STATUSES_IGNORE, 483
MPI_FILE, 474
MPI_FILE_NULL, 378, 430
MPI_FLOAT, 28, 86, 160, 162, 413
MPI_FLOAT_INT, 12, 165
MPI_GRAPH, 248
MPI_GROUP_EMPTY, 184, 189, 190, 197
MPI_GROUP_NULL, 184, 192
MPI_HOST, 260
MPI_IDENT, 187, 194
MPI_IN_PLACE, 132, 462, 465, 469
MPI_INFO_NULL, 291, 299, 309, 377, 378, 386
MPI_INT, 12, 28, 78, 162, 413, 414, 470, 488, 490
MPI_INTEGER, 27, 34, 162, 470, 471, 490
MPI_INTEGER1, 27
MPI_INTEGER2, 27, 414
MPI_INTEGER4, 27
MPI_INTEGER8, 474
MPI_INTEGER_KIND, 106, 486
MPI_IO, 260, 261
MPI_KEYVAL_INVALID, 225, 226
MPI_LAND, 161, 162
MPI_LASTUSEDCODE, 274
MPI_LB, 16, 89, 92, 96–98, 413
MPI_LOCK_EXCLUSIVE, 342
MPI_LOCK_SHARED, 342
MPI_LOGICAL, 27, 162
MPI_LONG, 28, 162
MPI_LONG_DOUBLE, 28, 162
MPI_LONG_DOUBLE_INT, 165
MPI_LONG_INT, 165
MPI_LONG_LONG, 28, 162, 562
MPI_LONG_LONG_INT, 28, 162, 562
MPI_LOR, 162
MPI_LXOR, 162
MPI_MAX, 160–162, 175
MPI_MAX_DATAREP_STRING, 387, 417
MPI_MAX_ERROR_STRING, 270, 275
MPI_MAX_INFO_KEY, 272, 287, 289, 290
MPI_MAX_INFO_VAL, 272, 287
MPI_MAX_OBJECT, 237, 563
MPI_MAX_OBJECT_NAME, 236
MPI_MAX_PORT_NAME, 308
MPI_MAX_PROCESSOR_NAME, 262, 564
MPI_MAXLOC, 162, 164, 165, 167
MPI_MIN, 161, 162
MPI_MINLOC, 162, 164, 165, 167
MPI_MODE_APPEND, 376, 377
MPI_MODE_CREATE, 376, 377, 384
MPI_MODE_DELETE_ON_CLOSE, 376–378
MPI_MODE_EXCL, 376, 377
MPI_MODE_NOCHECK, 345, 346
MPI_MODE_NOPRECEDE, 345
MPI_MODE_NOPUT, 345
MPI_MODE_NOSTORE, 345
MPI_MODE_NOSUCCEED, 345
MPI_MODE_RDONLY, 376, 377, 382
MPI_MODE_RDWR, 376, 377
MPI_MODE_SEQUENTIAL, 376, 377, 379, 380, 385, 390, 394, 403, 423, 565
MPI_MODE_UNIQUE_OPEN, 376, 377

