MPI-2: Extensions to the Message-Passing Interface

Message Passing Interface Forum

November 15, 2003

This work was supported in part by NSF and DARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

Abstract

This document describes the MPI-1.2 and MPI-2 standards. They are both extensions to the MPI-1.1 standard. The MPI-1.2 part of the document contains clarifications and corrections to the MPI-1.1 standard and defines MPI-1.2. The MPI-2 part of the document describes additions to the MPI-1 standard and defines MPI-2. These include miscellaneous topics, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings.

Contents

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-2), many people helped with this effort. Those who served as the primary coordinators 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	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
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.

Chapter 1

Introduction to MPI-2

1.1 Background

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [5]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 (see http://www.mpi-forum.org for official MPI document releases). Since that time, effort has been focused in five types of 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++. This document 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. 0-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) have been collected in Chapter 3 of this document, "Version 1.2 of MPI." This 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, and constitute the specification for MPI-2. This document specifies Version 2.0 of MPI. 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-1.2. 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. Some implementations may require changes to be MPI-1 compliant.

- MPI-2 compliance will mean compliance with all of MPI-2.
- 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-1.2 program and a valid MPI-2 program, and a valid MPI-1.2 program is a valid MPI-2 program.

1.2 Organization of this Document

This document is organized as follows:

- Chapter 2, MPI-2 Terms and Conventions, explains notational terms and conventions used throughout the MPI-2 document.
- Chapter 3, Version 1.2 of MPI, contains the specification of MPI-1.2, which has one new function and consists primarily of clarifications to MPI-1.1. It is expected that some implementations will need modification in order to become MPI-1 compliant, as the result of these clarifications.

The rest of this document contains the MPI-2 Standard Specification. It adds substantial new types of functionality to MPI, in most cases specifying functions for an extended computational model (e.g., dynamic process creation and one-sided communication) or for a significant new capability (e.g., parallel I/O).

The following is a list of the chapters in MPI-2, along with a brief description of each.

- Chapter 4, Miscellany, discusses items that don't fit elsewhere, in particular language interoperability.
- Chapter 5, Process Creation and Management, discusses the extension of MPI to remove the static process model in MPI. It defines routines that allow for creation of processes.
- Chapter 6, 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 7, Extended Collective Operations, extends the semantics of MPI-1 collective operations to include intercommunicators. It also adds more convenient methods of constructing intercommunicators and two new collective operations.
- Chapter 8, 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 9, I/O, defines MPI-2 support for parallel I/O.
- Chapter 10, Language Bindings, describes the C++ binding and discusses Fortran-90 issues.

The Appendices are:

- Annex A, Language Bindings, gives bindings for MPI-2 functions, and lists constants, error codes, etc.
- Annex B, MPI-1 C++ Language Binding, gives C++ bindings for MPI-1.

The MPI Function Index is a simple index showing the location of the precise definition of each MPI-2 function, together with C, C++, and Fortran bindings.

MPI-2 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 ??, 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 ??, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
- Chapter ??, Communicator ID, describes an approach to providing identifiers for communicators.
- Chapter ??, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
- Chapter ??, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
- Chapter ??, Split Collective Communication, describes a specification for certain nonblocking collective operations.
- Chapter ??, Real-Time MPI, discusses MPI support for real time processing.

Chapter 2

MPI-2 Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI-2 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

MPI-1 used informal naming conventions. In many cases, MPI-1 names for C functions are of the form Class_action_subset and in Fortran of the form CLASS_ACTION_SUBSET, but this rule is not uniformly applied. In MPI-2, an attempt has been made to standardize names of new functions according to the following rules. In addition, the C++ bindings for MPI-1 functions also follow these rules (see Section 2.6.4). C and Fortran function names for MPI-1 have not been changed.

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 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 MPI-1 functions 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:

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

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 OUT. It is marked this way even though the handle itself is not modified — we use the OUT attribute to denote that what the handle *references* is updated. Thus, in C++, IN arguments are 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 ANSI 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**. 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_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.*)

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, ANSI C, and C++, in particular. (Note that ANSI C has been replaced by ISO C. References in MPI to ANSI C now mean 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, but that users are recommended not to continue using, since MPI-2 provides better solutions. 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 `INTEGER`s and 64 bit addresses. In MPI-2, these functions have new names, and new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and 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, while the MPI-2 function MPI_TYPE_CREATE_RESIZED provides a more convenient mechanism to achieve the same effect.

The following is 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 principle 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.

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

2.6.2 Fortran Binding Issues

MPI-1.1 provided bindings for Fortran 77. MPI-2 retains these bindings 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 7 of the MPI-1 document and Annex A in the MPI-2 document.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 4.12.9.

Handles are represented in Fortran as `INTEGER`s. 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 user codes that are discussed in detail in Section 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 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-2 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 can't 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 ANSI 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 ANSI 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 (see Section B.13.5). 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-2 functions generally follow the naming rules given. In some circumstances, the new MPI-2 function is related to an MPI-1 function with a name that does not follow the naming conventions. In this circumstance, the language neutral name is in analogy to the MPI-1 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 for MPI-1 do reflect the naming rules and can differ from the C and Fortran names. Thus, the analogous name in C++ to the MPI-1 name is 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 B 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.

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in a 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 8.7.

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 Chapter 7 of the MPI-1 document and in Section 4.13 of this document. 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.

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-2 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 ANSI 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 ANSI 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

Version 1.2 of MPI

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 which version of the MPI Standard the implementation being used conforms to. There are small differences between MPI-1 and MPI-1.1. There are very few differences (only those discussed in this chapter) between MPI-1.1 and MPI-1.2, but large differences (the rest of this document) between MPI-1.2 and MPI-2.

3.1 Version Number

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    1
#define MPI_SUBVERSION 2
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 1)
PARAMETER (MPI_SUBVERSION = 2)
```

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
```

MPI_GET_VERSION is one of the few functions that can be called before MPI_INIT and after MPI_FINALIZE. Its C++ binding can be found in the Annex, Section B.11.

3.2 MPI-1.0 and MPI-1.1 Clarifications

As experience has been gained since the releases of the 1.0 and 1.1 versions of the MPI Standard, it has become apparent that some specifications were insufficiently clear. In this section we attempt to make clear the intentions of the MPI Forum with regard to the behavior of several MPI-1 functions. An MPI-1-compliant implementation should behave in accordance with the clarifications in this section.

3.2.1 Clarification of MPI_INITIALIZED

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.

3.2.2 Clarification of 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 3.1 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 3.2 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 3.3 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 3.4 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 the MPI-2 function 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 on MPI_COMM_WORLD.

> *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 3.5 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);
```

3.2.3 Clarification of status after MPI_WAIT and MPI_TEST

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, MPI_WAITANY) that 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).

3.2.4 Clarification of MPI_INTERCOMM_CREATE

The Problem: The MPI-1.1 standard says, in the discussion of MPI_INTERCOMM_CREATE, both that

> The groups must be disjoint

and that

> The leaders may be the same process.

To further muddy the waters, the reason given for "The groups must be disjoint" is based on concerns about the implementation of MPI_INTERCOMM_CREATE that are not applicable for the case where the leaders are the same process.

The Fix: Delete the text:

> (the two leaders could be the same process)

from the discussion of MPI_INTERCOMM_CREATE.

Replace the text:

> All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

with

> 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.*)

3.2.5 Clarification of MPI_INTERCOMM_MERGE

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.

3.2.6 Clarification of Binding of MPI_TYPE_SIZE

This clarification is needed in the MPI-1 description of MPI_TYPE_SIZE, since the issue repeatedly arises. It is a clarification of the binding.

> *Advice to users.* The MPI-1 Standard specifies that the output argument of MPI_TYPE_SIZE in C is of type `int`. The MPI Forum considered proposals to change this and decided to reiterate the original decision. (*End of advice to users.*)

3.2.7 Clarification of MPI_REDUCE

The current text on p. 115, lines 25–28, from MPI-1.1 (June 12, 1995) says:

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 4.9.2 and Section 4.9.3. User-defined operators may operate on general, derived datatypes.

This text is changed to:

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 4.9.2 and Section 4.9.3. 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.

> *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.

3.2.8 Clarification of Error Behavior of Attribute Callback Functions

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.

3.2.9 Clarification of MPI_PROBE and MPI_IPROBE

Page 52, lines 1 thru 3 (of MPI-1.1, the June 12, 1995 version without changebars) become:

“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.”

Rationale.

The following program shows that the MPI-1 definitions of cancel and probe are in conflict:

```
Process 0                          Process 1
----------                         ----------
MPI_Init();                        MPI_Init();
MPI_Isend(dest=1);
                                   MPI_Probe();
MPI_Barrier();                     MPI_Barrier();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Barrier();                     MPI_Barrier();
                                   MPI_Recv();
```

Since the send has been cancelled by process 0, the wait must be local (page 54, line 13) and must return before the matching receive. For the wait to be local, the send must be successfully cancelled, and therefore must not match the receive in process 1 (page 54 line 29).

However, it is clear that the probe on process 1 must eventually detect an incoming message. Page 52 line 1 makes it clear that the subsequent receive by process 1 must return the probed message.

The above are clearly contradictory, and therefore the text "…and the send is not successfully cancelled before the receive" must be added to line 3 of page 54.

An alternative solution (rejected) would be to change the semantics of cancel so that the call is not local if the message has been probed. This adds complexity to implementations, and adds a new concept of "state" to a message (probed or not). It would, however, preserve the feature that a blocking receive after a probe is local.

(*End of rationale.*)

3.2.10 Minor Corrections

The following corrections to MPI-1.1 are (all page and line numbers are for the June 12, 1995 version without changebars):

- Page 11, line 36 reads
  MPI_ADDRESS
  but should read
  MPI_ADDRESS_TYPE

- Page 19, lines 1–2 reads
  for (64 bit) C integers declared to be of type longlong int
  but should read
  for C integers declared to be of type long long

- Page 40, line 48 should have the following text added:

  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 2 of the MPI-2 Standard, . (*End of advice to users.*)

- Page 41, lines 16–18 reads
  A **empty** status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, and is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0.
  but should read
  A **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.

- Page 52, lines 46–48 read

```
100           CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status, ierr)
        ELSE
200           CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status, ierr)
```

  but should read

```
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)
```

- Page 53, lines 18–23 read

```
100               CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                0, status, ierr)
            ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                0, status, ierr)
```

  but should read

```
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)
```

- Page 59, line 3 should have the following text added:

  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 2 of the MPI-2 Standard, . (*End of advice to users.*)

- Page 59, lines 42–45 read

```
int  MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
              MPI_Comm comm, MPI_Status *status)
```

  but should read

```
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)
```

- Page 60, line 3 reads

```
SOURCE, RECV TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

  but should read

```
SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

- Page 70, line 16 should have the following text added:

  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 2 of the MPI-2 Standard, . (*End of advice to users.*)

- Page 71, line 10 reads
  and do not affect the the content of a message
  but should read
  and do not affect the content of a message

- Page 74, lines 39–45 read
  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.)

  A datatype may specify overlapping entries. If such a datatype is used in a receive operation, that is, if some part of the receive buffer is written more than once by the receive operation, then the call is erroneous.

The first part was an MPI-1.1 addition. The second part overlaps with it. The old text will be removed so it now reads
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.)

- Page 75, line 24 should have the following text added:
The datatype argument should match the argument provided by the receive call that set the status variable.

- Page 85, line 36 reads
"specified by outbuf and outcount"
but should read
"specified by outbuf and outsize."

- Page 90, line 3 reads
MPI_Pack_size(count, MPI_CHAR, &k2);
but should read
MPI_Pack_size(count, MPI_CHAR, comm, &k2);

- Page 90, line 10 reads
MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);
but should read
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);

- Page 97, line 41 reads

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{disp}[\texttt{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\texttt{i}], \texttt{recvtype}, \texttt{i}, ...).$$

but should read

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs}[\texttt{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\texttt{i}], \texttt{recvtype}, \texttt{i}, ...).$$

- Page 109, lines 26–27 and page 110, lines 28–29 reads
The `jth` block of data sent from each process is received by every process and placed in the `jth` block of the buffer recvbuf.
but should read
The block of data sent from the `jth` process is received by every process and placed in the `jth` block of the buffer recvbuf.

- Page 117, lines 22–23 reads
MPI provides seven such predefined datatypes.
but should read
MPI provides nine such predefined datatypes.

- Page 121, line 1 reads

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
```

but should read

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
```

- Page 122, lines 35–36 read
MPI_OP_FREE(op)

IN	op	operation (handle)

but should read
MPI_OP_FREE(op)

INOUT	op	operation (handle)

- Page 125, line 1 reads
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
but should read
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

- Page 141, lines 27–27 read

IN	ranges	an array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup

but should read

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

- Page 142, line 10 reads

IN	n	number of elements in array ranks (integer)

but should read

IN	n	number of triplets in array ranges (integer)

- Page 194, lines 30–31 reads
to the greatest possible, extent,
but should read
to the greatest possible extent,

- Page 194, line 48 reads
MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
but should read
MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)

- Page 195, line 15 should have the following text added:
In the Fortran language, the user routine should be of the form:

```
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE, .....)
INTEGER COMM, ERROR_CODE
```

> *Advice to users.* Users are discouraged from using a Fortran HANDLER_FUNCTION 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 HANDLER_FUNCTION. (*End of advice to users.*)

- Page 196, lines 1–2 reads
MPI_ERRHANDLER_FREE(errhandler)

IN	errhandler	MPI error handler (handle)

but should read
MPI_ERRHANDLER_FREE(errhandler)

INOUT errhandler MPI error handler (handle)

- Page 197, line 25 should have added:

  An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

- Page 201, line 28 reads
  ...of different language bindings is is done ....
  but should read
  ...of different language bindings is done ....

- Page 203, line 1 reads
  `MPI_PCONTROL(level)`

  but should read
  `MPI_PCONTROL(LEVEL)`

- Page 210, line 44 reads
  MPI_PENDING
  but should read
  MPI_ERR_PENDING

- Page 211, line 44 reads
  MPI_DOUBLE_COMPLEX
  but should be moved to Page 212, line 22 since it is an optional Fortran datatype.

- Page 212, add new lines of text at line 22 and line 25 to read:
  etc.
  Thus, the text will now read:

```
/* optional datatypes (Fortran) */
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_REAL2
MPI_REAL4
MPI_REAL8
etc.

/* optional datatypes (C) */
MPI_LONG_LONG_INT
etc.
```

- Page 213, line 28. The following text should be added:

```
/* Predefined functions in C and Fortran */
MPI_NULL_COPY_FN
MPI_NULL_DELETE_FN
MPI_DUP_FN
```

- Page 213, line 41. Add the line

```
MPI_Errhandler
```

- Page 214, line 9 reads

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
```

but should read

```
SUBROUTINE USER_FUNCTION( INVEC, INOUTVEC, LEN, TYPE)
```

- Page 214, lines 14 and 15 read

```
 PROCEDURE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
```

but should read

```
 SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
```

- Page 214, line 21 reads

```
PROCEDURE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
```

but should read

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
```

- Page 214, line 23 should have the following text added:
The handler-function for error handlers should be declared like this:

```
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE, .....)
INTEGER COMM, ERROR_CODE
```

- Page 216, lines 4–7 read

```
int  MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
              MPI_Comm comm, MPI_Status *status)
```

but should read

```
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)
```

- Page 220, lines 19–20 reads
  int double MPI_Wtime(void)
  int double MPI_Wtick(void)
  but should read
  double MPI_Wtime(void)
  double MPI_Wtick(void)

- Page 222, line 34 reads
  INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
  but should read
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

- Page 222, line 38 reads
  INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
  but should read
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

- Page 227, lines 19–20 reads
  MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
  INTEGER INTERCOMM, INTRACOMM, IERROR
  but should read
  MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
  INTEGER INTERCOMM, NEWINTRACOMM, IERROR

- Page 228, line 46 reads
  MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
  but should read
  MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)

- Page 229, line 33 reads
  MPI_PCONTROL(level)

but should read
`MPI_PCONTROL(LEVEL)`

Chapter 4

Miscellany

This chapter contains topics that do not fit conveniently into other chapters.

4.1 Portable MPI Process Startup

A number of implementations of MPI-1 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.

This is advice to implementors, rather than a required part of MPI-2. It is not suggested that this be the only way to start MPI programs. If an implementation does provide a command called `mpiexec`, however, it must be of the form described here.

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 5.3.4).

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 5.3.4 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 4.1 Start 16 instances of `myprog` on the current or default machine:

```
mpiexec -n 16 myprog
```

Example 4.2 Start 10 processes on the machine called `ferrari`:

```
mpiexec -n 10 -host ferrari myprog
```

Example 4.3 Start three copies of the same program with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 4.4 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 4.5 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.*)

4.2 Passing NULL to MPI_Init

In MPI-1.1, it is explicitly stated that an implementation is allowed to require that the arguments argc and argv passed by an application to MPI_INIT in C be the same arguments passed into the application as the arguments to main. In MPI-2 implementations are not allowed to impose this requirement. Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main. 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.*)

4.3 Version Number

The values for the MPI_VERSION and MPI_SUBVERSION for an MPI-2 implementation are 2 and 0 respectively. This applies both to the values of the above constants and to the values returned by MPI_GET_VERSION.

4.4 Datatype Constructor MPI_TYPE_CREATE_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 (integer)
IN	blocklength	size of block (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
```

4.5 Treatment of MPI_Status

The following features add to, but do not change, the functionality associated with MPI_STATUS.

4.5.1 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, particularly in MPI-2, 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 ANY and ALL 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 _ANY and _ALL 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 4.6 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
```

4.5.2 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.

4.6 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.

4.7 Committing a Committed Datatype

In MPI-1.2, the effect of calling MPI_TYPE_COMMIT with a datatype that is already committed is not specified. For MPI-2, it is specified that MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

4.8 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-2 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.*)

4.9 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-1 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.*)

4.10 The Info Object

Many of the routines in MPI-2 take an argument info. info is an opaque object with a handle of type MPI_Info in C, MPI::Info in C++, and INTEGER in Fortran. It consists of (key,value) pairs (both key and value are strings). A key may have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

If a function does not recognize a key, it will ignore it, unless otherwise specified. If an implementation recognizes a key but does not recognize the format of the corresponding value, the result is undefined.

Keys have an implementation-defined maximum length of MPI_MAX_INFO_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI_MAX_INFO_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI_MAX_INFO_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI_MAX_INFO_VAL might be very large, so it might not be wise to declare a string of that size. (*End of advice to users.*)

When it is an argument to a non-blocking routine, info is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, they mean the string representation of these types. An implementation may define its own rules for how info value strings are converted to other types, but to ensure portability, every implementation must support the following representations. Legal values for a boolean must include the strings “true” and “false” (all lowercase). For integers, legal values must include string representations of decimal values of integers that are within the range of a standard integer type in the program. (However it is possible that not every legal integer is a legal value for a given key.) On positive numbers, + signs are optional. No space may appear between a + or − sign and the leading digit of a number. For comma separated lists, the string must contain legal elements separated by commas. Leading and trailing spaces are stripped automatically from the types of info values described above and for each element of a comma separated list. These rules apply to all info values of these types. Implementations are free to specify a different interpretation for values of other info keys.

MPI_INFO_CREATE(info)

OUT	info	info object created (handle)

```
int MPI_Info_create(MPI_Info *info)
```

```
MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR
```

```
static MPI::Info MPI::Info::Create()
```

MPI_INFO_CREATE creates a new info object. The newly created object contains no key/value pairs.

MPI_INFO_SET(info, key, value)

INOUT	info	info object (handle)
IN	key	key (string)
IN	value	value (string)

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

```
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY, VALUE
```

```
void MPI::Info::Set(const char* key, const char* value)
```

MPI_INFO_SET adds the (key,value) pair to info, and overrides the value if a value for the same key was previously set. key and value are null-terminated strings in C. In Fortran, leading and trailing spaces in key and value are stripped. If either key or value are larger than the allowed maximums, the errors MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are raised, respectively.

MPI_INFO_DELETE(info, key)

INOUT	info	info object (handle)
IN	key	key (string)

```
int MPI_Info_delete(MPI_Info info, char *key)
```

```
MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY
```

```
void MPI::Info::Delete(const char* key)
```

MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info, the call raises an error of class MPI_ERR_INFO_NOKEY.

MPI_INFO_GET(info, key, valuelen, value, flag)

IN	info	info object (handle)
IN	key	key (string)
IN	valuelen	length of value arg (integer)
OUT	value	value (string)
OUT	flag	true if key defined, false if not (boolean)

```
int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,
             int *flag)
```

```
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG
```

```
bool MPI::Info::Get(const char* key, int valuelen, char* value) const
```

This function retrieves the value associated with key in a previous call to MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters available in value. If it is less than the actual size of the value, the value is truncated. In C, valuelen should be one less than the amount of allocated space to allow for the null terminator.

If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.

MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)

IN	info	info object (handle)
IN	key	key (string)
OUT	valuelen	length of value arg (integer)
OUT	flag	true if key defined, false if not (boolean)

```
int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,
             int *flag)
```

```
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    LOGICAL FLAG
    CHARACTER*(*) KEY
```

```
bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const
```

Retrieves the length of the value associated with key. If key is defined, valuelen is set to the length of its associated value and flag is set to true. If key is not defined, valuelen is not touched and flag is set to false. The length returned in C or C++ does not include the end-of-string character.

If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.

MPI_INFO_GET_NKEYS(info, nkeys)

IN	info	info object (handle)
OUT	nkeys	number of defined keys (integer)

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

```
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
    INTEGER INFO, NKEYS, IERROR
```

```
int MPI::Info::Get_nkeys() const
```

MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.

MPI_INFO_GET_NTHKEY(info, n, key)

IN	info	info object (handle)
IN	n	key number (integer)
OUT	key	key (string)

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
```

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
    INTEGER INFO, N, IERROR
    CHARACTER*(*) KEY
```

```
void MPI::Info::Get_nthkey(int n, char* key) const
```

This function returns the nth defined key in info. Keys are numbered $0 \ldots N-1$ where N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and $N-1$ are guaranteed to be defined. The number of a given key does not change as long as info is not modified with MPI_INFO_SET or MPI_INFO_DELETE.

MPI_INFO_DUP(info, newinfo)

IN	info	info object (handle)
OUT	newinfo	info object (handle)

```
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
```

```
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR
```

```
MPI::Info MPI::Info::Dup() const
```

MPI_INFO_DUP duplicates an existing info object, creating a new object, with the same (key,value) pairs and the same ordering of keys.

MPI_INFO_FREE(info)

INOUT	info	info object (handle)

```
int MPI_Info_free(MPI_Info *info)
```

```
MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR
```

```
void MPI::Info::Free()
```

This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is interpreted each time the info is passed to a routine. Changes to an info after return from a routine do not affect that interpretation.

4.11 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 6.4.3.)

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 4.7 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 4.8 Same example, in C

```
float  (* f)[100][100] ;
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

4.12 Language Interoperability

4.12.1 Introduction

It is not uncommon for library developers to use one language to develop an applications library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C, C++, and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

Initialization We need to specify how the MPI environment is initialized for all languages.

Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.

Interlanguage communication We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extendable to new languages, should MPI bindings be defined for such languages.

4.12.2 Assumptions

We assume that conventions exist for programs written in one language to call functions in written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C/C++ character strings may not be compatible with Fortran `CHARACTER` variables. However, we assume that a Fortran `INTEGER`, as well as a (sequence associated) Fortran array of `INTEGER`s, can be passed to a C or C++ program. We also assume that Fortran, C, and C++ have address-sized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that `INTEGER(KIND=MPI_OFFSET_KIND)` can be passed from Fortran to C as MPI_Offset.

4.12.3 Initialization

A call to MPI_INIT or MPI_THREAD_INIT, from any language, initializes MPI for execution in all languages.

> *Advice to users.* Certain implementations use the (inout) `argc`, `argv` arguments of the C/C++ version of MPI_INIT in order to propagate values for `argc` and `argv` to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

> *Advice to users.* The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

> *Advice to implementors.* Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

4.12.4 Transfer of Handles

Handles are passed between Fortran and C or C++ by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C or C++ handles in Fortran. Handles are passed between C and C++ using overloaded C++ operators called from C++ code. There is no direct access to C++ objects from C.

The type definition MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to int.

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa.

```
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
```

If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a valid C handle to that same communicator; if comm = MPI_COMM_NULL (Fortran value), then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then MPI_Comm_f2c returns an invalid C handle.

```
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
```

The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle to the same communicator; it maps a null handle into a null handle and an invalid handle into an invalid handle.

Similar functions are provided for the other types of opaque objects.

```
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Request MPI_Request_f2c(MPI_Fint request)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_File MPI_File_f2c(MPI_Fint file)

MPI_Fint MPI_File_c2f(MPI_File file)

MPI_Win MPI_Win_f2c(MPI_Fint win)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Info MPI_Info_f2c(MPI_Fint info)

MPI_Fint MPI_Info_c2f(MPI_Info info)
```

Example 4.9 The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
INTEGER DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END

/* C wrapper */

void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)
{
MPI_Datatype datatype;

datatype = MPI_Type_f2c( *f_handle);
*ierr = (MPI_Fint)MPI_Type_commit( &datatype);
*f_handle = MPI_Type_c2f(datatype);
return;
}
```

The same approach can be used for all other MPI functions. The call to MPI_xxx_f2c (resp. MPI_xxx_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

> *Rationale.* The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.
>
> Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

C and C++ The C++ language interface provides the functions listed below for mixed-language interoperability. The token <CLASS> is used below to indicate any valid MPI opaque handle name (e.g., Group), except where noted. For the case where the C++ class corresponding to <CLASS> has derived classes, functions are also provided for converting between the derived classes and the C MPI_<CLASS>.

The following function allows assignment from a C MPI handle to a C++ MPI handle.

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

The constructor below creates a C++ MPI object from a C MPI handle. This allows the automatic promotion of a C MPI handle to a C++ MPI handle.

```
MPI::<CLASS>::<CLASS>(const MPI_<CLASS>& data)
```

Example 4.10 In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

```
// C++ library function prototype
void cpp_lib_call(MPI::Comm& cpp_comm);

// Exported C function prototype
extern "C" {
void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm)
{
// the MPI_Comm (c_comm) is automatically promoted to MPI::Comm
cpp_lib_call(c_comm);
}
```

The following function allows conversion from C++ objects to C MPI handles. In this case, the casting operator is overloaded to provide the functionality.

```
MPI::<CLASS>::operator MPI_<CLASS>() const
```

Example 4.11 A C library routine is called from a C++ program. The C library routine is prototyped to take an `MPI_Comm` as an argument.

```
// C function prototype
extern "C" {
void c_lib_call(MPI_Comm c_comm);
}

void cpp_function()
{
// Create a C++ communicator, and initialize it with a dup of
//   MPI::COMM_WORLD
MPI::Intracomm cpp_comm(MPI::COMM_WORLD.Dup());
c_lib_call(cpp_comm);
}
```

> *Rationale.* Providing conversion from C to C++ via constructors and from C++ to C via casting allows the compiler to make automatic conversions. Calling C from C++ becomes trivial, as does the provision of a C or Fortran interface to a C++ library. (*End of rationale.*)

> *Advice to users.* Note that the casting and promotion operators return new handles by value. Using these new handles as INOUT parameters will affect the internal MPI object, but will *not* affect the original handle from which it was cast. (*End of advice to users.*)

It is important to note that all C++ objects and their corresponding C handles can be used interchangeably by an application. For example, an application can cache an attribute on MPI_COMM_WORLD and later retrieve it from MPI::COMM_WORLD.

4.12.5 Status

The following two procedures are provided in C to convert from a Fortran status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

```
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
```

If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, respectively. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

To do the conversion in the other direction, we have the following:

```
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)
```

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE.

> *Advice to users.* There is not a separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status. (*End of advice to users.*)

> *Rationale.* The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be MPI_Status**, which was considered an inferior solution. (*End of rationale.*)

4.12.6 MPI Opaque Objects

Unless said otherwise, opaque objects are "the same" in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail, issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see 4.12.9).

Example 4.12

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR
INTEGER (KIND=MPI_ADDRESS_KIND) ADDR

! create an absolute datatype for array R
CALL MPI_GET_ADDRESS( R, ADDR, IERR)
CALL MPI_TYPE_CREATE_STRUCT(1, 5, ADDR, MPI_REAL, TYPE, IERR)
CALL C_ROUTINE(TYPE)

/* C code */

void C_ROUTINE(MPI_Fint *ftype)
{
int count = 5;
int lens[2] = {1,1};
MPI_Aint displs[2];
MPI_Datatype types[2], newtype;

/* create an absolute datatype for buffer that consists   */
/*  of count, followed by R(5)                            */

MPI_Get_address(&count, &displs[0]);
displs[1] = 0;
types[0] = MPI_INT;
types[1] = MPI_Type_f2c(*ftype);
MPI_Type_create_struct(2, lens, displs, types, &newtype);
MPI_Type_commit(&newtype);

MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
```

```
/* the message sent contains an int count of 5, followed  */
/* by the 5 REAL entries of the Fortran array R.          */
}
```

Advice to implementors. The following implementation can be used: MPI addresses, as returned by MPI_GET_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have the same value in Fortran and C/C++, then an additional test for buf = MPI_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI_BOTTOM even in C/C++, so as to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still avoid the test buf = MPI_BOTTOM, by using the displacement from MPI_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are assciated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language), and is used to generate the right calling sequence when the callback function is invoked. (*End of advice to implementors.*)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a "`stdargs`" argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

Reduce Operations

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C, C++, and Fortran datatypes. (*End of advice to users.*)

Addresses

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI::Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits.

This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of $\geq$ 4 GB. What should be the value returned in Fortran by MPI_ADDRESS, for a variable with an address above 2^{32}? The design described here addresses this issue, while maintaining compatibility with current Fortran codes.

The constant MPI_ADDRESS_KIND is defined so that, in Fortran 90, `INTEGER(KIND=MPI_ADDRESS_KIND))` is an address sized integer type (typically, but not necessarily, the size of an `INTEGER(KIND=MPI_ADDRESS_KIND)` is 4 on 32 bit address machines and 8 on 64 bit address machines). Similarly, the constant MPI_INTEGER_KIND is defined so that `INTEGER(KIND=MPI_INTEGER_KIND)` is a default size INTEGER.

There are seven functions that have address arguments: MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT MPI_TYPE_LB and MPI_TYPE_UB.

Four new functions are provided to supplement the first four functions in this list. These functions are described in Section 4.14. The remaining three functions are supplemented by the new function MPI_TYPE_GET_EXTENT, described in that same section. The new functions have the same functionality as the old functions in C/C++, or on Fortran systems where default INTEGERs are address sized. In Fortran, they accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint are used in 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 an appropriate integer type. The old functions will continue to be provided, for backward compatibility. However, users are encouraged to switch to the new functions, in Fortran, so as to avoid problems on systems with an address range $> 2^{32}$, and to provide compatibility across languages.

4.12.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{TYPE,COMM,WIN}_KEYVAL_CREATE call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

> *Advice to implementors.* This requires that attributes be tagged either as "C," "C++" or "Fortran," and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 5.7 of the MPI-1 standard define attributes arguments to be of type `void*` in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C/C++ pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C/C++ callee, or vice-versa.

MPI will store, internally, address sized attributes. If Fortran INTEGERs are smaller, then the Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C/C++. These functions are described in Section 8.8. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer valued attributes. C and C++ attribute functions put and get address valued attributes. Fortran attribute functions put and get integer valued attributes. When an integer valued attribute is accessed from C or C++, then MPI_xxx_get_attr will return the address of (a pointer to) the integer valued attribute. When an address valued attribute is accessed from Fortran, then MPI_xxx_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style (MPI-2) attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if old style (MPI-1)attribute functions are used.

Example 4.13 A. C to Fortran

```
  C code

static int i = 5;
void *p;
p = &i;
MPI_Comm_put_attr(..., p);
....

 Fortran code

INTEGER(kind = MPI_ADDRESS_KIND) val
CALL MPI_COMM_GET_ATTR(...,val,...)
IF(val.NE.5) THEN CALL ERROR
```

B. Fortran to C

```
   Fortran code

INTEGER(kind=MPI_ADDRESS_KIND) val
val = 55555
CALL MPI_COMM_PUT_ATTR(...,val,ierr)

   C code
```

```
int *p;
MPI_Comm_get_attr(...,&p, ...);
if (*p != 55555) error();
```

The predefined MPI attributes can be integer valued or address valued. Predefined integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a Fortran call. I.e., in Fortran, MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address valued predefined attributes, such as MPI_WIN_BASE behave as if they were put by a C call. I.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to `(void *)`.

> *Rationale.* The design is consistent with the behavior specified in MPI-1 for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. (*End of rationale.*)

> *Advice to implementors.* Implementations should tag attributes either as address attributes or as integer attributes, according to whether they were set in C or in Fortran. Thus, the right choice can be made when the attribute is retrieved. (*End of advice to implementors.*)

4.12.8 Extra State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a `COMMON` array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

4.12.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 4.12.4. Constants that specify maximum lengths of strings (see Section A.2.1 for a listing) have a value one less in Fortran than C/C++ since in C/C++ the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

> *Advice to users.* This definition means that it is safe in C/C++ to allocate a buffer to receive a string using a declaration like

```
        char name [MPI_MAX_NAME_STRING];
```

(*End of advice to users.*)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM must be in Fortran the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better ...) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

4.12.10 Interlanguage Communication

The type matching rules for communications in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 4.14 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, MYRANK
INTEGER(KIND=MPI_ADDRESS_KIND) ADDR

! create an absolute datatype for array R
CALL MPI_GET_ADDRESS( R, ADDR, IERR)
CALL MPI_TYPE_CREATE_STRUCT(1, 5, ADDR, MPI_REAL, TYPE, IERR)
CALL MPI_TYPE_COMMIT(TYPE, IERR)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
   CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
   CALL C_ROUTINE(TYPE)
END IF


/* C code */
```

```
void C_ROUTINE(MPI_Fint *fhandle)
{
MPI_Datatype type;
MPI_Status status;

type = MPI_Type_f2c(*fhandle);

MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI_INTEGER and be received with datatype MPI_INT. However, such code is not portable.

4.13 Error Handlers

MPI-1 attached error handlers only to communicators. MPI-2 attaches them to three types of objects: communicators, windows, and files. The extension was done while maintaining only one type of error handler opaque object. On the other hand, there are, in C and C++, distinct typedefs for user defined error handling callback functions that accept, respectively, 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-1 function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

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.

4.13.1 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. 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
```

In C++, the user routine should be of the form:

```
typedef void MPI::Comm::Errhandler_fn(MPI::Comm &, int *, ...);
```

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.

4.13.2 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)
```

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.

4.13.3 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)
```

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.

4.14 New Datatype Manipulation Functions

New functions are provided to supplement the type manipulation functions that have address sized integer arguments. The new functions will use, in their Fortran binding, address-sized INTEGERs, thus solving problems currently encountered when the application address range is $> 2^{32}$. Also, a new, more convenient type constructor is provided to modify the lower bound and extent of a datatype. The deprecated functions replaced by the new functions here are listed in Section 2.6.1.

4.14.1 Type Constructors with Explicit Addresses

The four functions below supplement the four corresponding type constructor functions from MPI-1. The new functions are synonymous with the old functions in C/C++, or on Fortran systems where default INTEGERs are address sized. (The old names are not available in C++.) In Fortran, these functions accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint are used in 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. The old functions will continue to be provided for backward compatibility. However, users are encouraged to switch to the new functions, in both Fortran and C.

The new functions are listed below. The use of the old functions is deprecated.

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, STIDE, 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
```

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 (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
```

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (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 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[])
```

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)
```

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.*)

4.14.2 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 by the MPI-1 standard, Section 3.12.2).

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 recieve 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::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.*)

4.14.3 True Extent of Datatypes

Suppose we implement gather 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 new 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

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(Readers should compare this with the definitions in Section 3.12.3 of the MPI-1 standard, which describes the function MPI_TYPE_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.14.4 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.

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:

$$\begin{aligned} \mathsf{newtype} \;=\; & \text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \end{aligned}$$

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 4.1 defines the base step. Equation 4.2 defines the recursion step when order $=$ MPI_ORDER_FORTRAN, and Equation 4.3 defines the recursion step when order $=$ MPI_ORDER_C.

$$\begin{aligned}
&\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \qquad (4.1)\\
&\qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\})\\
&= \{(\mathsf{MPI_LB}, 0),\\
&\quad (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex),\\
&\quad (type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1},\\
&\qquad disp_{n-1} + (start_0 + 1) \times ex), \ldots\\
&\quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),\\
&\quad (\mathsf{MPI_UB}, size_0 \times ex)\}
\end{aligned}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \qquad (4.2)\\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype})\\
&= \text{Subarray}(ndims - 1, \{size_1, size_2, \ldots, size_{ndims-1}\},\\
&\qquad \{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\},\\
&\qquad \{start_1, start_2, \ldots, start_{ndims-1}\},\\
&\qquad\quad \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \mathsf{oldtype}))
\end{aligned}$$

$$\begin{aligned}
&\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \qquad (4.3)\\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype})\\
&= \text{Subarray}(ndims - 1, \{size_0, size_1, \ldots, size_{ndims-2}\},\\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\},\\
&\qquad \{start_0, start_1, \ldots, start_{ndims-2}\},\\
&\qquad\quad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \mathsf{oldtype}))
\end{aligned}$$

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 9.9.2.

4.14.5 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [12] 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). 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 4.15.) 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 in MPI-1.

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-1. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding functions provided in MPI-1. (*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 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

$$(\text{array\_of\_gsizes[i]} + \text{array\_of\_psizes[i]} - 1)/\text{array\_of\_psizes[i]}.$$

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:

$$\begin{aligned}
&\text{cyclic}(darg, gsize, r, psize, \mathsf{oldtype}) \\
&\quad = \{(\mathsf{MPI_LB}, 0), \\
&\qquad (type_0, disp_0 + r \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\qquad \ldots \\
&\qquad (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex), \\
&\\
&\qquad (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&\qquad (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
\end{aligned}$$

$$
\begin{aligned}
&\ldots\\
&(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),\\
&\qquad \vdots\\
&(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),\\
&(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex\\
&\qquad\qquad +psize \times darg \times ex \times (count - 1)),\\
&\ldots\\
&(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex\\
&\qquad\qquad +psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex\\
&\qquad\qquad +psize \times darg \times ex \times (count - 1)),\\
&(\mathsf{MPI\_UB}, gsize * ex)\}
\end{aligned}
$$

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 4.15 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)
```

4.15 New Predefined Datatypes

4.15.1 Wide Characters

A new datatype, MPI_WCHAR, is added, for the purpose of dealing with international character sets such as Unicode.

MPI_WCHAR is a C type that corresponds to the type `wchar_t` defined in `<stddef.h>`. There are no predefined reduction operations for MPI_WCHAR.

Rationale. The fact that MPI_CHAR is associated with the C datatype `char`, which in turn is often used as a substitute for the "missing" `byte` datatype in C makes it most natural to define this as a new datatype specifically for multi-byte characters. (*End of rationale.*)

4.15.2 Signed Characters and Reductions

MPI-1 doesn't allow reductions on signed or unsigned `char`s. Since this restriction (formally) prevents a C programmer from performing reduction operations on such types (which could be useful, particularly in an image processing application where pixel values are often represented as "unsigned char"), we now specify a way for such reductions to be carried out.

MPI-1.2 already has the C types MPI_CHAR and MPI_UNSIGNED_CHAR. However there is a problem here in that MPI_CHAR is intended to represent a character, not a small integer, and therefore will be translated between machines with different character representations.

To overcome this, a new MPI predefined datatype, MPI_SIGNED_CHAR, is added to the predefined datatypes of MPI-2, which corresponds to the ANSI C and ANSI C++ datatype `signed char`.

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 bit 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.*)

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. This is an extension to MPI-1.2, since MPI-1.2 does not allow the use of MPI_UNSIGNED_CHAR in reduction operations (and does not have the MPI_SIGNED_CHAR type).

In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable charater, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

4.15.3 Unsigned long long Type

A new type, MPI_UNSIGNED_LONG_LONG in C and MPI::UNSIGNED_LONG_LONG in C++ is added as an optional datatype.