MPI Constant and Predefined Handle Index 577

MPI_MODE_WRONLY, 376, 377
MPI_OFFSET_KIND, 18, 424
MPI_OP_NULL, 170
MPI_ORDER_C, 14, 88, 91, 92
MPI_ORDER_FORTRAN, 14, 88, 91
MPI_PACKED, 27, 28, 34, 121, 123, 127, 414, 453, 489
MPI_PROC_NULL, 26, 75, 134, 136, 138, 140, 148, 149, 161, 187, 253, 260, 261, 326, 563, 565
MPI_PROD, 161, 162
MPI_REAL, 27, 34, 162, 414, 470, 471, 477
MPI_REAL2, 27
MPI_REAL4, 27, 470, 474
MPI_REAL8, 27, 470
MPI_REPLACE, 332, 565
MPI_REQUEST_NULL, 52–55, 57–61, 360
MPI_ROOT, 134
MPI_SEEK_CUR, 398, 404
MPI_SEEK_END, 398, 404
MPI_SEEK_SET, 398, 399, 404
MPI_SHORT, 28, 162
MPI_SHORT_INT, 165
MPI_SIGNED_CHAR, 28, 162, 163, 562
MPI_SIMILAR, 187, 194, 211
MPI_SOURCE, 31
MPI_STATUS, 21, 33, 52
MPI_STATUS_IGNORE, 10, 17, 33, 359, 390, 462, 465, 469, 482, 483, 489, 561
MPI_STATUS_SIZE, 31
MPI_STATUSES_IGNORE, 14, 33, 359, 360, 462, 465, 482, 483, 561
MPI_SUBVERSION, 260
MPI_SUCCESS, 16, 18, 59, 61, 224–229, 231, 271, 272, 276, 277, 300, 419, 444, 445
MPI_SUM, 161, 162, 488
MPI_TAG, 31
MPI_TAG_UB, 28, 260, 486, 487
MPI_THREAD_FUNNELED, 368, 369
MPI_THREAD_MULTIPLE, 369, 371
MPI_THREAD_SERIALIZED, 369
MPI_THREAD_SINGLE, 368–370
MPI_TYPECLASS_COMPLEX, 476
MPI_TYPECLASS_INTEGER, 476
MPI_TYPECLASS_REAL, 476
MPI_UB, 12, 16, 89, 93, 96–98, 413
MPI_UNDEFINED, 32, 57, 58, 102, 187, 199, 248, 255, 256, 472, 562
MPI_UNEQUAL, 187, 194, 211
MPI_UNIVERSE_SIZE, 296, 316
MPI_UNSIGNED, 28, 162
MPI_UNSIGNED_CHAR, 28, 162, 163
MPI_UNSIGNED_LONG, 28, 162
MPI_UNSIGNED_LONG_LONG, 28, 162, 562
MPI_UNSIGNED_SHORT, 28, 162
MPI_VERSION, 260
MPI_WCHAR, 28, 163, 238, 414, 562
MPI_WIN_BASE, 324, 488
MPI_WIN_DISP_UNIT, 324
MPI_WIN_NULL, 324
MPI_WIN_SIZE, 324
MPI_WTIME_IS_GLOBAL, 260, 261, 278, 486

MPI Declarations Index

This index refers to declarations needed in C/C++, such as address kind integers, handles, etc. The underlined page numbers is the “main” reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI Declarations Index

This index refers to declarations needed in C/C++, such as address kind integers, handles, etc. The underlined page numbers is the “main” reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI::Aint, 15, 15, 19, 79, 79, 81, 84, 86, 94, 97, 98, 107, 127, 128, 322, 326, 328, 331, 413, 417, 441–444, 485, 485, 486
MPI::Cartcomm, 244, 450, 456
MPI::Comm, 26, 188, 193–196, 198, 201, 210, 211, 213, 223, 226, 227, 450, 456, 456
MPI::Datatype, 79, 450
MPI::Errhandler, 266, 267–270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::File, 269, 270, 276, 375, 377, 379–383, 385, 387, 391–404, 406–410, 413, 421, 422, 450, 480
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 358, 450
MPI::Group, 186, 186, 187–192, 196, 211, 324, 339, 340, 380, 450, 480
MPI::Info, 262, 287, 287, 288–291, 296, 299, 301, 308–312, 375, 378, 382, 383, 385, 450, 480
MPI::Intercomm, 450, 456
MPI::Intracomm, 450, 456
MPI::Offset, 15, 15, 19, 379, 380, 385, 387, 391–393, 398, 399, 403, 404, 406, 407, 417, 424
MPI::Op, 160, 168, 170, 172–175, 331, 450, 480
MPI::Prequest, 69, 450
MPI::Request, 49–51, 53, 53, 54, 57–61, 64, 67, 69–72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Status, 29, 32, 53, 57–61, 64, 65, 68, 73, 74, 102, 358, 364, 391, 392, 394–396, 400, 402, 403, 406–410, 450, 451, 482, 483
MPI::Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 450, 480
MPI_Aint, 15, 15, 18, 79, 79, 81, 84, 86, 94, 97, 98, 107, 127, 128, 322, 326, 328, 331, 413, 417, 441–444, 462, 485, 485, 486
MPI_Comm, 26, 188, 193–196, 198, 201, 210, 211, 213, 223, 226, 227
MPI_Datatype, 79, 466
MPI_Errhandler, 266, 267–270, 446, 447, 480
MPI_File, 269, 270, 276, 375, 377, 379–383, 385, 387, 391–404, 406–410, 413, 421, 422, 480
MPI_Fint, 479
MPI_Group, 186, 186, 187–192, 196, 211, 324, 339, 340, 380, 480
MPI_Info, 262, 287, 287, 288–291, 296, 299, 301, 308–312, 375, 378, 382, 383, 385, 480, 565
MPI_Offset, 15, 15, 18, 379, 380, 385, 387, 391–393, 398, 399, 403, 404, 406, 407, 417, 424, 424, 478
MPI_Op, 160, 168, 170, 172–175, 331, 480
MPI_Request, 49–51, 53, 53, 54, 57–61, 64, 67, 69–72, 358, 361, 393, 397, 398, 401, 480
MPI_Status, 29, 32, 53, 57–61, 64, 65, 68, 73, 74, 102, 358, 364, 391, 392, 394–396, 400, 402, 403, 406–410, 482, 483

578

MPI Declarations Index 579

MPI_Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 480

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

MPI_Comm_copy_attr_function, 17, 223
MPI_Comm_delete_attr_function, 17, 223
MPI_Comm_errhandler_fn, 17, 266
MPI_Copy_function, 17, 444
MPI_Datarep_conversion_function, 417
MPI_Datarep_extent_function, 417
MPI_Delete_function, 17, 445
MPI_File_errhandler_fn, 269
MPI_Grequest_cancel_function, 360
MPI_Grequest_free_function, 359
MPI_Grequest_query_function, 358
MPI_Handler_function, 17, 447
MPI_Type_copy_attr_function, 231
MPI_Type_delete_attr_function, 231
MPI_User_function, 168
MPI_Win_copy_attr_function, 228
MPI_Win_delete_attr_function, 228
MPI_Win_errhandler_fn, 268

580

MPI Function Index

The underlined page numbers refer to the function definitions.

MPI Function Index

The underlined page numbers refer to the function definitions.

MPI_ABORT, 169, 264, 279, 283, 318, 479, 564
MPI_ACCUMULATE, 321, 325, 326, 331, 332, 333, 352, 565
MPI_ADD_ERROR_CLASS, 274, 274
MPI_ADD_ERROR_CODE, 275
MPI_ADD_ERROR_STRING, 275, 275
MPI_ADDRESS, 17, 94, 443, 486
MPI_ALLGATHER, 129, 133, 152, 152, 153–155
MPI_ALLGATHERV, 129, 133, 153, 154
MPI_ALLOC_MEM, 18, 262, 263, 272, 323, 327, 343, 462
MPI_ALLREDUCE, 129, 132–134, 161, 168, 172, 172, 562
MPI_ALLTOALL, 129, 133, 134, 155, 155, 157
MPI_ALLTOALLV, 129, 132–134, 156, 157
MPI_ALLTOALLW, 129, 133, 134, 158, 158, 159
MPI_ATTR_DELETE, 17, 227, 233, 445, 446
MPI_ATTR_GET, 17, 227, 233, 260, 446, 486
MPI_ATTR_PUT, 17, 226, 233, 445, 486
MPI_BARRIER, 129, 133, 135, 135, 426
MPI_BCAST, 129, 133, 136, 136, 451
MPI_BSEND, 39, 47, 262, 280
MPI_BSEND_INIT, 69, 72
MPI_BUFFER_ATTACH, 21, 45, 53
MPI_BUFFER_DETACH, 45, 280
MPI_CANCEL, 42, 53, 64, 67, 67, 68, 357, 360, 361
MPI_CART_COORDS, 243, 251, 251, 564
MPI_CART_CREATE, 243, 244, 244, 245, 246, 249, 254, 255, 563
MPI_CART_GET, 243, 249, 250, 563
MPI_CART_MAP, 243, 255, 255
MPI_CART_RANK, 243, 250, 250, 564
MPI_CART_SHIFT, 243, 252, 253, 253, 564
MPI_CART_SUB, 243, 254, 254, 255, 564
MPI_CARTDIM_GET, 243, 249, 249, 563
MPI_CLOSE_PORT, 308, 309, 311
MPI_COMM_ACCEPT, 307–309, 309, 310, 316, 318
MPI_COMM_C2F, 479
MPI_COMM_CALL_ERRHANDLER, 276, 277
MPI_COMM_CLONE, 457
MPI_COMM_COMPARE, 194, 211
MPI_COMM_CONNECT, 272, 310, 310, 317, 318
MPI_COMM_CREATE, 192, 196, 196, 197, 199, 210, 243
MPI_COMM_CREATE_ERRHANDLER, 17, 265, 266, 267, 446, 502
MPI_COMM_CREATE_KEYVAL, 17, 222, 223, 233, 444, 486, 563
MPI_COMM_DELETE_ATTR, 17, 222, 225, 226, 227, 233, 446
MPI_COMM_DISCONNECT, 233, 301, 318, 319, 319
MPI_COMM_DUP, 188, 192, 195, 195, 197, 202, 212, 214, 222, 224, 227, 233, 240, 444
MPI_COMM_DUP_FN, 17, 224, 224
MPI_COMM_F2C, 479
MPI_COMM_FREE, 192, 196, 201, 202, 212, 214, 225, 227, 233, 283, 301, 318, 319, 445, 453
MPI_COMM_FREE_KEYVAL, 17, 222, 225, 233, 445