Rationale. The ISO C9X committee has voted to include `long long` and `unsigned long long` as standard C types. (*End of rationale.*)

4.16 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the "external32" data format specified in Section 9.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but for MPI-2 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_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, incount, datatype, 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
```

4.17 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 4.12.4, 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.*)

4.18 Profiling Interface

The profiling interface, as described in Chapter 8 of MPI-1.1, must be supported for all MPI-2 functions, except those allowed as macros (See Section 4.17). This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_ for each MPI function. The profiling interface in C++ is described in Section 10.1.10.

For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version. This is a change from MPI-1.2.

Chapter 5

Process Creation and Management

5.1 Introduction

MPI-1 provides an interface that allows processes in a parallel program to communicate with one another. MPI-1 specifies neither how the processes are created, nor how they establish communication. Moreover, an MPI-1 application is static; that is, no processes can be added to or deleted from an application after it has been started.

MPI users have asked that the MPI-1 model be extended to allow process creation and management after an MPI application has been started. A major impetus comes from the PVM [7] research effort, which has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control in MPI-2 because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. MPI-2 assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for adding process management to MPI are both technical and practical. Important classes of message passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features is a practical stumbling block to migration.

While process management is essential, adding it to MPI should not compromise the portability or performance of MPI applications. In particular:

- The MPI-2 process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.

- MPI must not take over operating system responsibilities. It should instead provide a clean interface between an application and system software.

- MPI must continue to guarantee communication determinism, i.e., process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.
- MPI-1 programs must work under MPI-2, i.e., the MPI-1 static process model must be a special case of the MPI-2 dynamic model.

The MPI-2 process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI-2 does not change the concept of communicator. Once a communicator is built, it behaves as specified in MPI-1. A communicator is never changed once created, and it is always created using deterministic collective operations.

5.2 The MPI-2 Process Model

The MPI-2 process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other.

5.2.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager, which can range from a parallel operating system (CMOST) to layered software (POE) to an `rsh` command (p4).

MPI_COMM_SPAWN starts MPI processes and establishes communication with them, returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different binaries (or the same binary with different arguments), placing them in the same MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the existing group abstraction to represent processes. A process is identified by a (group, rank) pair.

5.2.2 The Runtime Environment

The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

- **MPP managed by a batch queueing system**. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

- **Network of workstations with PVM**. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.

- **Network of workstations managed by a load balancing system**. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.

- **Large SMP with Unix**. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — `pvm_addhosts`, `pvm_config`, `pvm_tasks`, etc., possibly modified to return an MPI (group,rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.

- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

- An attribute MPI_UNIVERSE_SIZE on MPI_COMM_WORLD tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

5.3 Process Manager Interface

5.3.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process may belong to several groups.

5.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

> *Advice to users.* It is possible in MPI to start a static SPMD or MPMD application by starting first one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI-1 application. (*End of advice to users.*)

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

IN	command	name of program to be spawned (string, significant only at root)
IN	argv	arguments to command (array of strings, significant only at root)
IN	maxprocs	maximum number of processes to start (integer, significant only at root)
IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	intracommunicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
OUT	array_of_errcodes	one code per process (array of integer)

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info,
              int root, MPI_Comm comm, MPI_Comm *intercomm,
```

```
            int array_of_errcodes[])

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
            ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
            const char* argv[], int maxprocs, const MPI::Info& info,
            int root, int array_of_errcodes[]) const

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
            const char* argv[], int maxprocs, const MPI::Info& info,
            int root) const
```

MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is collective over comm, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the as the ordering of the group of the comm in the parents and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

Advice to users. An implementation may automatically establish communication before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned intercommunicator can be used immediately). (*End of advice to users.*)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in `$HOME/pvm3/bin/$PVM_ARCH`). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (*End of advice to implementors.*)

If the program named in command does not call MPI_INIT, but instead forks a process that calls MPI_INIT, the results are undefined. Implementations may allow this case to work but are not required to.

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and C++ and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in C, C++ and Fortran to indicate an empty argument list. In C and C++, this constant is the same as NULL.

Example 5.1 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:

```
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

In Fortran:

```
CHARACTER*25 command, argv(3)
command = ' ocean '
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_COMM_SPAWN(command, argv, ...)
```

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] of MPI_COMM_SPAWN, etc. Second, argv of MPI_COMM_SPAWN must be null-terminated, so that its length can be determined. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv of main(), the MPI implementation may add the arguments to the argv that is passed to MPI_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set $\{m_i : 0 \leq m_i \leq \mathsf{maxprocs}\}$ of allowed values for the number of processes spawned. The set $\{m_i\}$ does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes, MPI_COMM_SPAWN returns successfully and the number of spawned processes, m, is given by the size of the remote group of intercomm. If m is less than maxproc, reasons why the other processes were not spawned are given in array_of_errcodes as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI_COMM_SPAWN raises an error of class MPI_ERR_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than maxprocs processes may be returned is called soft. See Section 5.3.4 for more information on the soft key for info.

> *Advice to users.* By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0 \ldots N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (*End of advice to users.*)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C, MPI::Info in C++ and INTEGER in Fortran. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated char* in C, character*(*) in Fortran). Routines to create and manipulate the info argument are described in Section 4.10.

For the SPAWN calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over process locations should use MPI_INFO_NULL.

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 5.3.4). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ($0 \le m <$ maxprocs) processes are spawned, m of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. In C++ this constant does not exist, and the array_of_errcodes argument may be omitted from the argument list.

> *Advice to implementors.* MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4. (*End of advice to implementors.*)

MPI_COMM_GET_PARENT(parent)

OUT	parent	the parent communicator (handle)

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

```
MPI_COMM_GET_PARENT(PARENT, IERROR)
    INTEGER PARENT, IERROR
```

```
static MPI::Intercomm MPI::Comm::Get_parent()
```

If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the "parent" intercommunicator of the current process. This parent intercommunicator is created implicitly inside of MPI_INIT and is the same intercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.

After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.

Advice to users. MPI_COMM_GET_PARENT returns a handle to a single intercommunicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. (*End of advice to users.*)

Rationale. The desire of the Forum was to create a constant MPI_COMM_PARENT similar to MPI_COMM_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI_COMM_DISCONNECT, which is explicitly allowed. (*End of rationale.*)

5.3.3 Starting Multiple Executables and Establishing Communication

While MPI_COMM_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI_COMM_WORLD.

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)

IN	count	number of commands (positive integer, significant to MPI only at root — see advice to users)
IN	array_of_commands	programs to be executed (array of strings, significant only at root)
IN	array_of_argv	arguments for commands (array of array of strings, significant only at root)
IN	array_of_maxprocs	maximum number of processes to start for each command (array of integer, significant only at root)
IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root)
IN	root	rank of process in which previous arguments are examined (integer)
IN	comm	intracommunicator containing group of spawning processes (handle)
OUT	intercomm	intercommunicator between original group and newly spawned group (handle)
OUT	array_of_errcodes	one error code per process (array of integer)

```
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
              char **array_of_argv[], int array_of_maxprocs[],
              MPI_Info array_of_info[], int root, MPI_Comm comm,
              MPI_Comm *intercomm, int array_of_errcodes[])
```

```
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
```

```
MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
             const char* array_of_commands[], const char** array_of_argv[],
             const int array_of_maxprocs[], const MPI::Info array_of_info[],
             int root, int array_of_errcodes[])
```

```
MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
             const char* array_of_commands[], const char** array_of_argv[],
             const int array_of_maxprocs[], const MPI::Info array_of_info[],
             int root)
```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the jth argument to command number i.

Rationale. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv *must* be the same as count. (*End of rationale.*)

Advice to users. The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be `(char ***)0` in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null (`(char *)0` in C and empty string in Fortran).

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first command, m_2 by the second, etc. The processes corresponding to the first command have ranks $0, 1, \ldots, m_1-1$. The processes in the second command have ranks $m_1, m_1+1, \ldots, m_1+m_2-1$. The processes in the third have ranks $m_1+m_2, m_1+m_2+1, \ldots, m_1+m_2+m_3-1$, etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDs whereas

MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, so the two methods are not completely equivalent. There are also two performance-related reasons why, if you need to spawn multiple executables, you may want to use MPI_COMM_SPAWN_MULTIPLE instead of calling MPI_COMM_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (*End of advice to users.*)

The array_of_errcodes argument is 1-dimensional array of size $\sum_{i=1}^{count} n_i$, where n_i is the ith element of array_of_maxprocs. Command number i corresponds to the n_i contiguous slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^{i} n_j\right] - 1$. Error codes are treated as for MPI_COMM_SPAWN.

Example 5.2 Examples of array_of_argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program "atmos" with argument "atmos.grd" in C:

```
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
char *argv1[] = {"atmos.grd", (char *)0};
array_of_argv[0] = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Here's how you do it in Fortran:

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = ' ocean '
array_of_argv(1, 1) = ' -gridfile '
array_of_argv(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '

commands(2) = ' atmos '
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
```

5.3.4 Reserved Keys

The following keys are reserved. An implementation is not required to interpret these keys, but if it does interpret the key, it must provide the functionality described.

host Value is a hostname. The format of the hostname is determined by the implementation.

arch Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.

wdir Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.

path Value is a directory or set of directories where the implementation should look for the executable. The format of path is determined by the implementation.

file Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.

soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

1. `a` means a
2. `a:b` means $a, a+1, a+2, \ldots, b$
3. `a:b:c` means $a, a+c, a+2c, \ldots, a+ck$, where for $c > 0$, k is the largest integer for which $a + ck \leq b$ and for $c < 0$, k is the largest integer for which $a + ck \geq b$. If $b > a$ then c must be positive. If $b < a$ then c must be negative.

Examples:

1. `a:b` gives a range between a and b
2. `0:N` gives full "soft" functionality
3. `1,2,4,8,16,32,64,128,256,512,1024,2048,4096` allows power-of-two number of processes.
4. `2:10000:2` allows even number of processes.
5. `2:10:2,7` allows 2, 4, 6, 7, 8, or 10 processes.

5.3.5 Spawn Example

Manager-worker Example, Using MPI_SPAWN.

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
   if (world_size != 1)    error("Top heavy with management");

   MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                &universe_sizep, &flag);
   if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
        scanf("%d", &universe_size);
   } else universe_size = *universe_sizep;
   if (universe_size == 1) error("No room to start workers");

   /*
    * Now spawn the workers. Note that there is a run-time determination
    * of what type of worker to spawn, and presumably this calculation must
    * be done at run time and cannot be calculated before starting
    * the program. If everything is known when the application is
    * first started, it is generally better to start them all at once
    * in a single MPI_COMM_WORLD.
    */

   choose_worker_program(worker_program);
   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
             MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
             MPI_ERRCODES_IGNORE);
   /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0,..
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
    * "everyone".
    */

   MPI_Finalize();
   return 0;
}

/* worker */

#include "mpi.h"
int main(int argc, char *argv[])
{
   int size;
   MPI_Comm parent;
   MPI_Init(&argc, &argv);
   MPI_Comm_get_parent(&parent);
   if (parent == MPI_COMM_NULL) error("No parent!");
   MPI_Comm_remote_size(parent, &size);
   if (size != 1) error("Something's wrong with the parent");
```

```
   /*
    * Parallel code here.
    * The manager is represented as the process with rank 0 in (the remote
    * group of) MPI_COMM_PARENT.  If the workers need to communicate among
    * themselves, they can use MPI_COMM_WORLD.
    */

   MPI_Finalize();
   return 0;
}
```

5.4 Establishing Communication

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

1. Two parts of an application that are started independently need to communicate.

2. A visualization tool wants to attach to a running process.

3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI intercommunicator, where the two groups of the intercommunicator are the original sets of of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

> *Advice to users.* While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that doesn't participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

5.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication — Catch 22.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client doesn't really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal, the user gives this address to the client program.
- The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers.

A port_name is a *system-supplied* string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port_name with the MPI_OPEN_PORT routine. It accepts a connection to a given port with MPI_COMM_ACCEPT. A client uses port_name to connect to the server.

By itself, the port_name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port_name to the client. It would be more convenient if a server could specify that it be known by an *application-supplied* service_name so that the client could connect to that service_name without knowing the port_name.

An MPI implementation may allow the server to publish a (port_name, service_name) pair with MPI_PUBLISH_NAME and the client to retrieve the port name from the service name with MPI_LOOKUP_NAME. This allows three levels of portability, with increasing levels of functionality.

1. Applications that do not rely on the ability to publish names are the most portable. Typically the port_name must be transferred "by hand" from server to client.
2. Applications that use the MPI_PUBLISH_NAME mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published.
3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.

5.4.2 Server Routines

A server makes itself available with two routines. First it must call MPI_OPEN_PORT to establish a port at which it may be contacted. Secondly it must call MPI_COMM_ACCEPT to accept connections from clients.

MPI_OPEN_PORT(info, port_name)

IN	info	implementation-specific information on how to establish an address (handle)
OUT	port_name	newly established port (string)

```
int MPI_Open_port(MPI_Info info, char *port_name)
```

```
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR
```

```
void MPI::Open_port(const MPI::Info& info, char* port_name)
```

This function establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system, possibly using information in the info argument.

MPI copies a system-supplied port name into port_name. port_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_PORT_NAME.

> *Advice to users.* The system copies the port name into port_name. The application must pass a buffer of sufficient size to hold this value. (*End of advice to users.*)

port_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.

> *Advice to implementors.* These examples are not meant to constrain implementations. A port_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (*End of advice to implementors.*)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with MPI_CLOSE_PORT and released by the system.

> *Advice to implementors.* Since the user may type in port_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (*End of advice to implementors.*)

info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI_INFO_NULL in order to get the implementation defaults.

MPI_CLOSE_PORT(port_name)

IN	port_name	a port (string)

```
int MPI_Close_port(char *port_name)
```

```
MPI_CLOSE_PORT(PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERROR
```

```
void MPI::Close_port(const char* port_name)
```

This function releases the network address represented by port_name.

MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)

IN	port_name	port name (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	intracommunicator over which call is collective (handle)
OUT	newcomm	intercommunicator with client as remote group (handle)

```
int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm,
             MPI_Comm *newcomm)
```

```
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

```
MPI::Intercomm MPI::Intracomm::Accept(const char* port_name,
             const MPI::Info& info, int root) const
```

MPI_COMM_ACCEPT establishes communication with a client. It is collective over the calling communicator. It returns an intercommunicator that allows communication with the client.

The port_name must have been established through a call to MPI_OPEN_PORT.

info is a implementation-defined string that may allow fine control over the ACCEPT call.

5.4.3 Client Routines

There is only one routine on the client side.

MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)

IN	port_name	network address (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	intracommunicator over which call is collective (handle)
OUT	newcomm	intercommunicator with server as remote group (handle)

```
int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
             MPI_Comm comm, MPI_Comm *newcomm)
```

```
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

```
MPI::Intercomm MPI::Intracomm::Connect(const char* port_name,
             const MPI::Info& info, int root) const
```

This routine establishes communication with a server specified by port_name. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an MPI_COMM_ACCEPT.

If the named port does not exist (or has been closed), MPI_COMM_CONNECT raises an error of class MPI_ERR_PORT.

If the port exists, but does not have a pending MPI_COMM_ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls MPI_COMM_ACCEPT. In the case of a time out, MPI_COMM_CONNECT raises an error of class MPI_ERR_PORT.

> *Advice to implementors.* The time out period may be arbitrarily short or long. However, a high quality implementation will try to queue connection attempts so that a server can handle simultaneous requests from several clients. A high quality implementation may also provide a mechanism, through the info arguments to MPI_OPEN_PORT, MPI_COMM_ACCEPT and/or MPI_COMM_CONNECT, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port_name is the address of the server. It must be the same as the name returned by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port_name, an implementation may accept them as well. For instance, if port_name is (hostname:port), an implementation may accept (ip_address:port) as well.

5.4.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service_name, port_name) pair is published by the server, and may be retrieved by a client using the service_name only. An MPI implementation defines the *scope* of the service_name, that is, the domain over which the service_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High quality implementations will give some control to users through the info arguments to name publishing functions. Examples are given in the descriptions of individual functions.

MPI_PUBLISH_NAME(service_name, info, port_name)

IN	service_name	a service name to associate with the port (string)
IN	info	implementation-specific information (handle)
IN	port_name	a port name (string)

```
int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)
```

```
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

```
void MPI::Publish_name(const char* service_name, const MPI::Info& info,
            const char* port_name)
```

This routine publishes the pair (port_name, service_name) so that an application may retrieve a system-supplied port_name using a well-known service_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI_PUBLISH_NAME.

MPI permits publishing more than one service_name for a single port_name. On the other hand, if service_name has already been published within the scope determined by info, the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

Note that while service_name has a limited scope, determined by the implementation, port_name always has global scope within the communication universe used by the implementation (i.e., it is globally unique).

port_name should be the name of a port established by MPI_OPEN_PORT and not yet deleted by MPI_CLOSE_PORT. If it is not, the result is undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of "ocean" running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI_PUBLISH_NAME after the first may fail. There is no universal solution to this.

To handle these situations, a high quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

MPI_UNPUBLISH_NAME(service_name, info, port_name)

IN	service_name	a service name (string)
IN	info	implementation-specific information (handle)
IN	port_name	a port name (string)

```
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)
```

```
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

```
void MPI::Unpublish_name(const char* service_name, const MPI::Info& info,
             const char* port_name)
```

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI_ERR_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, the implementation may require that info passed to MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish a name.

MPI_LOOKUP_NAME(service_name, info, port_name)

IN	service_name	a service name (string)
IN	info	implementation-specific information (handle)
OUT	port_name	a port name (string)

```
int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)
```

```
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
```

```
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

```
void MPI::Lookup_name(const char* service_name, const MPI::Info& info,
           char* port_name)
```

This function retrieves a port_name published by MPI_PUBLISH_NAME with service_name. If service_name has not been published, it raises an error in the error class MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the largest possible port name (see discussion above under MPI_OPEN_PORT).

If an implementation allows multiple entries with the same service_name within the same scope, a particular port_name is chosen in a way determined by the implementation.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.

5.4.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

ip_port Value contains IP port number at which to establish a port. (Reserved for MPI_OPEN_PORT only).

ip_address Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_OPEN_PORT call is made, the results are undefined. (Reserved for MPI_OPEN_PORT only).

5.4.6 Client/Server Examples

Simplest Example — Completely Portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```
    char myport[MPI_MAX_PORT_NAME];
    MPI_Comm intercomm;
    /* ... */
    MPI_Open_port(MPI_INFO_NULL, myport);
    printf("port name is: %s\n", myport);

    MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
    /* do something with intercomm */
```

The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:

```
    MPI_Comm intercomm;
    char name[MPI_MAX_PORT_NAME];
```

```
    printf("enter port name: ");
    gets(name);
    MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

Ocean/Atmosphere - Relies on Name Publishing

In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names.

```
    MPI_Open_port(MPI_INFO_NULL, port_name);
    MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);

    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
    /* do something with intercomm */
    MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

On the client side:

```
    MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                      &intercomm);
```

Simple Client-Server Example.

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag `1`. A message with tag `0` tells the server to exit.

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm client;
    MPI_Status status;
    char port_name[MPI_MAX_PORT_NAME];
    double buf[MAX_DATA];
    int    size, again;

    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 1) error(FATAL, "Server too big");
    MPI_Open_port(MPI_INFO_NULL, port_name);
    printf("server available at %s\n",port_name);
    while (1) {
        MPI_Comm_accept( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                         &client );
        again = 1;
        while (again) {
```

```
            MPI_Recv( buf, MAX_DATA, MPI_DOUBLE,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status );
            switch (status.MPI_TAG) {
                case 0: MPI_Comm_free( &client );
                        MPI_Close_port(port_name);
                        MPI_Finalize();
                        return 0;
                case 1: MPI_Comm_disconnect( &client );
                        again = 0;
                        break;
                case 2: /* do something */
                ...
                default:
                        /* Unexpected message type */
                        MPI_Abort( MPI_COMM_WORLD, 1 );
                }
            }
        }
}
```

Here is the client.

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm server;
    double buf[MAX_DATA];
    char port_name[MPI_MAX_PORT_NAME];

    MPI_Init( &argc, &argv );
    strcpy(port_name, argv[1] );/* assume server's name is cmd-line arg */

    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                      &server );

    while (!done) {
        tag = 2; /* Action to perform */
        MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
        /* etc */
        }
    MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
    MPI_Comm_disconnect( &server );
    MPI_Finalize();
    return 0;
}
```

5.5 Other Functionality

5.5.1 Universe Size

Many "dynamic" MPI applications are expected to exist in a static runtime environment, in which resources have been allocated before the application is run. When a user (or possibly a batch system) runs one of these quasi-static applications, she will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

MPI provides an attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows the application to obtain this information in a portable manner. This attribute indicates the total number of processes that are expected. In Fortran, the attribute is the integer value. In C, the attribute is a pointer to the integer value. An application typically subtracts the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE is determined by the application startup mechanism in a way not specified by MPI. (The size of MPI_COMM_WORLD is another example of such a parameter.)