581

582 MPI Function Index

MPI_COMM_GET_ATTR, 17, 222, 226, 226, 233, 445, 488
MPI_COMM_GET_ERRHANDLER, 17, 265, 267, 447, 564
MPI_COMM_GET_NAME, 236, 236, 237, 563
MPI_COMM_GET_PARENT, 237, 297, 300, 300, 301
MPI_COMM_GROUP, 14, 186, 188, 188, 192–194, 211, 265, 564
MPI_COMM_JOIN, 319, 319, 320
MPI_COMM_KEYVAL_CREATE, 501
MPI_COMM_NULL_COPY_FN, 17, 224, 224
MPI_COMM_NULL_DELETE_FN, 17, 225, 225
MPI_COMM_RANK, 193, 193, 211
MPI_COMM_REMOTE_GROUP, 211
MPI_COMM_REMOTE_SIZE, 211, 212
MPI_COMM_SET_ATTR, 17, 222, 225, 226, 233, 445
MPI_COMM_SET_ERRHANDLER, 17, 265, 267, 447
MPI_COMM_SET_NAME, 235, 235, 236
MPI_COMM_SIZE, 21, 193, 193, 194, 211
MPI_COMM_SPAWN, 285, 294–296, 296, 297, 299–304, 316, 317
MPI_COMM_SPAWN_MULTIPLE, 285, 294, 295, 300, 301, 302, 303, 317
MPI_COMM_SPLIT, 196, 198, 199, 200, 240, 243, 244, 246, 254–256
MPI_COMM_TEST_INTER, 210, 210
MPI_DIMS_CREATE, 243–245, 245
MPI_DUP_FN, 17, 224, 444
MPI_ERRHANDLER_C2F, 480
MPI_ERRHANDLER_CREATE, 17, 266, 446
MPI_ERRHANDLER_F2C, 480
MPI_ERRHANDLER_FREE, 265, 270, 564
MPI_ERRHANDLER_GET, 17, 265, 267, 447, 564
MPI_ERRHANDLER_SET, 17, 267, 447
MPI_ERROR_CLASS, 271, 271, 273
MPI_ERROR_STRING, 270, 271, 273, 275
MPI_EXSCAN, 129, 133, 161, 168, 175, 175
MPI_FILE_C2F, 480
MPI_FILE_CALL_ERRHANDLER, 276, 277
MPI_FILE_CLOSE, 319, 375, 377, 378
MPI_FILE_CREATE_ERRHANDLER, 265, 269, 269, 502
MPI_FILE_DELETE, 377, 378, 378, 382, 384, 430
MPI_FILE_F2C, 480
MPI_FILE_GET_AMODE, 381, 381
MPI_FILE_GET_ATOMICITY, 422, 422
MPI_FILE_GET_BYTE_OFFSET, 394, 399, 399, 404
MPI_FILE_GET_ERRHANDLER, 265, 270, 430, 564
MPI_FILE_GET_FILE_EXTENT, 413
MPI_FILE_GET_GROUP, 380, 381
MPI_FILE_GET_INFO, 383, 383, 384, 565
MPI_FILE_GET_POSITION, 398, 399
MPI_FILE_GET_POSITION_SHARED, 403, 404, 404, 423
MPI_FILE_GET_SIZE, 380, 380, 425
MPI_FILE_GET_TYPE_EXTENT, 412, 413, 419
MPI_FILE_GET_VIEW, 387, 387
MPI_FILE_IREAD, 388, 397, 397, 404, 420
MPI_FILE_IREAD_AT, 388, 393, 393
MPI_FILE_IREAD_SHARED, 388, 401, 401
MPI_FILE_IWRITE, 388, 398, 398
MPI_FILE_IWRITE_AT, 388, 393, 394
MPI_FILE_IWRITE_SHARED, 388, 401, 402
MPI_FILE_OPEN, 273, 367, 375, 375, 377, 382–384, 386, 399, 424, 425, 430, 431
MPI_FILE_PREALLOCATE, 379, 379, 380, 421, 425
MPI_FILE_READ, 388, 394, 394, 395, 397, 424, 425
MPI_FILE_READ_ALL, 388, 395, 395, 405
MPI_FILE_READ_ALL_BEGIN, 388, 405, 406, 407, 420
MPI_FILE_READ_ALL_END, 388, 405, 406, 408, 420
MPI_FILE_READ_AT, 388, 391, 391, 392, 393
MPI_FILE_READ_AT_ALL, 388, 391, 392

MPI Function Index 583

MPI_FILE_READ_AT_ALL_BEGIN, 388, 406
MPI_FILE_READ_AT_ALL_END, 388, 406
MPI_FILE_READ_ORDERED, 388, 402, 403
MPI_FILE_READ_ORDERED_BEGIN, 388, 409
MPI_FILE_READ_ORDERED_END, 388, 409
MPI_FILE_READ_SHARED, 388, 400, 400, 401, 403
MPI_FILE_SEEK, 398, 398, 399
MPI_FILE_SEEK_SHARED, 403, 403, 404, 423
MPI_FILE_SET_ATOMICITY, 377, 421, 421, 422
MPI_FILE_SET_ERRHANDLER, 265, 269, 430
MPI_FILE_SET_INFO, 382, 382, 383, 384, 565
MPI_FILE_SET_SIZE, 379, 379, 380, 421, 423, 425
MPI_FILE_SET_VIEW, 89, 273, 376, 382–385, 385, 386, 387, 399, 404, 411, 417, 424, 431, 565
MPI_FILE_SYNC, 378, 388, 420–422, 422, 423, 427
MPI_FILE_WRITE, 388, 396, 396, 398, 424
MPI_FILE_WRITE_ALL, 388, 396, 396
MPI_FILE_WRITE_ALL_BEGIN, 388, 408
MPI_FILE_WRITE_ALL_END, 388, 408
MPI_FILE_WRITE_AT, 388, 392, 392, 393, 394
MPI_FILE_WRITE_AT_ALL, 388, 392, 393
MPI_FILE_WRITE_AT_ALL_BEGIN, 388, 407
MPI_FILE_WRITE_AT_ALL_END, 388, 407
MPI_FILE_WRITE_ORDERED, 388, 402, 403, 403
MPI_FILE_WRITE_ORDERED_BEGIN, 388, 410
MPI_FILE_WRITE_ORDERED_END, 388, 410
MPI_FILE_WRITE_SHARED, 388, 400, 401–403
MPI_FINALIZE, 14, 23, 260, 279, 279, 280–284, 318, 319, 366, 375, 479, 483, 564
MPI_FINALIZED, 20, 279, 281, 283, 284, 284, 479
MPI_FREE_MEM, 263, 263, 272
MPI_GATHER, 129, 132, 133, 137, 139, 140, 147, 148, 152, 153
MPI_GATHERV, 129, 133, 139, 139, 140, 141, 149, 154
MPI_GET, 321, 325, 326, 328, 333, 565
MPI_GET_ADDRESS, 17, 79, 94, 94, 95, 104, 443, 465–467, 484, 485
MPI_GET_COUNT, 32, 32, 33, 52, 102, 103, 364, 390, 562
MPI_GET_ELEMENTS, 52, 102, 102, 103, 364, 365, 390
MPI_GET_PROCESSOR_NAME, 261, 262, 564
MPI_GET_VERSION, 259, 260, 279, 281
MPI_GRAPH_CREATE, 243, 246, 246, 248, 249, 256, 563
MPI_GRAPH_GET, 243, 248, 249
MPI_GRAPH_MAP, 243, 256, 256
MPI_GRAPH_NEIGHBORS, 243, 251, 251
MPI_GRAPH_NEIGHBORS_COUNT, 243, 251, 251
MPI_GRAPHDIMS_GET, 243, 248, 248
MPI_GREQUEST_COMPLETE, 358–361, 361
MPI_GREQUEST_START, 358, 358, 502
MPI_GROUP_C2F, 480
MPI_GROUP_COMPARE, 187, 190
MPI_GROUP_DIFFERENCE, 189
MPI_GROUP_EXCL, 190, 190, 192
MPI_GROUP_F2C, 480
MPI_GROUP_FREE, 192, 192, 193, 194, 265, 564
MPI_GROUP_INCL, 190, 190, 191
MPI_GROUP_INTERSECTION, 189
MPI_GROUP_RANGE_EXCL, 191, 192
MPI_GROUP_RANGE_INCL, 191, 191
MPI_GROUP_RANK, 186, 194
MPI_GROUP_SIZE, 186, 193
MPI_GROUP_TRANSLATE_RANKS, 187, 187, 563
MPI_GROUP_UNION, 188
MPI_IBSEND, 50, 53, 72