Possibilities for how MPI_UNIVERSE_SIZE might be set include

- A `-universe_size` argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through the MPI process startup mechanism, such as `mpiexec`) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, MPI_UNIVERSE_SIZE is not updated, and the application must find out about the change through direct communication with the runtime system.

5.5.2 Singleton MPI_INIT

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI_INIT. Such a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and

MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

> *Advice to implementors.* To start an MPI-1 application with more than one process requires some special coordination. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.
>
> When an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. MPI-1 does not say what happens if these special steps were not taken — presumably this is treated as an error in starting the MPI application. MPI-2 recommends the following behavior.
>
> If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.
>
> In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either
>
> 1. Create the environment (e.g., start a daemon) or
> 2. Raise an error if it cannot create the environment and the environment has not been started independently.
>
> A high quality implementation will try to create a singleton MPI process and not raise an error.
>
> (*End of advice to implementors.*)

5.5.3 MPI_APPNUM

There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementation-specific startup mechanism that can handle multiple process specifications, MPI_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

```
mpiexec spec0 [: spec1 : spec2 : ...]
```

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM doesn't make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer that overrides the default value for MPI_APPNUM in the child.

Rationale. When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI_COMM_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI_APPNUM provides such a general mechanism. (*End of rationale.*)

5.5.4 Releasing Connections

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI_COMM_CONNECT and MPI_COMM_ACCEPT, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- Two processes are **connected** if there is a communication path (direct or indirect) between them. More precisely:
  1. Two processes are connected if
     (a) they both belong to the same communicator (inter- or intra-, including MPI_COMM_WORLD) *or*
     (b) they have previously belonged to a communicator that was freed with MPI_COMM_FREE instead of MPI_COMM_DISCONNECT *or*
     (c) they both belong to the group of the same window or filehandle.
  2. If A is connected to B and B to C, then A is connected to C.
- Two processes are **disconnected** (also **independent**) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
- Processes which are connected, but don't share the same MPI_COMM_WORLD may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.

The following additional rules apply to MPI-1 functions:

- MPI_FINALIZE is collective over a set of connected processes.
- MPI_ABORT does not abort independent processes. As in MPI-1, it may abort all processes in MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.
- If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.

MPI_COMM_DISCONNECT(comm)

INOUT	comm	communicator (handle)

```
int MPI_Comm_disconnect(MPI_Comm *comm)
```

```
MPI_COMM_DISCONNECT(COMM, IERROR)
    INTEGER COMM, IERROR
```

```
void MPI::Comm::Disconnect()
```

This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.

It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.

MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.

MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE and MPI_FILE_CLOSE to remove all communication paths between the two processes. Notes that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (*End of advice to users.*)

Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (*End of rationale.*)

5.5.5 Another Way to Establish MPI Communication

MPI_COMM_JOIN(fd, intercomm)

IN	fd	socket file descriptor
OUT	intercomm	new intercommunicator (handle)

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

```
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
    INTEGER FD, INTERCOMM, IERROR
```

```
static MPI::Intercomm MPI::Comm::Join(const int fd)
```

MPI_COMM_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [14, 17]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN and should return MPI_COMM_NULL.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

> *Advice to users.* An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (*End of advice to users.*)

> *Advice to implementors.* A high quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (*End of advice to implementors.*)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Non-blocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI_COMM_JOIN. The two processes are referred to as the local and remote processes.

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an intercommunicator, but is able to leave the socket in its original state, with no pending communication, it succeeds and sets intercomm to MPI_COMM_NULL.

The socket must be quiescent before MPI_COMM_JOIN is called and after MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a `read` on the socket will not read any data that was written to the socket before the remote process called MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a `read` will not read any data that was written to the socket before the remote process returned from MPI_COMM_JOIN. It is the responsibility of the application to ensure the first condition, and the responsibility of the MPI implementation to ensure the second. In a multithreaded application, the application must ensure that one thread does not access the socket while another is calling MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently.

> *Advice to implementors.* MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (*End of advice to implementors.*)

MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI_COMM_JOIN on two connected processes (see Section 5.5.4 for the definition of connected) is undefined.

The returned communicator may be used to establish MPI communication with additional processes, through the usual MPI communicator creation mechanisms.

Chapter 6

One-Sided Communications

6.1 Introduction

Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or update at other processes. However, processes may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This may require all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form `A = B(map)`, where `map` is a permutation vector, and `A, B` and `map` are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver; and *synchronization* of sender with receiver. The RMA design separates these two functions. Three communication calls are provided: MPI_PUT (remote write), MPI_GET (remote read) and MPI_ACCUMULATE (remote update). A larger number of synchronization calls are provided that support different synchronization styles. The design is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be imposed by the user, using synchronization calls; the implementation can delay communication operations until the synchronization calls occur, for efficiency.

The design of the RMA functions allows implementors to take advantage, in many cases, of fast communication mechanisms provided by various platforms, such as coherent or noncoherent shared memory, DMA engines, hardware-supported put/get operations, communication coprocessors, etc. The most frequently used RMA communication mechanisms can be layered on top of message passing. However, support for asynchronous communication agents (handlers, threads, etc.) is needed, for certain RMA functions, in a distributed memory environment.

We shall denote by **origin** the process that performs the call, and by **target** the process in which the memory is accessed. Thus, in a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

6.2 Initialization

6.2.1 Window Creation

The initialization operation allows each process in an intracommunicator group to specify, in a collective operation, a "window" in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call.

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)

IN	base	initial address of window (choice)
IN	size	size of window in bytes (nonnegative integer)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle)
IN	comm	communicator (handle)
OUT	win	window object returned by the call (handle)

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, MPI_Win *win)
```

```
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

```
static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int
             disp_unit, const MPI::Info& info, const MPI::Intracomm& comm)
```

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor disp_unit specified by the target process, at window creation.

> *Rationale.* The window size is specified using an address sized integer, so as to allow windows that span more than 4 GB of address space. (Even if the physical memory size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) (*End of rationale.*)

Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) `sizeof(type)`, for a window that consists of an array of elements of type `type`. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (*End of advice to users.*)

The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key is predefined:

no_locks — if set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to erroneous results.

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 4.11) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allow "good" memory to be used for static variables.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

MPI_WIN_FREE(win)

INOUT	win	window object (handle)

```
int MPI_Win_free(MPI_Win *win)
```

```
MPI_WIN_FREE(WIN, IERROR)
```

```
    INTEGER WIN, IERROR
```

```
void MPI::Win::Free()
```

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: i.e., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. When the call returns, the window memory can be freed.

> *Advice to implementors.* MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win called free. This, to ensure that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. (*End of advice to implementors.*)

6.2.2 Window Attributes

The following three attributes are cached with a window, when the window is created.

MPI_WIN_BASE	window base address.
MPI_WIN_SIZE	window size, in bytes.
MPI_WIN_DISP_UNIT	displacement unit associated with the window.

In C, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag) and MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag) will return in base a pointer to the start of the window win, and will return in size and disp_unit pointers to the size and displacement unit of the window, respectively. And similarly, in C++.

In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror) and MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror) will return in base, size and disp_unit the (integer representation of) the base address, the size and the displacement unit of the window win, respectively. (The window attribute access functions are defined in Section 8.8.)

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

MPI_WIN_GET_GROUP(win, group)

IN	win	window object (handle)
OUT	group	group of processes which share access to the window (handle)

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

```
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR
```

```
MPI::Group MPI::Win::Get_group() const
```

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window. associated with win. The group is returned in group.

6.3 Communication Calls

MPI supports three RMA communication calls: MPI_PUT transfers data from the caller memory (origin) to the target memory; MPI_GET transfers data from the target memory to the caller memory; and MPI_ACCUMULATE updates locations in the target memory, e.g. by adding to these locations values sent from the caller memory. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 6.4.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call, until the subsequent synchronization call completes.

Rationale. The rule above is more lenient than for message passing, where we do not allow two concurrent sends, with overlapping send buffers. Here, we allow two concurrent puts with overlapping send buffers. The reasons for this relaxation are

1. Users do not like that restriction, which is not very natural (it prohibits concurrent reads).
2. Weakening the rule does not prevent efficient implementation, as far as we know.
3. Weakening the rule is important for performance of RMA: we want to associate one synchronization call with as many RMA operations is possible. If puts from overlapping buffers cannot be concurrent, then we need to needlessly add synchronization points in the code.

(*End of rationale.*)

It is erroneous to have concurrent conflicting accesses to the same memory location in a window; if a location is updated by a put or accumulate operation, then this location cannot be accessed by a load or another RMA operation until the updating operation has completed at the target. There is one exception to this rule; namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred in some order. In addition, a window cannot concurrently be updated by a put or accumulate operation and by a local store operation. This, even if these two updates access different locations in the window. The last restriction enables more efficient implementations of RMA operations on many systems. These restrictions are described in more detail in Section 6.7.

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all three calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

Rationale. The choice of supporting "self-communication" is the same as for message passing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

6.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (nonnegative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (nonnegative integer)
IN	target_disp	displacement from start of window to target buffer (nonnegative integer)
IN	target_count	number of entries in target buffer (nonnegative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)

```
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target_rank, MPI_Aint target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
```

```
void MPI::Win::Put(const void* origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype) const
```

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base + target_disp×disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The target_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The target_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process, by the same sequence of calls used to define it at the origin process. The target datatype must contain only relative displacements, not absolute addresses. The same holds for get and accumulate.

> *Advice to users.* The target_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment, if only portable datatypes are used (portable datatypes are defined in Section 2.4).
>
> The performance of a put transfer can be significantly affected, on some systems, from the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI_ALLOC_MEM may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

> *Advice to implementors.* A high quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurred. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

6.3.2 Get

MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

OUT	origin_addr	initial address of origin buffer (choice)
IN	origin_count	number of entries in origin buffer (nonnegative integer)
IN	origin_datatype	datatype of each entry in origin buffer (handle)
IN	target_rank	rank of target (nonnegative integer)
IN	target_disp	displacement from window start to the beginning of the target buffer (nonnegative integer)
IN	target_count	number of entries in target buffer (nonnegative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	win	window object used for communication (handle)

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target_rank, MPI_Aint target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
```

```
void MPI::Win::Get(const void *origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype) const
```

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window, and the copied data must fit, without truncation, in the origin buffer.

6.3.3 Examples

Example 6.1 We show how to implement the generic indirect assignment `A = B(map)`, where `A`, `B` and `map` have the same distribution, and `map` is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
```

```
REAL A(m), B(m)

INTEGER otype(p), oindex(m),   & ! used to construct origin datatypes
     ttype(p), tindex(m),      & ! used to construct target datatypes
     count(p), total(p),       &
     sizeofreal, win, ierr

! This part does the work that depends on the locations of B.
! Can be reused while this does not change

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,   &
                     comm, win, ierr)

! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change

! Compute number of entries to be received from each process

DO i=1,p
  count(i) = 0
END DO
DO i=1,m
  j = map(i)/m+1
  count(j) = count(j)+1
END DO

total(1) = 0
DO i=2,p
  total(i) = total(i-1) + count(i-1)
END DO

DO i=1,p
  count(i) = 0
END DO

! compute origin and target indices of entries.
! entry i at current process is received from location
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1..p and k = 1..m

DO i=1,m
  j = map(i)/m+1
  k = MOD(map(i),m)+1
  count(j) = count(j)+1
  oindex(total(j) + count(j)) = i
  tindex(total(j) + count(j)) = k
```

```
END DO

! create origin and target datatypes for each get operation
DO i=1,p
  CALL MPI_TYPE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1),   &
                              MPI_REAL, otype(i), ierr)
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
  CALL MPI_TYPE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1),   &
                             MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
  CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
RETURN
END
```

Example 6.2 A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER sizeofreal, win, ierr

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/p
  k = MOD(map(i),p)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
```

```
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

6.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process.

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

IN	origin_addr	initial address of buffer (choice)
IN	origin_count	number of entries in buffer (nonnegative integer)
IN	origin_datatype	datatype of each buffer entry (handle)
IN	target_rank	rank of target (nonnegative integer)
IN	target_disp	displacement from start of window to beginning of target buffer (nonnegative integer)
IN	target_count	number of entries in target buffer (nonnegative integer)
IN	target_datatype	datatype of each entry in target buffer (handle)
IN	op	reduce operation (handle)
IN	win	window object (handle)

```
int MPI_Accumulate(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
```

```
void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype, const MPI::Op& op) const
```

Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count and origin_datatype) to the buffer specified by arguments target_count and target_datatype, at offset target_disp, in the target window specified by target_rank and win, using the operation op. This is like MPI_PUT except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for MPI_REDUCE can be used. User-defined functions cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative function $f(a, b) = b$; i.e., the current value in the target memory is replaced by the value supplied by the origin.

> *Advice to users.* MPI_PUT is a special case of MPI_ACCUMULATE, with the operation MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

Example 6.3 We want to compute $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$. The arrays A, B and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, sizeofreal, win, ierr
REAL A(m), B(m)

CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/p
  k = MOD(map(i),p)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL,   &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 6.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, then the code computes $\mathtt{B} = \mathtt{A(map^{-1})}$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 6.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

6.4 Synchronization Calls

RMA communications fall in two categories:

- **active target** communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.
- **passive target** communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

   This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

   These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

3. Finally, shared and exclusive locks are provided by the two functions MPI_WIN_LOCK and MPI_WIN_UNLOCK. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a “billboard” model, where processes can, at random times, access or update different parts of the billboard.

   These two calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK and terminated by a call to MPI_WIN_UNLOCK. Only one target window can be accessed during that epoch with win.

Figure 6.1 illustrates the general synchronization pattern for active target communication. The synchronization between `post` and `start` ensures that the put call of the origin process does not start until the target process exposes the window (with the `post` call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between `complete` and `wait` ensures that the put call of the origin process completes before the window is unexposed (with the `wait` call). The target process will execute following local accesses to the target window only after the `wait` returned.

Figure 6.1 shows operations occurring in the natural temporal order implied by the synchronizations: the `post` occurs before the matching `start`, and `complete` occurs before the matching `wait`. However, such **strong** synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow **weak** synchronization, as illustrated in Figure 6.2. The access to the target window is delayed until the window is exposed, after the `post`. However the `start` may complete earlier; the `put` and `complete` may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 6.3 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The `lock` and `unlock` calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the `put` by origin 1 will precede the `get` by origin 2.

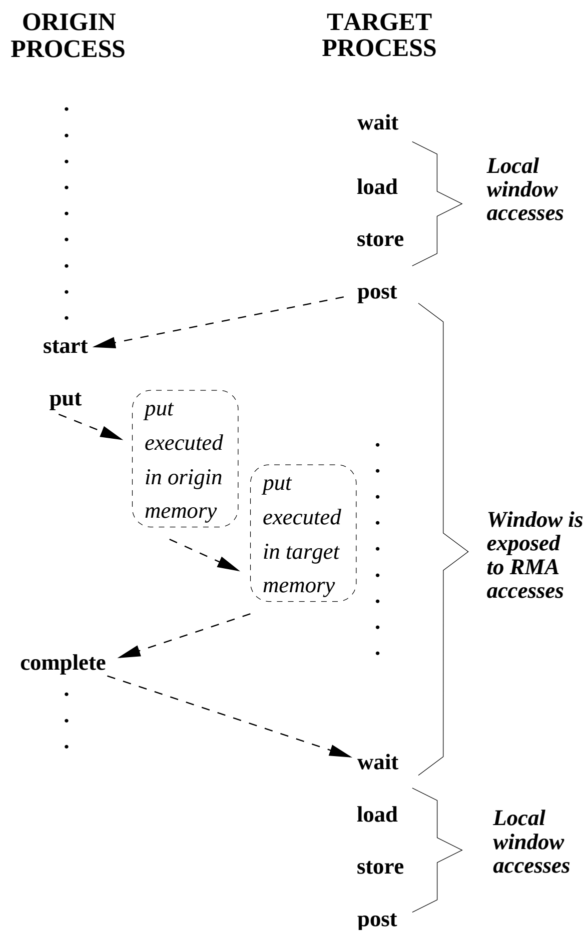

Figure 6.1: active target communication. Dashed arrows represent synchronizations (ordering of events).

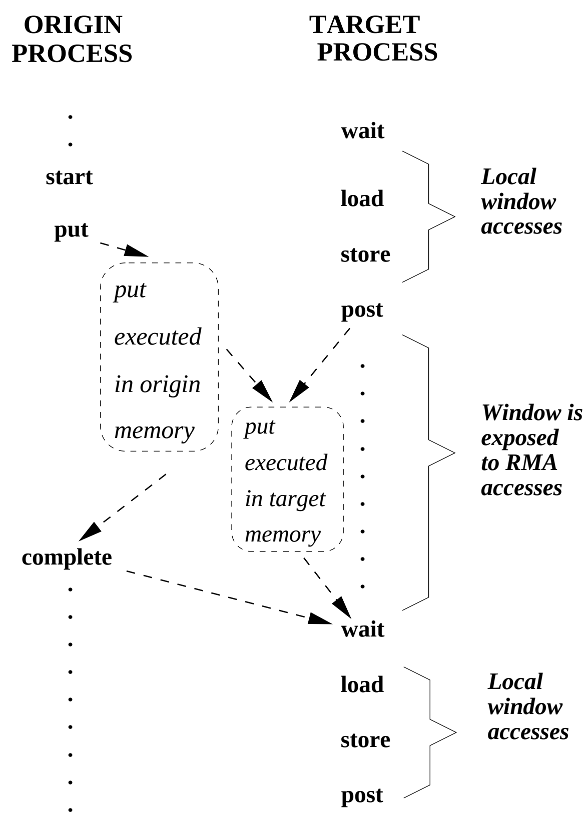

Figure 6.2: active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

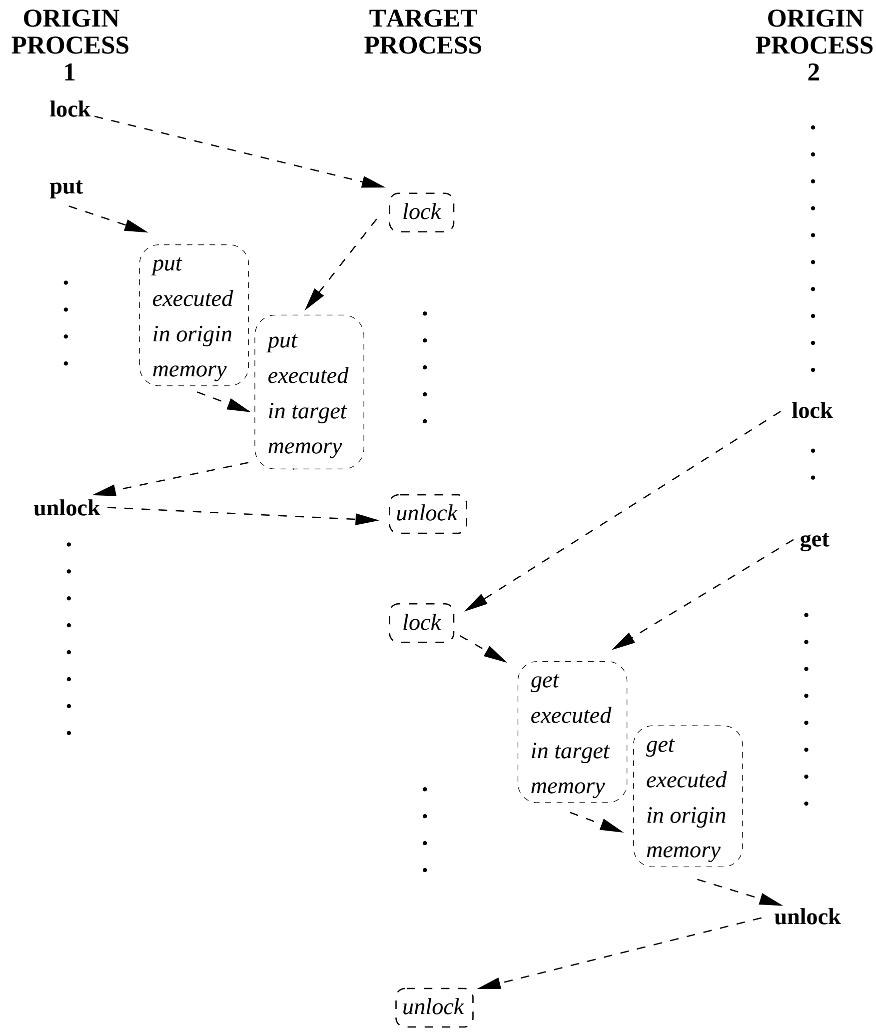

Figure 6.3: passive target communication. Dashed arrows represent synchronizations (ordering of events).

6.4.1 Fence

MPI_WIN_FENCE(assert, win)

IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_fence(int assert, MPI_Win win)
```

```
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

```
void MPI::Win::Fence(int assert) const
```

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of `post, start, complete, wait`.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with assert = MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 6.4.4. A value of assert = 0 is always valid.

> *Advice to users.* Calls to MPI_WIN_FENCE should both precede and follow calls to put, get or accumulate that are synchronized with fence calls. (*End of advice to users.*)

6.4.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

```
void MPI::Win::Start(const MPI::Group& group, int assert) const
```

Starts an RMA access epoch for win. RMA calls issued on win during this epoch must access only windows at processes in group. Each process in group must issue a matching call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not required to.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 6.4.4. A value of assert = 0 is always valid.

MPI_WIN_COMPLETE(win)

IN	win	window object (handle)

```
int MPI_Win_complete(MPI_Win win)
```

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

```
void MPI::Win::Complete() const
```

Completes an RMA access epoch on win started by a call to MPI_WIN_START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns.

MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

Example 6.4

```
MPI_Win_start(group, flag, win);
MPI_Put(...,win);
MPI_Win_complete(win);
```

The call to MPI_WIN_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. This still leaves much choice to implementors. The call to MPI_WIN_START can block until the matching call to MPI_WIN_POST occurs at all target processes. One can also have implementations where the call to MPI_WIN_START is nonblocking, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurred; or implementations where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls can complete before any target process called MPI_WIN_POST — the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)

IN	group	group of origin processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

```
void MPI::Win::Post(const MPI::Group& group, int assert) const
```

Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.

MPI_WIN_WAIT(win)

IN	win	window object (handle)

```
int MPI_Win_wait(MPI_Win win)
```

```
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

```
void MPI::Win::Wait() const
```

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 6.4 illustrates the use of these four functions. Process 0 puts data in the windows

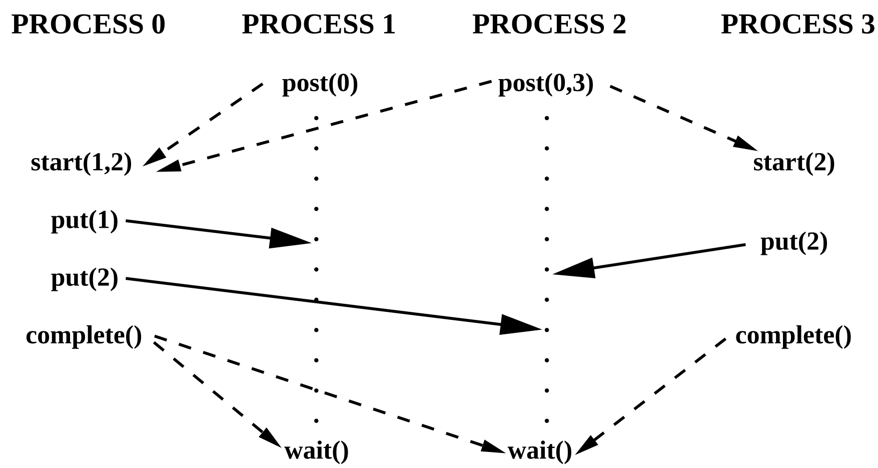

Figure 6.4: active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

MPI_WIN_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (logical)

```
int MPI_Win_test(MPI_Win win, int *flag)
```

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

```
bool MPI::Win::Test() const
```

This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if MPI_WIN_WAIT would return, flag = false, otherwise. The effect of return of MPI_WIN_TEST with flag = true is the same as the effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

MPI_WIN_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in group, using wincomm. No need to wait for the completion of these sends.

MPI_WIN_START(group,0,win) initiate a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.

MPI_WIN_COMPLETE(win) initiate a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.

MPI_WIN_WAIT(win) initiate a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice-versa.

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs, in general: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = < V, E >$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST($ingroup_i$, ...), followed by a call to MPI_WIN_START($outgroup_i$,...), where $outgroup_i = \{j \ : \ ij \in E\}$ and $ingroup_i = \{j \ : \ ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

6.4.3 Lock

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)
IN	rank	rank of locked window (nonnegative integer)
IN	assert	program assertion (integer)
IN	win	window object (handle)

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
```

```
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
```

```
void MPI::Win::Lock(int lock_type, int rank, int assert) const
```

Starts an RMA access epoch. Only the window at the process with rank rank can be accessed by RMA operations on win during that epoch.

MPI_WIN_UNLOCK(rank, win)

IN	rank	rank of window (nonnegative integer)
IN	win	window object (handle)

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

```
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

```
void MPI::Win::Unlock(int rank) const
```

Completes an RMA access epoch started by a call to MPI_WIN_LOCK(...,win). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock call, and to protect local load/store accesses to a locked local window executed between the lock and unlock call. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. I.e., a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

Rationale. An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM (Section 4.11). Locks can be used portably only in such memory.

Rationale. The implementation of passive target communication when memory is not shared requires an asynchronous agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for 3-rd party communication in shared memory machines.

The downside of this decision is that passive target communication cannot be used without taking advantage of nonstandard Fortran features: namely, the availability of C-like pointers; these are not supported by some Fortran compilers (g77 and Windows/NT compilers, at the time of writing). Also, passive target communication cannot be portably targeted to COMMON blocks, or other statically declared Fortran arrays. (*End of rationale.*)

Consider the sequence of calls in the example below.

Example 6.5

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win)
MPI_Put(..., rank, ..., win)
MPI_Win_unlock(rank, win)
```

The call to MPI_WIN_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI_WIN_LOCK may block until an exclusive lock on the window is acquired; or, the call MPI_WIN_LOCK may not block, while the call to MPI_PUT blocks until a lock is acquired; or, the first two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired — the update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs. However, if the call to MPI_WIN_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns.

6.4.4 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE and MPI_WIN_LOCK is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program — it is erroneous to provides incorrect information. Users may always provide assert = 0 to indicate a general case, where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent, shared memory machines. Users should consult their implementation manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations, whenever available. (*End of advice to users.*)

Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (*End of advice to implementors.*)

assert is the bit-vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE and MPI_MODE_NOSUCCEED. The significant options are listed below, for each call.

Advice to users. C/C++ users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector `IOR` intrinsic. Fortran 77 users can use (non-portably) bit vector `IOR` on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (*End of advice to users.*)

MPI_WIN_START:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

MPI_WIN_POST:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.

MPI_MODE_NOSTORE — the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

MPI_WIN_FENCE:

MPI_MODE_NOSTORE — the local window was not updated by local stores (or local get or receive calls) since last synchronization.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

MPI_MODE_NOPRECEDE — the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_WIN_LOCK:

MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

Advice to users. Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

6.4.5 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as argument to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message passing, datatypes must be committed before they can be used in RMA communication.

6.5 Examples

Example 6.6 The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array `A`, which contains the origin and target buffers of the put calls.

```
...
while(!converged(A)){
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                         todisp[i], 1, totype[i], win);
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }
```

The same code could be written with get, rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 6.7 Same generic example, with more computation/communication overlap. We assume that the update phase is broken in two subphases: the first, where the “boundary,” which is involved in communication, is updated, and the second, where the “core,” which neither use nor provide communicated data, is updated.

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
```

```
                  fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
  }
```

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the `update_core` call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Example 6.8 Same code as in Example 6.6, rewritten using post-start-complete-wait.

```
...
while(!converged(A)){
  update(A);
  MPI_Win_post(fromgroup, 0, win);
  MPI_Win_start(togroup, 0, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                         todisp[i], 1, totype[i], win);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

Example 6.9 Same example, with split phases, as in Example 6.7.

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
  MPI_Win_start(fromgroup, 0, win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                  fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

Example 6.10 A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array `A0` is updated using values of array `A1`, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window `wini` consists of array `Ai`.

```
...
if (!converged(A0,A1))
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
```

```
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
while(!converged(A0, A1)){
  /* communication on A0 and computation on A1 */
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
               fromdisp0[i], 1, fromtype0[i], win0);
  update1(A1); /* local update of A1 that is
                  concurrent with communication that updates A0 */
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
  MPI_Win_complete(win0);
  MPI_Win_wait(win0);

  /* communication on A1 and computation on A0 */
  update2(A0, A1); /* local update of A0 that depends on A1 (and A0)*/
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
  update1(A0); /* local update of A0 that depends on A0 only,
                 concurrent with communication that updates A1 */
  if (!converged(A0,A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
  MPI_Win_complete(win1);
  MPI_Win_wait(win1);
  }
```

A process posts the local window associated with `win0` before it completes RMA accesses to the remote windows associated with `win1`. When the `wait(win1)` call returns, then all neighbors of the calling process have posted the windows associated with `win0`. Conversely, when the `wait(win0)` call returns, then all neighbors of the calling process have posted the windows associated with `win1`. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

Put calls can be used, instead of get calls, if the area of array `A0` (resp. `A1`) used by the `update(A1, A0)` (resp. `update(A0, A1)`) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

6.6 Error Handling

6.6.1 Error Handlers

Errors occurring during calls to MPI_WIN_CREATE(...,comm,...) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

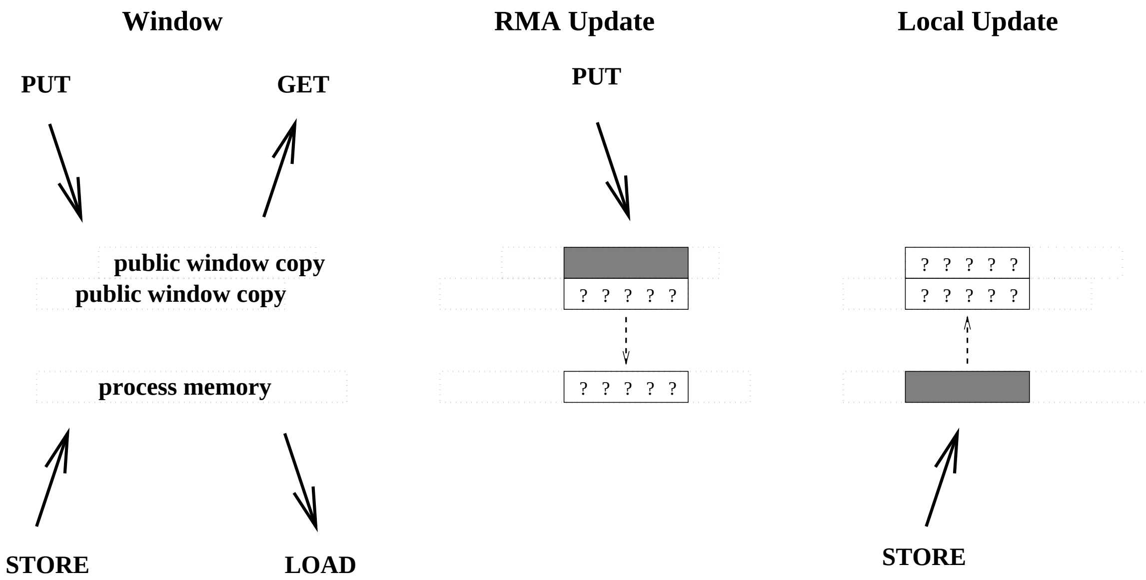

Figure 6.5: Schematic description of window

The default error handler associated with win is MPI_ERRORS_ARE_FATAL. Users may change this default by explicitly associating a new error handler with win (see Section 4.13).

6.6.2 Error Classes

The following new error classes are defined

MPI_ERR_WIN	invalid win argument
MPI_ERR_BASE	invalid base argument
MPI_ERR_SIZE	invalid size argument
MPI_ERR_DISP	invalid disp 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

6.7 Semantics and Correctness

The semantics of RMA operations is best understood by assuming that the system maintains a separate *public* copy of each window, in addition to the original location in process memory (the *private* window copy). There is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 6.5.

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specifies the latest time at which an update of one window copy becomes visible in another overlapping copy.

1. An RMA operation is completed at the origin by the ensuing call to MPI_WIN_COMPLETE, MPI_WIN_FENCE or MPI_WIN_UNLOCK that synchronizes this access at the origin.
2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.
3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.
4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK then the operation is completed at the target by that same call to MPI_WIN_UNLOCK.
5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, or MPI_WIN_UNLOCK is executed on that window by the window owner.
6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE, or MPI_WIN_LOCK is executed on that window by the window owner.

The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK. On the other hand, the update of private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed until the process executes a suitable synchronization call. Updates to a public window copy can also be delayed until the window owner executes a synchronization call, if fences or post-start-complete-wait synchronization is used. Only when lock synchronization is used does it becomes necessary to update the public window copy, even if the window owner does not execute any related synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

A correct program must obey the following rules.

1. A location in a window must not be accessed locally once an update to that location has started, until the update becomes visible in the private window copy in process memory.

2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates that use the same operation, with the same predefined datatype, on the same window.

3. A put or accumulate must not access a target window once a local update or a put or accumulate update to another (overlapping) target window have started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a local update in process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

A program is erroneous if it violates these rules.

Rationale. The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were locally updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Advice to users. A user can write correct programs by following the following rules:

fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.

post-start-complete-wait: A window should not be updated locally while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

lock: Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for local accesses and for RMA accesses.

changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two overlapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete. (*End of advice to users.*)

6.7.1 Atomicity

The outcome of concurrent accumulates to the same location, with the same operation and predefined datatype, is as if the accumulates where done at that location in some serial order. On the other hand, if two locations are both updated by two accumulate calls, then the updates may occur in reverse order at the two locations. Thus, there is no guarantee that the entire call to MPI_ACCUMULATE is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to MPI_ACCUMULATE, cannot be accessed by load or an RMA call other than accumulate, until the MPI_ACCUMULATE call has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative.

6.7.2 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled, then it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 6.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occur, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 6.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

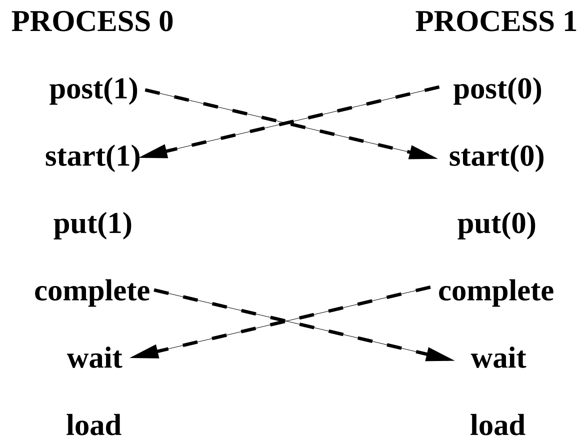

Figure 6.6: Symmetric communication

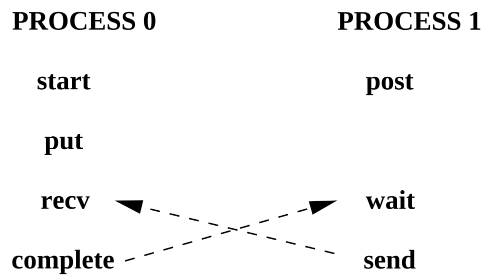

Figure 6.7: Deadlock situation

Consider the code illustrated in Figure 6.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice-versa. Consider the code illustrated in Figure 6.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 6.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

Rationale. MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 6.8, the put and complete calls of process 0 should complete

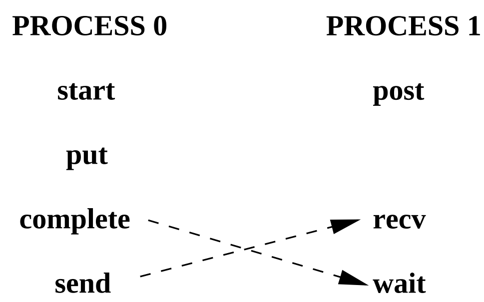

Figure 6.8: No deadlock

while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users. (*End of rationale.*)

6.7.3 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory.

The problem is illustrated by the following code:

Source of Process 1	**Source of Process 2**	**Executed in Process 2**
`bbbb = 777`	`buff = 999`	`reg_A:=999`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
`call MPI_PUT(bbbb`		`stop appl. thread`
`into buff of process 2)`		`buff:=777 in PUT handler`
		`continue appl. thread`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	

```
ccc = buff                    ccc:=reg_A
```

In this example, variable buff is allocated in the register reg_A and therefore ccc will have the old value of buff and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 2.

MPI implementations will avoid this problem for standard conforming C programs. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in COMMON blocks, or to variables that were declared VOLATILE (while VOLATILE is not a standard Fortran declaration, it is supported by many Fortran compilers). Details and an additional solution are discussed in Section 2, "A Problem with Register Optimization," . See also, "Problems Due to Data Copying and Sequence Association,", for additional Fortran problems.

Chapter 7

Extended Collective Operations

7.1 Introduction

MPI-1 defined collective communication for intracommunicators and two routines, MPI_INTERCOMM_CREATE and MPI_COMM_DUP, for creating new intercommunicators. In addition, in order to avoid argument aliasing problems with Fortran, MPI-1 requires separate send and receive buffers for collective operations. MPI-2 introduces extensions of many of the MPI-1 collective routines to intercommunicators, additional routines for creating intercommunicators, and two new collective routines: a generalized all-to-all and an exclusive scan. In addition, a way to specify "in place" buffers is provided for many of the intracommunicator collective operations.

7.2 Intercommunicator Constructors

The current MPI interface provides only two intercommunicator construction routines:

- MPI_INTERCOMM_CREATE, creates an intercommunicator from two intracommunicators,
- MPI_COMM_DUP, duplicates an existing intercommunicator (or intracommunicator).

The other communicator constructors, MPI_COMM_CREATE and MPI_COMM_SPLIT, currently apply only to intracommunicators. These operations in fact have well-defined semantics for intercommunicators [20].

In the following discussions, 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).

In addition, the specification of collective operations (Section 4.1 of MPI-1) requires that all collective routines are called with matching arguments. For the intercommunicator extensions, this is weakened to matching for all members of the same local group.