584 *MPI Function Index*

MPI_INFO_C2F, 480
MPI_INFO_CREATE, 288, 288
MPI_INFO_DELETE, 272, 288, 289, 290
MPI_INFO_DUP, 290, 291
MPI_INFO_F2C, 480
MPI_INFO_FREE, 291, 383
MPI_INFO_GET, 287, 289, 565
MPI_INFO_GET_NKEYS, 287, 290, 290, 565
MPI_INFO_GET_NTHKEY, 287, 290, 565
MPI_INFO_GET_VALUELEN, 287, 289, 565
MPI_INFO_SET, 288, 288, 289, 290
MPI_INIT, 14, 23, 185, 217, 260, 278, 278, 279, 281, 282, 284, 297–300, 316, 317, 368–370, 437, 479, 483
MPI_INIT_THREAD, 185, 368, 369–371, 479
MPI_INITIALIZED, 279, 281, 282, 282, 284, 370, 479
MPI_INTERCOMM_CREATE, 195, 212, 213, 214
MPI_INTERCOMM_MERGE, 209, 212, 213, 214
MPI_IPROBE, 32, 64, 64, 65, 66, 367
MPI_IRECV, 51, 464, 466, 467
MPI_IRSEND, 51
MPI_IS_THREAD_MAIN, 368, 370
MPI_ISEND, 11, 49, 72, 280, 464
MPI_ISSEND, 50
MPI_KEYVAL_CREATE, 17, 223, 225, 444, 445, 504
MPI_KEYVAL_FREE, 17, 225, 233, 445
MPI_LB, 17
MPI_LOOKUP_NAME, 272, 307, 311, 312, 313
MPI_NULL_COPY_FN, 17, 224, 444
MPI_NULL_DELETE_FN, 17, 225, 445
MPI_OP_C2F, 480
MPI_OP_CREATE, 168, 168, 170, 501
MPI_OP_F2C, 480
MPI_OP_FREE, 170
MPI_OPEN_PORT, 307, 308, 308, 309–311, 313
MPI_PACK, 47, 121, 124, 127, 414, 418
MPI_PACK_EXTERNAL, 7, 127, 127, 474, 562
MPI_PACK_EXTERNAL_SIZE, 128
MPI_PACK_SIZE, 47, 123, 124
MPI_PCONTROL, 435–437, 437
MPI_PROBE, 30, 32, 33, 64, 65, 65, 66, 67, 367
MPI_PUBLISH_NAME, 307, 311, 311, 312, 313
MPI_PUT, 321, 325, 326, 326, 328, 332, 333, 340, 344, 346, 565
MPI_QUERY_THREAD, 370, 371
MPI_RECV, 26, 29, 31–33, 65, 67, 78, 101, 102, 122, 131, 138, 365, 426, 451, 466–468
MPI_RECV_INIT, 71, 71
MPI_REDUCE, 129, 133, 134, 160, 160, 161, 168–170, 172–175, 332
MPI_REDUCE_SCATTER, 129, 133, 134, 161, 168, 173, 173
MPI_REGISTER_DATAREP, 273, 415, 417–419, 431, 503
MPI_REQUEST_C2F, 480
MPI_REQUEST_F2C, 480
MPI_REQUEST_FREE, 21, 54, 55, 67, 72, 280, 359–361
MPI_REQUEST_GET_STATUS, 33, 64, 359
MPI_RSEND, 40
MPI_RSEND_INIT, 70
MPI_SCAN, 129, 133, 161, 168, 174, 174, 175, 176
MPI_SCATTER, 129, 133, 147, 147, 149
MPI_SCATTERV, 129, 133, 148, 149, 150, 174
MPI_SEND, 25, 26, 27, 33, 35, 78, 101, 121, 256, 376, 426, 451, 463, 466
MPI_SEND_INIT, 69, 72
MPI_SENDRECV, 73, 252
MPI_SENDRECV_REPLACE, 74
MPI_SIZEOF, 469, 475, 476
MPI_SSEND, 40
MPI_SSEND_INIT, 70
MPI_START, 71, 71, 72
MPI_STARTALL, 72, 72
MPI_STATUS_C2F, 483
MPI_STATUS_F2C, 482
MPI_STATUS_SET_CANCELLED, 364
MPI_STATUS_SET_ELEMENTS, 364, 364
MPI_TEST, 11, 33, 52, 53, 54, 55, 57, 58, 67, 72, 280, 360, 361, 389, 390