MPI_COMM_CREATE(comm_in, group, comm_out)

IN	comm_in	original communicator (handle)
IN	group	group of processes to be in new communicator (handle)
OUT	comm_out	new communicator (handle)

```
MPI::Intercomm MPI::Intercomm::Create(const Group& group) const
```

```
MPI::Intracomm MPI::Intracomm::Create(const Group& group) const
```

The C and Fortran language bindings are identical to those in MPI-1, so are omitted here.
If comm_in 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 7.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of comm_out. 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.*)

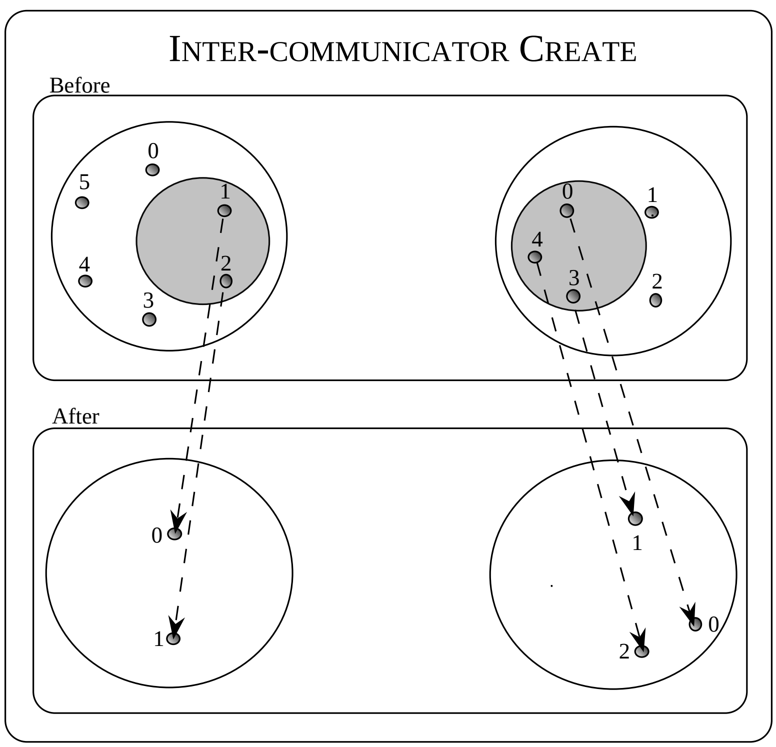

Figure 7.1: Intercommunicator create using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

Example 7.1 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_in, color, key, comm_out)

IN	comm_in	original communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	comm_out	new communicator (handle)

```
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const
```

```
MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

The C and Fortran language bindings are identical to those in MPI-1, so are omitted here.

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 7.2). 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 7.2 (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.

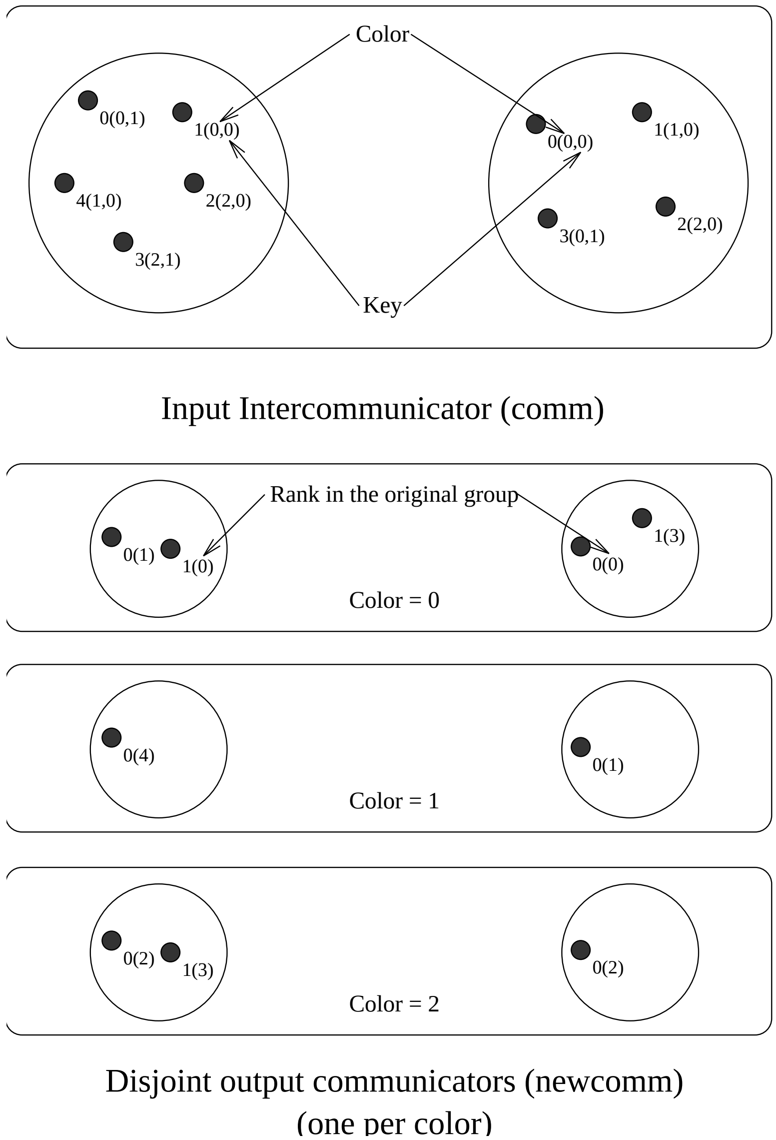

Figure 7.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.

```
/* 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 */
...

/* 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 );
```

7.3 Extended Collective Operations

7.3.1 Intercommunicator Collective Operations

In the MPI-1 standard (Section 4.2), collective operations only apply to intracommunicators; however, most MPI collective operations can be generalized to intercommunicators. To understand how MPI can be extended, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [20]):

All-To-All All processes contribute to the result. All processes receive the result.

- MPI_Allgather, MPI_Allgatherv

- MPI_Alltoall, MPI_Alltoallv
- 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_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 pattern of MPI_Scan does not fit this taxonomy.

The extension of collective communication from intracommunicators to intercommunicators is best described in terms of the left and right 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 7.3). 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 7.4). 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.

For MPI-2, the following intracommunicator collective operations also apply to intercommunicators:

- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
- MPI_REDUCE, MPI_ALLREDUCE,
- MPI_REDUCE_SCATTER,
- MPI_BARRIER.

(MPI_ALLTOALLW is a new function described in Section 7.3.5.)

These functions use exactly the same argument list as their MPI-1 counterparts and also work on intracommunicators, as expected. No new language bindings are consequently needed for Fortran or C. However, in C++, the bindings have been "relaxed"; these member functions have been moved from the `MPI::Intercomm` class to the `MPI::Comm` class. But since the collective operations do not make sense on a C++ `MPI::Comm` (since it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual. In an MPI-2 implementation, the bindings in this chapter supersede the corresponding bindings for MPI-1.2.

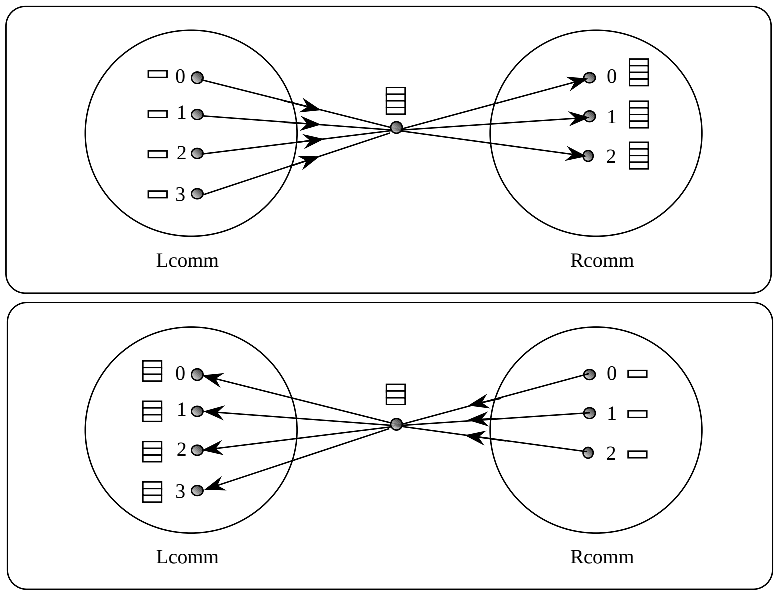

Figure 7.3: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

7.3.2 Operations that Move Data

Two additions are made to many collective communication calls:

- 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.

  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.*)

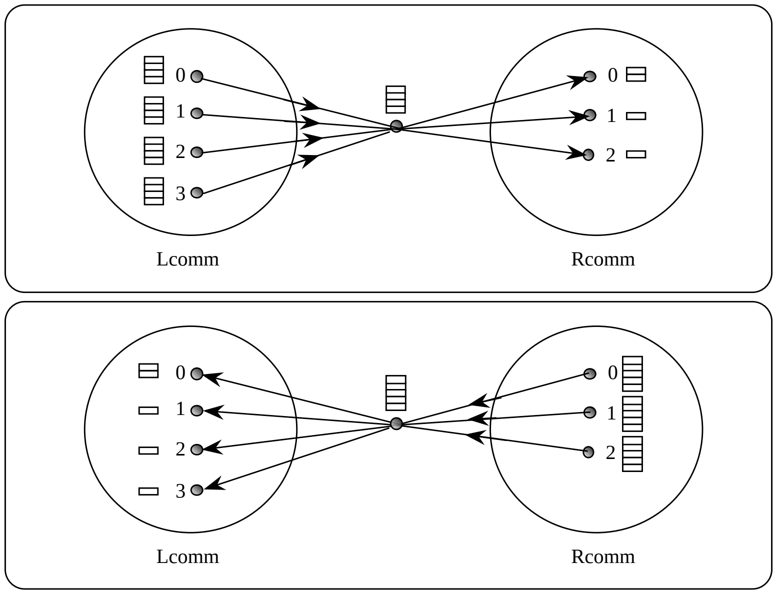

Figure 7.4: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

> *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.*)

- Collective communication applies to intercommunicators. 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. 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.

  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.

  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.*)

In the following, the definitions of the collective routines are provided to enhance the readability and understanding of the associated text. They do not change the definitions of the argument lists from MPI-1. The C and Fortran language bindings for these routines are unchanged from MPI-1, and are not repeated here. Since new C++ bindings for the intercommunicator versions are required, they are included. The text provided for each routine is appended to the definition of the routine in MPI-1.

Broadcast

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
void MPI::Comm::Bcast(void* buffer, int count,
              const MPI::Datatype& datatype, int root) const = 0
```

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 receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

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 (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 (integer, significant only at root)
IN	recvtype	data type of recv buffer elements (handle, significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
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
```

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 (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	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 (handle, significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
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
```

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.

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 (integer, significant only at root)
IN	sendtype	data type of send buffer elements (handle, significant only at root)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
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
```

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	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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
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
```

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.

"All" Forms and All-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 (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0
```

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. Specifically, the outcome of a call to MPI_ALLGATHER in the "in place" case is as if all processes executed n calls to

```
MPI_GATHER( MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcount,
                             recvtype, root, comm )
```

for `root = 0, ..., n - 1`.

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 (integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	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)

```
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
```

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. Specifically, the outcome of a call to MPI_ALLGATHER in the "in place" case is as if all processes executed n calls to

```
MPI_GATHERV( MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, recvbuf, recvcounts,
                             displs, recvtype, root, comm )
```

for `root = 0, ..., n - 1`.

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.

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 (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
            MPI::Datatype& sendtype, void* recvbuf, int recvcount,
            const MPI::Datatype& recvtype) const = 0
```

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	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	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)

```
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
```

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.

7.3.3 Reductions

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 (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)

```
void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
              const MPI::Datatype& datatype, const MPI::Op& op, int root)
              const = 0
```

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.

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 (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
              const MPI::Datatype& datatype, const MPI::Op& op) const = 0
```

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 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 the same count value.

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	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)

```
void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
            int recvcounts[], const MPI::Datatype& datatype,
            const MPI::Op& op) const = 0
```

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. Note that the area occupied by the input data may be either longer or shorter than the data filled 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 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.*)

7.3.4 Other Operations

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
void MPI::Comm::Barrier() const = 0
```

For MPI-2, comm may be an intercommunicator or an intracommunicator. If comm is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in the local group of the intercommunicator may exit the barrier when all of the processes in the remote group have entered the barrier.

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 (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

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 illegal for intercommunicators.

7.3.5 Generalized All-to-all Function

One of the basic data movement operations needed in parallel signal processing is the 2-D matrix transpose. This operation has motivated a generalization of the MPI_ALLTOALLV function. This new collective operation is MPI_ALLTOALLW; the "W" indicates that it is an extension to MPI_ALLTOALLV.

The following function 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.

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.*)

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 (integer)
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
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (handle)
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 (integer)
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
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (handle)
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
```

No “in place” option is supported.

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

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.

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.

7.3.6 Exclusive Scan

MPI-1 provides an inclusive scan operation. The exclusive scan is described here.

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 (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
```

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.

> *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 provided in MPI-1 as MPI_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.

The reason that MPI-1 chose the inclusive scan is that the definition of behavior on processes zero and one was thought to offer too many complexities in definition, particularly for user-defined operations. (*End of rationale.*)

Chapter 8

External Interfaces

8.1 Introduction

This chapter begins with calls used to create **generalized requests**. The objective of this MPI-2 addition is to allow users of MPI to be able to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 8.3 deals with setting the information found in status. This is needed for generalized requests.

Section 8.4 allows users to associate names with communicators, windows, and datatypes. This will allow debuggers and profilers to identify communicators, windows, and datatypes with more useful labels. Section 8.5 allows users to add error codes, classes, and strings to MPI. With users being able to layer functionality on top of MPI, it is desirable for them to use the same error mechanisms found in MPI.

Section 8.6 deals with decoding datatypes. 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 chapter continues, in Section 8.7, with a discussion of how threads are to be handled in MPI-2. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. Section 8.8 has information on caching on communicators, datatypes, and windows. Finally, Section 8.9 discusses duplicating a datatype.

8.2 Generalized Requests

The goal of this MPI-2 extension is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI_WAIT or MPI_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

> *Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is very difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI when the operation completes. This is done by making a call to MPI_GREQUEST_COMPLETE. MPI maintains the "completion" status of generalized requests. Any other request state has to be maintained by the user.

A new generalized request is started with

MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)

IN	query_fn	callback function invoked when request status is queried (function)
IN	free_fn	callback function invoked when request is freed (function)
IN	cancel_fn	callback function invoked when request is cancelled (function)
IN	extra_state	extra state
OUT	request	generalized request (handle)

```
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
            MPI_Grequest_free_function *free_fn,
            MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
            MPI_Request *request)
```

```
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
            IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
static MPI::Grequest
            MPI::Grequest::Start(const MPI::Grequest::Query_function
            query_fn, const MPI::Grequest::Free_function free_fn,
            const MPI::Grequest::Cancel_function cancel_fn,
            void *extra_state)
```

> *Advice to users.* Note that a generalized request belongs, in C++, to the class MPI::Grequest, which is a derived class of MPI::Request. It is of the same type as regular requests, in C and Fortran. (*End of advice to users.*)

The call starts a generalized request and returns a handle to it in request.

The syntax and meaning of the callback functions are listed below. All callback functions are passed the extra_state argument that was associated with the request by the starting call MPI_GREQUEST_START. This can be used to maintain user-defined state for the request. In C, the query function is

```
typedef int MPI_Grequest_query_function(void *extra_state,
            MPI_Status *status);
```

in Fortran

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

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

query_fn function computes the status that should be returned for the generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by MPI_TEST_CANCELLED).

query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that completed the generalized request associated with this callback. The callback function is also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI will pass a valid status object to query_fn, and this status will be ignored upon return of the callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE is called on the request; it may be invoked several times for the same generalized request, e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

and in Fortran

```
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

```
typedef int MPI::Grequest::Free_function(void* extra_state);
```

free_fn function is invoked to clean up user-allocated resources when the generalized request is freed.

free_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that completed the generalized request associated with this callback. free_fn is invoked after the call to query_fn for the same request. However, if the MPI call completed multiple generalized requests, the order in which free_fn callback functions are invoked is not specified by MPI.

free_fn callback is also invoked for generalized requests that are freed by a call to MPI_REQUEST_FREE (no call to WAIT_{WAIT|TEST}{ANY|SOME|ALL} will occur for such a request). In this case, the callback function will be called either in the MPI call MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), whichever happens last. I.e., in this case the actual freeing code is executed as soon as both calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request is not deallocated until after free_fn completes. Note that free_fn will be invoked only once per request by a correct program.

Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after free_fn completes since MPI does not deallocate the object until then. Since free_fn is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after free_fn executes. At this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid accessing this stale handle. This is a special case where MPI defers deallocating the object until a later time that is known by the user. (*End of advice to users.*)

In C, the cancel function is

```
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
```

in Fortran

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
```

and in C++

```
typedef int MPI::Grequest::Cancel_function(void* extra_state,
            bool complete);
```

cancel_fn function is invoked to start the cancelation of a generalized request. It is called by MPI_REQUEST_CANCEL(request). MPI passes to the callback function complete=true if MPI_GREQUEST_COMPLETE was already called on the request, and complete=false otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will return the error code returned by the last callback, namely free_fn. If one or more of the requests in a call to MPI_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free_fn callback function. However, if the MPI function was passed MPI_STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query_fn must **not** set the error field of status since query_fn may be called by MPI_WAIT or MPI_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query_fn is invoked and can decide correctly when to put in the error field of status the returned error code. (*End of advice to users.*)

MPI_GREQUEST_COMPLETE(request)

INOUT	request	generalized request (handle)

```
int MPI_Grequest_complete(MPI_Request request)
```

```
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

```
void MPI::Grequest::Complete()
```

The call informs MPI that the operations represented by the generalized request request are complete. (See definitions in Section 2.4.) A call to MPI_WAIT(request, status) will return and a call to MPI_TEST(request, flag, status) will return flag=true only after a call to MPI_GREQUEST_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, all these calls are supposed to be local and nonblocking. Therefore, the callback functions query_fn, free_fn, or cancel_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to MPI_GREQUEST_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

8.2.1 Examples

Example 8.1 This example shows the code for a user-defined reduce operation on an `int` using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```
typedef struct {
   MPI_Comm comm;
   int tag;
```

```
   int root;
   int valin;
   int *valout;
   MPI_Request request;
   } ARGS;


int myreduce(MPI_Comm comm, int tag, int root,
              int valin, int *valout, MPI_Request *request)
{
ARGS *args;
pthread_t thread;

/* start request */
MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);

args = (ARGS*)malloc(sizeof(ARGS));
args->comm = comm;
args->tag = tag;
args->root = root;
args->valin = valin;
args->valout = valout;
args->request = *request;

/* spawn thread to handle request */
/* The availability of the pthread_create call is system dependent */
pthread_create(&thread, NULL, reduce_thread, args);

return MPI_SUCCESS;
}


/* thread code */
void reduce_thread(void *ptr)
{
int lchild, rchild, parent, lval, rval, val;
MPI_Request req[2];
ARGS *args;

args = (ARGS*)ptr;

/* compute left,right child and parent in tree; set
   to MPI_PROC_NULL if does not exist  */
/* code not shown */
...

MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
```

```
MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
val = lval + args->valin + rval;
MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
if (parent == MPI_PROC_NULL) *(args->valout) = val;
MPI_Grequest_complete((args->request));
free(ptr);
return;
}

int query_fn(void *extra_state, MPI_Status *status)
{
/* always send just one int */
MPI_Status_set_elements(status, MPI_INT, 1);
/* can never cancel so always true */
MPI_Status_set_cancelled(status, 0);
/* choose not to return a value for this */
status->MPI_SOURCE = MPI_UNDEFINED;
/* tag has not meaning for this generalized request */
status->MPI_TAG = MPI_UNDEFINED;
/* this generalized request never fails */
return MPI_SUCCESS;
}


int free_fn(void *extra_state)
{
/* this generalized request does not need to do any freeing */
/* as a result it never fails here */
return MPI_SUCCESS;
}


int cancel_fn(void *extra_state, int complete)
{
/* This generalized request does not support cancelling.
   Abort if not already done.  If done then treat as if cancel failed. */
if (!complete) {
  fprintf(stderr, "Cannot cancel generalized request - aborting program\n");
  MPI_Abort(MPI_COMM_WORLD, 99);
  }
return MPI_SUCCESS;
}
```

8.3 Associating Information with Status

In MPI-1, requests were associated with point-to-point operations. In MPI-2 there are several different types of requests. These range from new MPI calls for I/O to generalized requests.

It is desirable to allow these calls use the same request mechanism. This allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

In MPI-2, each call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful value for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, new calls are provided:

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT	status	status to associate count with (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)

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

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

```
void MPI::Status::Set_elements(const MPI::Datatype& datatype, int count)
```

This call modifies the opaque part of status so that a call to MPI_GET_ELEMENTS will return count. MPI_GET_COUNT will return a compatible value.

Rationale. The number of elements is set instead of the count because the former can deal with nonintegral number of datatypes. (*End of rationale.*)

A subsequent call to MPI_GET_COUNT(status, datatype, count) or to MPI_GET_ELEMENTS(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS.

Rationale. This is similar to the restriction that holds when when count is set by a receive operation: in that case, the calls to MPI_GET_COUNT and MPI_GET_ELEMENTS must use a datatype with the same signature as the datatype used in the receive call. (*End of rationale.*)

MPI_STATUS_SET_CANCELLED(status, flag)

INOUT	status	status to associate cancel flag with (Status)
IN	flag	if true indicates request was cancelled (logical)

```
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
```

```
MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

```
void MPI::Status::Set_cancelled(bool flag)
```

If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.

> *Advice to users.* Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable results and is strongly discouraged. (*End of advice to users.*)

8.4 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.

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 MPI_COMM_PARENT 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)

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
```

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 the I/O chapter in MPI-2. 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 new functions are provided to do this. They are all local. No functions are provided to free error handlers or 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 not affected by new user-defined error codes and classes. As in MPI-1, it is a constant value. 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 4.13 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)

```
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 it is 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 Decoding a Datatype

MPI-1 provides datatype objects, which allow users to specify an arbitrary layout of data in memory. The layout information, once put in a datatype, could not be decoded from the datatype. There are several cases, however, where accessing the layout information in opaque datatype objects would be useful.

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.

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

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

For 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 new MPI-2 calls all use address sized arguments.

Rationale. For recreating the original call, it is important to know if address information may have been truncated. The MPI-1 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 (non-negative integer)
IN	max_addresses	number of elements in array_of_addresses (non-negative integer)
IN	max_datatypes	number of elements in array_of_datatypes (non-negative 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. This is analogous to the topology calls in MPI-1. (*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 MPI-1 datatype constructor calls, the address arguments in Fortran are of type `INTEGER`. In the new MPI-2 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 old MPI-1 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 new MPI-2 calls for datatype constructors for the deprecated MPI-1 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)

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.

Example 8.2 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;
}
```

8.7 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, it is not required that all MPI implementations fulfill all the requirements specified in this section.

This section generally assumes a thread package similar to POSIX threads [11], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

8.7.1 General

In a thread-compliant implementation, an MPI process is a process that may be multi-threaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

> *Rationale.* This model corresponds to the POSIX model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations where MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (*End of rationale.*)

> *Advice to users.* It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (*End of advice to users.*)

The two main requirements for a thread-compliant implementation are listed below.

1. All MPI calls are *thread-safe*. I.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.

2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Example 8.3 Process 0 consists of two threads. The first thread executes a blocking send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status). I.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

> *Advice to implementors.* MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

8.7.2 Clarifications

Initialization and Completion The call to MPI_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all the process threads have completed their MPI calls, and have no pending communications or I/O operations.

> *Rationale.* This constraint simplifies implementation. (*End of rationale.*)

Multiple threads completing the same request. A program where two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI_WAIT{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test which violates this rule is erroneous.

> *Rationale.* This is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)

Probe A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process.

Collective calls Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent threads issue such calls on the same communicator, window or file handle, it is up to the user to make sure the calls are correctly ordered, using interthread synchronization.

Exception handlers An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that is distinct from the thread that will return the error code.

> *Rationale.* The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

> *Rationale.* Few C library functions are signal safe, and many have cancellation points — points where the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe." (*End of rationale.*)

> *Advice to users.* Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to `sigwait` for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

> *Advice to implementors.* The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

8.7.3 Initialization

The following function may be used to initialize MPI, and initialize the MPI thread environment, instead of MPI_INIT.

MPI_INIT_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
              int *provided)
```

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR
```

```
int MPI::Init_thread(int& argc, char**& argv, int required)
```

```
int MPI::Init_thread(int required)
```

Advice to users. In C and C++, the passing of argc and argv is optional. In C, this is accomplished by passing the appropriate null pointer. In C++, this is accomplished with two separate bindings to cover these two cases. This is as with MPI_INIT as discussed in Section 4.2. (*End of advice to users.*)

This call initializes MPI in the same way that a call to MPI_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

MPI_THREAD_SINGLE Only one thread will execute.

MPI_THREAD_FUNNELED The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are "funneled" to the main thread).

MPI_THREAD_SERIALIZED The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").

MPI_THREAD_MULTIPLE Multiple threads may call MPI, with no restrictions.

These values are monotonic; i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.

Different processes in MPI_COMM_WORLD may require different levels of thread support.

The call returns in provided information about the actual level of thread support that will be provided by MPI. It can be one of the four values listed above.

The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to mpiexec). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

A **thread compliant** MPI implementation will be able to return provided = MPI_THREAD_MULTIPLE. Such an implementation may always return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required. At the other extreme, an MPI library that is not thread compliant may always return provided = MPI_THREAD_SINGLE, irrespective of the value of required.

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required = MPI_THREAD_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to mpiexec. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; on the other hand, a call to MPI_INIT will initialize the MPI thread support level to MPI_THREAD_SINGLE.

Rationale. Various optimizations are possible when MPI code is executed single-threaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits well many applications. E.g., if the process code is a sequential Fortran/C/C++ program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C/ C++/Fortran library calls that are not thread safe, e.g., in an environment where `malloc` is not thread safe, then `malloc` should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

MPI_QUERY_THREAD(provided)

OUT	provided	provided level of thread support (integer)

```
int MPI_Query_thread(int *provided)
```

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR
```

`int MPI::Query_thread()`

The call returns in provided the current level of thread support. This will be the value returned in provided by MPI_INIT_THREAD, if MPI was initialized by a call to MPI_INIT_THREAD().

MPI_IS_THREAD_MAIN(flag)

OUT	flag	true if calling thread is main thread, false otherwise (logical)

`int MPI_Is_thread_main(int *flag)`

```
MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

`bool MPI::Is_thread_main()`

This function can be called by a thread to find out whether it is the main thread (the thread that called MPI_INIT or MPI_INIT_THREAD).

All routines listed in this section must be supported by all MPI implementations.

Rationale. MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions be able to link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (*End of rationale.*)

Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI call other than MPI_INITIALIZED should be executed by these threads, until MPI_INIT_THREAD is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (*End of advice to users.*)

8.8 New Attribute Caching Functions

Caching on communicators has been a very useful feature. In MPI-2 it is expanded to include caching on windows and datatypes.

Rationale. In one extreme you 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 in MPI-1 is the potential for size differences between Fortran integers and C pointers. To overcome this problem with attribute caching on communicators, new functions are also given for this case. The new functions to cache on datatypes and windows also address this issue. For a general discussion of the address size problem, see Section 4.12.6.

The MPI-1.2 clarification, described in Section 3.2.8, about the effect of returning other than MPI_SUCCESS from attribute callbacks applies to these new versions as well.

8.8.1 Communicators

The new functions that are replacements for the MPI-1 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)
```

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 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.

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.

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);
```

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)
```

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 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
```

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)
```

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.

8.8.2 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

```
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);
```

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(const MPI::Win& win, 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)
```

8.8.3 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);
```

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)
```

8.9 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 new 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. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 8.6. The newtype has the same committed state as the old type.

Chapter 9

I/O

9.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [15], collective buffering [1, 2, 16, 19, 22], and disk-directed I/O [13]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

9.1.1 Definitions

file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.

displacement A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."

etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are nonnegative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

filetype A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be nonnegative and monotonically nondecreasing.

view A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 9.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

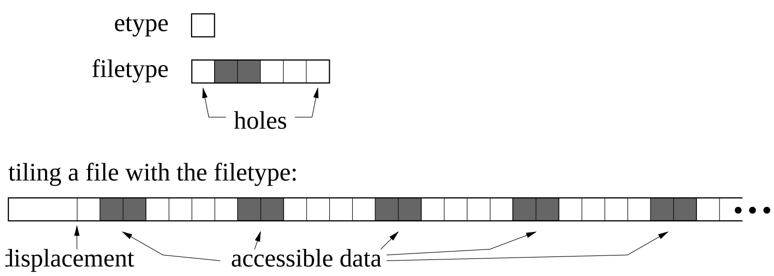

Figure 9.1: Etypes and filetypes

A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 9.2).

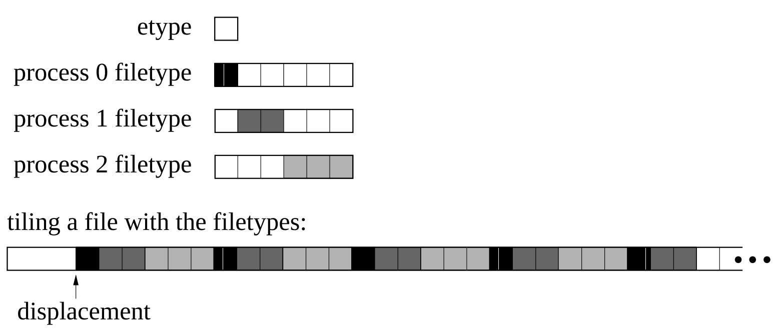

Figure 9.2: Partitioning a file among parallel processes

offset An *offset* is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 9.2 is the position of the 8th etype in the file after the displacement. An "explicit offset" is an offset that is used as a formal parameter in explicit data access routines.

file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.

file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.

file handle A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

9.2 File Manipulation

9.2.1 Opening a File

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)
IN	filename	name of file to open (string)
IN	amode	file access mode (integer)
IN	info	info object (handle)
OUT	fh	new file handle (handle)

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
             MPI_File *fh)
```

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR
```

```
static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
             const char* filename, int amode, const MPI::Info& info)
```

MPI_FILE_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_FILE_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference the same file. (Values for info may vary.) comm must be an intracommunicator; it is erroneous to pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN are raised using the default file error handler (see Section 9.7). A process can open a file independently of other processes by using the MPI_COMM_SELF communicator. The file handle returned, fh, can be subsequently used to access the file until the file is closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O behavior.

The format for specifying the file name in the filename argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (*End of advice to implementors.*)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY — read only,
- MPI_MODE_RDWR — reading and writing,
- MPI_MODE_WRONLY — write only,
- MPI_MODE_CREATE — create the file if it does not exist,
- MPI_MODE_EXCL — error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE — delete file on close,
- MPI_MODE_UNIQUE_OPEN — file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL — file will only be accessed sequentially,
- MPI_MODE_APPEND — set initial position of all file pointers to end of file.

Advice to users. C/C++ users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector `IOR` intrinsic. Fortran 77 users can use (non-portably) bit vector `IOR` on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (*End of advice to users.*)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [11]. Exactly one of MPI_MODE_RDONLY, MPI_MODE_RDWR, or MPI_MODE_WRONLY, must be specified. It is erroneous to specify MPI_MODE_CREATE or MPI_MODE_EXCL in conjunction with MPI_MODE_RDONLY; it is erroneous to specify MPI_MODE_SEQUENTIAL together with MPI_MODE_RDWR.

The MPI_MODE_DELETE_ON_CLOSE mode causes the file to be deleted (equivalent to performing an MPI_FILE_DELETE) when the file is closed.

The MPI_MODE_UNIQUE_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

> *Advice to users.* For MPI_MODE_UNIQUE_OPEN, *not opened elsewhere* includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI_MODE_UNIQUE_OPEN is specified, the user is responsible for ensuring that no such external events take place. (*End of advice to users.*)

The MPI_MODE_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI_MODE_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI_FILE_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI_ERR_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 9.2.8). The constant MPI_INFO_NULL can be used when no info needs to be specified.

> *Advice to users.* Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (*End of advice to users.*)

Files are opened by default using nonatomic mode file consistency semantics (see Section 9.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

9.2.2 Closing a File

MPI_FILE_CLOSE(fh)

INOUT	fh	file handle (handle)

```
int MPI_File_close(MPI_File *fh)
```

```
MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
```

```
void MPI::File::Close()
```

MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.

> *Advice to users.* If the file is deleted on close, and there are other processes currently accessing the file, the status of the file and the behavior of future accesses by these processes are implementation dependent. (*End of advice to users.*)

The user is responsible for ensuring that all outstanding nonblocking requests and split collective operations associated with fh made by a process have completed before that process calls MPI_FILE_CLOSE.

The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to MPI_FILE_NULL.

9.2.3 Deleting a File

MPI_FILE_DELETE(filename, info)

IN	filename	name of file to delete (string)
IN	info	info object (handle)

```
int MPI_File_delete(char *filename, MPI_Info info)
```

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR
```

```
static void MPI::File::Delete(const char* filename, const MPI::Info& info)
```

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 9.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default error handler (see Section 9.7).

9.2.4 Resizing a File

MPI_FILE_SET_SIZE(fh, size)

INOUT	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

```
void MPI::File::Set_size(MPI::Offset size)
```

MPI_FILE_SET_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space—use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

> *Advice to users.* It is possible for the file pointers to point beyond the end of file after a MPI_FILE_SET_SIZE operation truncates a file. This is legal, and equivalent to seeking beyond the current end of file. (*End of advice to users.*)

All nonblocking requests and split collective operations on fh must be completed before calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that conflicts with operations that access bytes at displacements between the old and new file sizes (see Section 9.6.1).

9.2.5 Preallocating Space for a File

MPI_FILE_PREALLOCATE(fh, size)

INOUT	fh	file handle (handle)
IN	size	size to preallocate file (integer)

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

```
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

```
void MPI::File::Preallocate(MPI::Offset size)
```

MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the group must pass identical values for size. Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE has the same effect as writing undefined data. If size is larger than the current file size, the file size increases to size. If size is less than or equal to the current file size, the file size is unchanged.

The treatment of file pointers, pending nonblocking accesses, and file consistency is the same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

> *Advice to users.* In some implementations, file preallocation may be expensive. (*End of advice to users.*)

9.2.6 Querying the Size of a File

MPI_FILE_GET_SIZE(fh, size)

IN	fh	file handle (handle)
OUT	size	size of the file in bytes (integer)

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

```
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

```
MPI::Offset MPI::File::Get_size() const
```

MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a data access operation (see Section 9.6.1).

9.2.7 Querying File Parameters

MPI_FILE_GET_GROUP(fh, group)

IN	fh	file handle (handle)
OUT	group	group which opened the file (handle)

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

```
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR
```

```
MPI::Group MPI::File::Get_group() const
```

MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to open the file associated with fh. The group is returned in group. The user is responsible for freeing group.

MPI_FILE_GET_AMODE(fh, amode)

IN	fh	file handle (handle)
OUT	amode	file access mode used to open the file (integer)

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

```
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR
```

```
int MPI::File::Get_amode() const
```

MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with fh.

Example 9.1 In Fortran 77, decoding an amode bit vector will require a routine such as the following:

```
      SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
!
!   TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
!   IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE
!
      INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
      BIT_FOUND = 0
      CP_AMODE = AMODE
 100  CONTINUE
      LBIT = 0
      HIFOUND = 0
      DO 20 L = MAX_BIT, 0, -1
         MATCHER = 2**L
         IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
            HIFOUND = 1
            LBIT = MATCHER
            CP_AMODE = CP_AMODE - MATCHER
         END IF
  20  CONTINUE
      IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
      IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
          CP_AMODE .GT. 0) GO TO 100
      END
```

This routine could be called successively to decode amode, one bit at a time. For example, the following code fragment would check for MPI_MODE_RDONLY.

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
   PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
   PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

9.2.8 File Info

Hints specified via info (see Section 4.10) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. However, hints do not change the semantics of any of the I/O interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

MPI_FILE_SET_INFO(fh, info)

INOUT	fh	file handle (handle)
IN	info	info object (handle)

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR
```

```
void MPI::File::Set_info(const MPI::Info& info)
```

MPI_FILE_SET_INFO sets new values for the hints of the file associated with fh. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. (*End of advice to users.*)

MPI_FILE_GET_INFO(fh, info_used)

IN	fh	file handle (handle)
OUT	info_used	new info object (handle)

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

```
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR
```

```
MPI::Info MPI::File::Get_info() const
```

MPI_FILE_GET_INFO returns a new info object containing the hints of the file associated with fh. The current setting of all hints actually used by the system related to this open file is returned in info_used. The user is responsible for freeing info_used via MPI_INFO_FREE.

Advice to users. The info object returned in info_used will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (*End of advice to users.*)

Reserved File Hints

Some potentially useful hints (info key values) are outlined below. The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described. (For more details on "info," see Section 4.10.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., file_perm is only useful during file creation).

access_style **(comma separated list of strings):** This hint specifies the manner in which the file will be accessed until the file is closed or until the access_style key value is altered. The hint value is a comma separated list of the following: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.

collective_buffering **(boolean) [SAME]:** This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Legal values for this key are true and false. Collective buffering parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and cb_nodes.

cb_block_size **(integer) [SAME]:** This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (CYCLIC) pattern.

cb_buffer_size **(integer) [SAME]:** This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of cb_block_size.

cb_nodes **(integer) [SAME]:** This hint specifies the number of target nodes to be used for collective buffering.

chunked **(comma separated list of integers) [SAME]:** This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

chunked_item **(comma separated list of integers) [SAME]:** This hint specifies the size of each array entry, in bytes.

chunked_size **(comma separated list of integers) [SAME]:** This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.

filename **(string):** This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.

file_perm **(string) [SAME]:** This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of legal values for this key is implementation dependent.

io_node_list **(comma separated list of strings) [SAME]:** This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.

nb_proc **(integer) [SAME]:** This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

num_io_nodes **(integer) [SAME]:** This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.

striping_factor **(integer) [SAME]:** This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

striping_unit **(integer) [SAME]:** This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

9.3 File Views

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT	fh	file handle (handle)
IN	disp	displacement (integer)
IN	etype	elementary datatype (handle)
IN	filetype	filetype (handle)
IN	datarep	data representation (string)
IN	info	info object (handle)

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
             MPI_Datatype filetype, char *datarep, MPI_Info info)
```

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

```
void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
             const MPI::Datatype& filetype, const char* datarep,
             const MPI::Info& info)
```

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 9.5.1 for further details.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

> *Advice to users.* disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 9.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

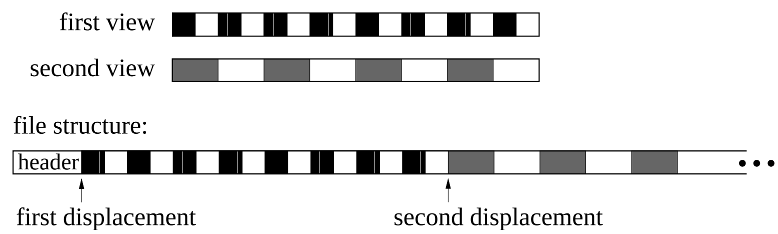

Figure 9.3: Displacements

> (*End of advice to users.*)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are nonnegative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

> *Advice to users.* In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 9.5). (*End of advice to users.*)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype and filetype arguments can be changed via future calls to MPI_FILE_SET_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype.

The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 9.2.8). The constant MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 9.5) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SET_VIEW—otherwise, the call to MPI_FILE_SET_VIEW is erroneous.

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

IN	fh	file handle (handle)
OUT	disp	displacement (integer)
OUT	etype	elementary datatype (handle)
OUT	filetype	filetype (handle)
OUT	datarep	data representation (string)

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
             MPI_Datatype *filetype, char *datarep)
```

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP, INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

```
void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype& etype,
             MPI::Datatype& filetype, char* datarep) const
```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

9.4 Data Access

9.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided:

positioning	**synchronism**	**coordination**	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI_FILE_READ and MPI_FILE_WRITE.

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee that data has been transferred to the storage device.

Positioning

MPI provides three types of positioning for data access routines: explicit offsets, individual file pointers, and shared file pointers. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain _AT in their name (e.g., MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position given directly as an argument—no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 9.4.2.

The names of the individual file pointer routines contain no positional qualifier (e.g., MPI_FILE_WRITE). Operations with individual file pointers are described in Section 9.4.3. The data access routines that use shared file pointers contain _SHARED or _ORDERED in their name (e.g., MPI_FILE_WRITE_SHARED). Operations with shared file pointers are described in Section 9.4.4.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

where *count* is the number of *datatype* items to be accessed, $elements(X)$ is the number of predefined datatypes in the typemap of X, and *old_file_offset* is the value of the implicit offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes relative to the current view.

Synchronism

MPI supports blocking and nonblocking I/O routines.

A *blocking* I/O call will not return until the I/O request is completed.

A *nonblocking* I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed concurrently with computation. A separate *request complete* call (MPI_WAIT, MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 9.4.5).

Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 9.6.4, for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI-1 communication functions; see Section 3.12.5 in [6]. The data is accessed from those parts of the file specified by the current view (Section 9.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

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 2, . (*End of advice to users.*)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS, respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. In C++, the status argument is optional. The status can be passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

9.4.2 Data Access with Explicit Offsets

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section.