MPI Function Index 585

MPI_TEST_CANCELLED, 52–54, 68, 68, 359, 365, 390
MPI_TESTALL, 57, 60, 60, 359, 360, 363, 366
MPI_TESTANY, 52, 57, 58, 58, 62, 359, 360, 363, 366
MPI_TESTSOME, 57, 61, 62, 359, 360, 363, 366
MPI_TOPO_TEST, 243, 248, 248
MPI_TYPE_C2F, 480
MPI_TYPE_COMMIT, 99, 99, 480
MPI_TYPE_CONTIGUOUS, 12, 79, 79, 81, 96, 106, 374, 413
MPI_TYPE_CREATE_DARRAY, 12, 32, 90, 90, 106
MPI_TYPE_CREATE_F90_COMPLEX, 12, 106, 108, 415, 469, 472, 473, 474
MPI_TYPE_CREATE_F90_INTEGER, 12, 106, 108, 415, 469, 472, 473, 474
MPI_TYPE_CREATE_F90_REAL, 12, 106, 108, 415, 469, 471, 472–474
MPI_TYPE_CREATE_HINDEXED, 12, 17, 79, 84, 84, 86, 87, 106, 441
MPI_TYPE_CREATE_HVECTOR, 12, 17, 79, 81, 81, 106, 441
MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106
MPI_TYPE_CREATE_KEYVAL, 222, 230, 233, 486, 563
MPI_TYPE_CREATE_RESIZED, 16, 17, 97, 106, 413
MPI_TYPE_CREATE_STRUCT, 12, 17, 79, 86, 87, 106, 158, 442
MPI_TYPE_CREATE_SUBARRAY, 12, 14, 87, 89, 91, 106
MPI_TYPE_DELETE_ATTR, 222, 233, 233
MPI_TYPE_DUP, 12, 100, 100, 106
MPI_TYPE_DUP_FN, 231, 231
MPI_TYPE_EXTENT, 17, 97, 443, 486
MPI_TYPE_F2C, 480
MPI_TYPE_FREE, 100, 108, 231
MPI_TYPE_FREE_KEYVAL, 222, 232, 233
MPI_TYPE_GET_ATTR, 222, 232, 233
MPI_TYPE_GET_CONTENTS, 105, 107, 108, 109
MPI_TYPE_GET_ENVELOPE, 105, 105, 107, 108, 473
MPI_TYPE_GET_EXTENT, 17, 97, 99, 443, 476, 483, 486
MPI_TYPE_GET_NAME, 238
MPI_TYPE_GET_TRUE_EXTENT, 98
MPI_TYPE_HINDEXED, 17, 84, 106, 442, 486
MPI_TYPE_HVECTOR, 17, 82, 106, 441, 486
MPI_TYPE_INDEXED, 12, 82, 83, 83, 84, 85, 106
MPI_TYPE_KEYVAL_CREATE, 502
MPI_TYPE_LB, 17, 97, 443
MPI_TYPE_MATCH_SIZE, 469, 475, 476
MPI_TYPE_NULL_COPY_FN, 231, 231
MPI_TYPE_NULL_DELETE_FN, 231
MPI_TYPE_SET_ATTR, 222, 232, 233
MPI_TYPE_SET_NAME, 237
MPI_TYPE_SIZE, 95, 95
MPI_TYPE_STRUCT, 16, 17, 86, 96, 106, 442, 486
MPI_TYPE_UB, 17, 97, 444, 486
MPI_TYPE_UB, MPI_TYPE_LB, 97
MPI_TYPE_VECTOR, 12, 80, 80, 81, 84, 106
MPI_UB, 17
MPI_UNPACK, 122, 122, 123, 127, 418
MPI_UNPACK_EXTERNAL, 7, 128, 474
MPI_UNPUBLISH_NAME, 272, 312, 312
MPI_WAIT, 31, 33, 52, 53, 53, 54–57, 59, 67, 72, 280, 357, 360, 361, 367, 389, 390, 404, 420, 422, 467, 468
MPI_WAITALL, 57, 59, 59, 60, 359, 360, 363, 366, 458
MPI_WAITANY, 42, 52, 57, 57, 58, 62, 359, 360, 363, 366
MPI_WAITSOME, 57, 60, 61–63, 359, 360, 363, 366
MPI_WIN_C2F, 480
MPI_WIN_CALL_ERRHANDLER, 276, 277
MPI_WIN_COMPLETE, 324, 334, 339, 339, 340, 342, 349, 350
MPI_WIN_CREATE, 322, 323, 348, 367
MPI_WIN_CREATE_ERRHANDLER, 265, 267, 268, 502

586 MPI Function Index

MPI_WIN_CREATE_KEYVAL, 222, 228, 233, 486, 563
MPI_WIN_DELETE_ATTR, 222, 230, 233
MPI_WIN_DUP_FN, 228, 228
MPI_WIN_F2C, 480
MPI_WIN_FENCE, 324, 334, 338, 338, 344, 349–352
MPI_WIN_FREE, 229, 319, 323, 324
MPI_WIN_FREE_KEYVAL, 222, 229, 233
MPI_WIN_GET_ATTR, 222, 230, 233, 324, 488
MPI_WIN_GET_ERRHANDLER, 265, 268, 564
MPI_WIN_GET_GROUP, 324, 325
MPI_WIN_GET_NAME, 238
MPI_WIN_KEYVAL_CREATE, 501
MPI_WIN_LOCK, 262, 323, 334, 342, 343, 344, 350
MPI_WIN_NULL_COPY_FN, 228, 228
MPI_WIN_NULL_DELETE_FN, 228
MPI_WIN_POST, 324, 334, 339, 340, 340, 341–345, 352
MPI_WIN_POST, MPI_WIN_FENCE, 350
MPI_WIN_SET_ATTR, 222, 229, 233
MPI_WIN_SET_ERRHANDLER, 265, 268
MPI_WIN_SET_NAME, 238
MPI_WIN_START, 334, 339, 339, 340, 342, 344, 345, 348
MPI_WIN_TEST, 341, 341
MPI_WIN_UNLOCK, 262, 334, 343, 344, 349, 350, 352
MPI_WIN_WAIT, 324, 334, 340, 340, 341–343, 350, 352
MPI_WTICK, 21, 278, 278
MPI_WTIME, 21, 261, 277, 277, 278
mpiexec, 279, 283, 285, 285, 369
mpirun, 284, 285

PMPI_, 435
PMPI_WTICK, 21
PMPI_WTIME, 21