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
              MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
              const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
              const MPI::Datatype& datatype)
```

MPI_FILE_READ_AT reads a file beginning at the position specified by offset.

MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
              const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
              const MPI::Datatype& datatype)
```

MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface.

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.

MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_AT_ALL is a collective version of the blocking MPI_FILE_WRITE_AT interface.

MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype)
```

MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.

9.4.3 Data Access with Individual File Pointers

MPI maintains one individual file pointer per process per file handle. The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the individual file pointers maintained by MPI. The shared file pointer is not used nor updated.

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 9.4.2, with the following modification:

- the offset is defined to be the current value of the MPI-maintained individual file pointer.

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section.

MPI_FILE_READ(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)
```

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
             MPI::Status& status)
```

```
void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype)
```

MPI_FILE_READ reads a file using the individual file pointer.

Example 9.2 The following Fortran code fragment is an example of reading a file until the end of file is reached:

```
!   Read a preexisting input file until all data has been read.
!   Call routine "process_input" if all requested data is read.
!   The Fortran 90 "exit" statement exits the loop.
```

```
      integer   bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
      parameter (bufsize=100)
      real      localbuffer(bufsize)

      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
      call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                          MPI_INFO_NULL, ierr )
      totprocessed = 0
      do
         call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
                             status, ierr )
         call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
         call process_input( localbuffer, numread )
         totprocessed = totprocessed + numread
         if ( numread < bufsize ) exit
      enddo

      write(6,1001) numread, bufsize, totprocessed
1001  format( "No more data:  read", I3, "and expected", I3, &
              "Processed total of", I6, "before terminating job." )

      call MPI_FILE_CLOSE( myfh, ierr )
```

MPI_FILE_READ_ALL(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.

MPI_FILE_WRITE(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Status *status)
```

```
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write(const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_WRITE writes a file using the individual file pointer.

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write_all(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write_all(const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE interface.

MPI_FILE_IREAD(fh, buf, count, datatype, request)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             MPI_Request *request)
```

```
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
MPI::Request MPI::File::Iread(void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.

Example 9.3 The following Fortran code fragment illustrates file pointer update semantics:

```
!   Read the first twenty real words in a file into two local
!   buffers.  Note that when the first MPI_FILE_IREAD returns,
!   the file pointer has been updated to point to the
!   eleventh real word in the file.

      integer   bufsize, req1, req2
      integer, dimension(MPI_STATUS_SIZE) :: status1, status2
      parameter (bufsize=10)
      real      buf1(bufsize), buf2(bufsize)

      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
      call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                          MPI_INFO_NULL, ierr )
      call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
                           req1, ierr )
      call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
                           req2, ierr )

      call MPI_WAIT( req1, status1, ierr )
      call MPI_WAIT( req2, status2, ierr )

      call MPI_FILE_CLOSE( myfh, ierr )
```

MPI_FILE_IWRITE(fh, buf, count, datatype, request)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
MPI::Request MPI::File::Iwrite(const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.

MPI_FILE_SEEK(fh, offset, whence)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (state)

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

```
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Seek(MPI::Offset offset, int whence)
```

MPI_FILE_SEEK updates the individual file pointer according to whence, which has the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of file plus offset

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

MPI_FILE_GET_POSITION(fh, offset)

IN	fh	file handle (handle)
OUT	offset	offset of individual pointer (integer)

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

```
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
MPI::Offset MPI::File::Get_position() const
```

MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file pointer in etype units relative to the current view.

> *Advice to users.* The offset can be used in a future call to MPI_FILE_SEEK using whence = MPI_SEEK_SET to return to the current position. To set the displacement to the current file pointer position, first convert offset into an absolute byte position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting displacement. (*End of advice to users.*)

MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

IN	fh	file handle (handle)
IN	offset	offset (integer)
OUT	disp	absolute byte position of offset (integer)

```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
              MPI_Offset *disp)
```

```
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
```

```
MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const
```

MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of offset relative to the current view of fh is returned in disp.

9.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the shared file pointer maintained by MPI. The individual file pointers are not used nor updated.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 9.4.2, with the following modifications:

- the offset is defined to be the current value of the MPI-maintained shared file pointer,
- the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and
- the use of shared file pointer routines is erroneous unless all processes use the same file view.

For the noncollective shared file pointer routines, the serialization ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After a shared file pointer operation is initiated, the shared file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

Noncollective Operations

MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_shared(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Read_shared(void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_READ_SHARED reads a file using the shared file pointer.

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
```

```
            MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Write_shared(const void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_shared(const void* buf, int count,
            const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.

MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI::Request MPI::File::Iread_shared(void* buf, int count,
            const MPI::Datatype& datatype)
```

MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED interface.

MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
```

```
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_IWRITE_SHARED is a nonblocking version of the MPI_FILE_WRITE_SHARED interface.

Collective Operations

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

> *Advice to users.* There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

> *Advice to implementors.* Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_ordered(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Read_ordered(void* buf, int count,
            const MPI::Datatype& datatype)
```

MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED interface.

MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	status	status object (Status)

```
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write_ordered(const void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write_ordered(const void* buf, int count,
            const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED interface.

Seek

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the following two routines (MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED).

MPI_FILE_SEEK_SHARED(fh, offset, whence)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (state)

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
```

```
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Seek_shared(MPI::Offset offset, int whence)
```

MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which has the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of file plus offset

MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values for offset and whence.

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

MPI_FILE_GET_POSITION_SHARED(fh, offset)

IN	fh	file handle (handle)
OUT	offset	offset of shared pointer (integer)

```
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
```

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
MPI::Offset MPI::File::Get_position_shared() const
```

MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the shared file pointer in etype units relative to the current view.

> *Advice to users.* The offset can be used in a future call to MPI_FILE_SEEK_SHARED using whence = MPI_SEEK_SET to return to the current position. To set the displacement to the current file pointer position, first convert offset into an absolute byte position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with the resulting displacement. (*End of advice to users.*)

9.4.5 Split Collective Data Access Routines

MPI provides a restricted form of "nonblocking collective" I/O operations for all data accesses using split collective data access routines. These routines are referred to as "split" collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid many (though not all) of the problems described in "A Problem with Register Optimization," Section 2.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

  ```
  MPI_File_read_all_begin(fh, ...);
  ...
  MPI_File_read_all(fh, ...);
  ...
  MPI_File_read_all_end(fh, ...);
  ```

  is erroneous.
- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL).

For the purpose of consistency semantics (Section 9.6.1), a matched pair of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype)
```

```
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_READ_AT_ALL_END(fh, buf, status)

IN	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_at_all_end(void* buf, MPI::Status& status)
```

```
void MPI::File::Read_at_all_end(void* buf)
```

MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype)
```

```
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
              int count, const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status)
```

```
void MPI::File::Write_at_all_end(const void* buf)
```

MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype)
```

```
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
```

```
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
```

```
void MPI::File::Read_all_begin(void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_READ_ALL_END(fh, buf, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_all_end(void* buf, MPI::Status& status)
```

```
void MPI::File::Read_all_end(void* buf)
```

MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype)
```

```
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
```

```
void MPI::File::Write_all_begin(const void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_ALL_END(fh, buf, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write_all_end(const void* buf, MPI::Status& status)
```

```
void MPI::File::Write_all_end(const void* buf)
```

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype)
```

```
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
```

```
void MPI::File::Read_ordered_begin(void* buf, int count,
             const MPI::Datatype& datatype)
```

MPI_FILE_READ_ORDERED_END(fh, buf, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Read_ordered_end(void* buf, MPI::Status& status)
```

```
void MPI::File::Read_ordered_end(void* buf)
```

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype)
```

```
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
```

```
void MPI::File::Write_ordered_begin(const void* buf, int count,
              const MPI::Datatype& datatype)
```

MPI_FILE_WRITE_ORDERED_END(fh, buf, status)

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
```

```
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status)
```

```
void MPI::File::Write_ordered_end(const void* buf)
```

9.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 9.5.2) as well as the data conversion functions (Section 9.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 9.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations similar to POSIX `cp`, `rm`, and `mv` can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: "native," "internal," and "external32." An implementation may support additional data representations. MPI also supports user-defined data representations (see Section 9.5.3). The native and internal data representations are implementation dependent, while the external32 representation is common to all MPI implementations and facilitates file interoperability. The data representation is specified in the *datarep* argument to MPI_FILE_SET_VIEW.

> *Advice to users.* MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)

"native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

> *Advice to users.* This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)

> *Advice to implementors.* When implementing read and write operations on top of MPI message passing, the message data should be typed as MPI_BYTE to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

"internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation.

> *Rationale.* This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (*End of rationale.*)

> *Advice to implementors.* Since "external32" is a superset of the functionality provided by "internal," an implementation may choose to implement "internal" as "external32." (*End of advice to implementors.*)

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 9.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2, page 25-27, of the MPI-1 document). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

> *Advice to implementors.* When implementing read and write operations on top of MPI message passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

9.5.1 Datatypes for File Interoperability

If the file data representation is other than "native," care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file.

For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

> *Advice to users.* One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The routine MPI_FILE_GET_FILE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user defined data representations. Otherwise, the etype and filetype must be constructed so that their typemap and extent are the same on any architecture. This can be achieved if the they have an explicit upper bound and lower bound (defined either using MPI_LB and MPI_UB markers, or using MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype that is used in the construction of the etype and filetype, if this datatype is replicated contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implictly, by a blocklength argument that is greater than one. If an etype or filetype is not portable, and has a typemap or extent that is architecture dependent, then the data layout specified by it on a file is implementation dependent.
>
> File data representations other than "native" may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an etype built from MPI_INT and another uses an etype built from MPI_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (*End of advice to users.*)

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

IN	fh	file handle (handle)
IN	datatype	datatype (handle)
OUT	extent	datatype extent (integer)

```
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
              MPI_Aint *extent)
```

```
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
```

```
MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const
```

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 9.5.3), MPI uses the dtype_file_extent_fn callback to calculate the extent.

> *Advice to implementors.* In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype_file_extent_fn (see Section 9.5.3). (*End of advice to implementors.*)

9.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI_INTEGER2) is not required.

All floating point values are in big-endian IEEE format [9] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For Fortran LOGICAL and C++ bool, 0 implies false and nonzero implies true. Fortran COMPLEX and DOUBLE COMPLEX are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [10]. Wide characters (of type MPI_WCHAR) are in Unicode format [23].

All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [9], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

> *Advice to implementors.* The MPI treatment of "NaN" is similar to the approach used in XDR (see ftp://ds.internic.net/rfc/rfc1832.txt). (*End of advice to implementors.*)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file.

> *Advice to implementors.* All bytes of LOGICAL and bool must be checked to determine the value. (*End of advice to implementors.*)

> *Advice to users.* The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

```
Type                  Length
------------------    ------
MPI_PACKED               1
MPI_BYTE                 1
MPI_CHAR                 1
MPI_UNSIGNED_CHAR        1
MPI_SIGNED_CHAR          1
MPI_WCHAR                2
MPI_SHORT                2
MPI_UNSIGNED_SHORT       2
MPI_INT                  4
MPI_UNSIGNED             4
MPI_LONG                 4
MPI_UNSIGNED_LONG        4
MPI_FLOAT                4
MPI_DOUBLE               8
MPI_LONG_DOUBLE         16

MPI_CHARACTER            1
MPI_LOGICAL              4
MPI_INTEGER              4
MPI_REAL                 4
MPI_DOUBLE_PRECISION     8
MPI_COMPLEX            2*4
MPI_DOUBLE_COMPLEX     2*8

Optional Type         Length
------------------    ------
MPI_INTEGER1             1
MPI_INTEGER2             2
MPI_INTEGER4             4
MPI_INTEGER8             8
MPI_LONG_LONG            8
MPI_UNSIGNED_LONG_LONG   8

MPI_REAL4                4
MPI_REAL8                8
MPI_REAL16              16
```

The size of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 10.2.5.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (*End of advice to implementors.*)

9.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state)

IN	datarep	data representation identifier (string)
IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)
IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)
IN	extra_state	extra state

```
int MPI_Register_datarep(char *datarep,
             MPI_Datarep_conversion_function *read_conversion_fn,
             MPI_Datarep_conversion_function *write_conversion_fn,
             MPI_Datarep_extent_function *dtype_file_extent_fn,
             void *extra_state)
```

```
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
             DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

```
void MPI::Register_datarep(const char* datarep,
             MPI::Datarep_conversion_function* read_conversion_fn,
             MPI::Datarep_conversion_function* write_conversion_fn,
             MPI::Datarep_extent_function* dtype_file_extent_fn,
             void* extra_state)
```

The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn with the data representation identifier datarep. datarep can then be used as an argument to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. MPI_REGISTER_DATAREP is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler (see Section 9.7). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

Extent Callback

```
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
             MPI_Aint *file_extent, void *extra_state);
```

```
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
```

```
typedef MPI::Datarep_extent_function(const MPI::Datatype& datatype,
             MPI::Aint& file_extent, void* extra_state);
```

The function dtype_file_extent_fn must return, in file_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

Datarep Conversion Functions

```
typedef int MPI_Datarep_conversion_function(void *userbuf,
             MPI_Datatype datatype, int count, void *filebuf,
             MPI_Offset position, void *extra_state);
```

```
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
             POSITION, EXTRA_STATE, IERROR)
    <TYPE> USERBUF(*), FILEBUF(*)
    INTEGER COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) POSITION
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
typedef MPI::Datarep_conversion_function(void* userbuf,
             MPI::Datatype& datatype, int count, void* filebuf,
             MPI::Offset position, void* extra_state);
```

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read or write function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI_PACK and MPI_UNPACK in MPI-1, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

1. Get file extent of all data items
2. Allocate a filebuf large enough to hold all count data items
3. Read data from file into filebuf
4. Call read_conversion_fn to convert data and place it into userbuf
5. Deallocate filebuf

(*End of advice to implementors.*)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the position to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the read or write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 9.4, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

9.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 9.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 10.2.5).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatiblity with another implementation's "native" or "internal" representation.

Advice to users. Section 10.2.5, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (*End of advice to users.*)

9.6 Consistency and Semantics

9.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. User-imposed consistency may be obtained using program order and calls to MPI_FILE_SYNC.

Let FH_1 be the set of file handles created from one particular collective open of the file FOO, and FH_2 be the set of file handles created from a different collective open of FOO. Note that nothing restrictive is said about FH_1 and FH_2: the sizes of FH_1 and FH_2 may be different, the groups of processes used for each open may or may not intersect, the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 9.4.5) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

Advice to users. For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (*End of advice to users.*)

Assume that A_1 and A_2 are two data access operations. Let D_1 (D_2) be the set of absolute byte displacements of every byte accessed in A_1 (A_2). The two data accesses *overlap* if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) SEQ_{fh} is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, SEQ_1 and SEQ_2, we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a}, and A_2 is a data access operation using fh_{1b}. If for any access A_1, there is no access A_2 that conflicts with A_1, then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with SEQ_1. To guarantee sequential consistency when there are write sequences, MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 9.6.10, for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY(fh, flag)

INOUT	fh	file handle (handle)
IN	flag	true to set atomic mode, false to set nonatomic mode (logical)

```
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

```
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
```

```
void MPI::File::Set_atomicity(bool flag)
```

Let FH be the set of file handles created by one collective open. The consistency semantics for data access operations using FH is set by collectively calling MPI_FILE_SET_ATOMICITY on FH. MPI_FILE_SET_ATOMICITY is collective; all processes in the group must pass identical values for fh and flag. If flag is true, atomic mode is set; if flag is false, nonatomic mode is set.

Changing the consistency semantics for an open file only affects new data accesses. All completed data accesses are guaranteed to abide by the consistency semantics in effect during their execution. Nonblocking data accesses and split collective operations that have not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode consistency semantics.

> *Advice to implementors.* Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (*End of advice to implementors.*)

MPI_FILE_GET_ATOMICITY(fh, flag)

IN	fh	file handle (handle)
OUT	flag	true if atomic mode, false if nonatomic mode (logical)

```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

```
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
```

```
bool MPI::File::Get_atomicity() const
```

MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.

MPI_FILE_SYNC(fh)

INOUT	fh	file handle (handle)

```
int MPI_File_sync(MPI_File fh)
```

```
MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
```

```
void MPI::File::Sync()
```

Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process. MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see above).

MPI_FILE_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SYNC—otherwise, the call to MPI_FILE_SYNC is erroneous.

9.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

> *Rationale.* This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. (*End of rationale.*)

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) followed by the write.

9.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

9.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in MPI-1 [6], Section 4.12.

Collective file operations are collective over a dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

9.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

9.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

9.6.7 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer of kind MPI_OFFSET_KIND, defined in mpif.h and the mpi module.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an `INTEGER` of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 4.12).

9.6.8 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via *info* when a file is created (see Section 9.2.8).

9.6.9 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.

When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).

> *Advice to users.* Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 9.6.1, are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

> *Advice to users.* Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

9.6.10 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of `b` will be `5`. If nonatomic mode is set, the results of the read are undefined.

```
/* Process 0 */
int  i, a[10] ;
int  TRUE = 1;

for ( i=0;i<10;i++)
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh0, TRUE ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
```

```
/* Process 1 */
int  b[10] ;
int  TRUE = 1;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh1, TRUE ) ;
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;
```

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing temporal order with, for example, calls to MPI_BARRIER.

Advice to users. Routines other than MPI_BARRIER may be used to impose temporal order. In the example above, process 0 could use MPI_SEND to send a 0 byte message, received by process 1 using MPI_RECV. (*End of advice to users.*)

Alternatively, a user can impose consistency with nonatomic mode set:

```
/* Process 0 */
int  i, a[10] ;
for ( i=0;i<10;i++)
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh0 ) ;
```

```
/* Process 1 */
int  b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_sync( fh1 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
```

The “sync-barrier-sync” construct is required because:

- The barrier ensures that the write on process 0 occurs before the read on process 1.
- The first sync guarantees that the data written by all processes is transferred to the storage device.
- The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by eliminating the apparently superfluous second "sync" call for each process.

```
/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
/* Process 0 */
int  i, a[10] ;
for ( i=0;i<10;i++)
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;

/* Process 1 */
int  b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI_FILE_SYNC blocks.

> *Advice to users.* Some implementations may choose to implement MPI_FILE_SYNC as a temporally synchronizing function. When using such an implementation, the "sync-barrier-sync" construct above can be replaced by a single "sync." The results of using such code with an implementation for which MPI_FILE_SYNC is not temporally synchronizing is undefined. (*End of advice to users.*)

Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

```
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ;   Use this to set atomic mode. */
```

```
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Waitall(2, reqs, statuses) ;
```

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ;   Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

defines the same ordering as:

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;
MPI_File_read_at(fh,  10, &b, 1, MPI_INT, &status ) ;
```

Since

- nonconcurrent operations on a single file handle are sequentially consistent, and
- the program fragments specify an order for the operations,

MPI guarantees that both program fragments will read the value 4 into b. There is no need to set atomic mode for this example.

Similar considerations apply to conflicting accesses of the form:

```
MPI_File_write_all_begin(fh,...) ;
MPI_File_iread(fh,...) ;
MPI_Wait(fh,...) ;
MPI_File_write_all_end(fh,...) ;
```

Recall that constraints governing consistency and semantics are not relevant to the following:

```
MPI_File_write_all_begin(fh,...) ;
MPI_File_read_all_begin(fh,...) ;
MPI_File_read_all_end(fh,...) ;
MPI_File_write_all_end(fh,...) ;
```

since split collective operations on the same file handle may not overlap (see Section 9.4.5).

9.7 I/O Error Handling

By default, communication errors are fatal—MPI_ERRORS_ARE_FATAL is the default error handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g., "file not found") than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

> *Advice to users.* MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (*End of advice to users.*)

Like communicators, each file handle has an error handler associated with it. The MPI-2 I/O error handling routines are defined in Section 4.13.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL,

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

9.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the following error classes. In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

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

9.9 Examples

9.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function `compute_buffer()`.

```
/*=========================================================================
 *
 * Function:            double_buffer
```

```
 *
 * Synopsis:
 *     void double_buffer(
 *             MPI_File fh,                            ** IN
 *             MPI_Datatype buftype,                   ** IN
 *             int bufcount                            ** IN
 *     )
 *
 * Description:
 *     Performs the steps to overlap computation with a collective write
 *     by using a double-buffering technique.
 *
 * Parameters:
 *     fh                previously opened MPI file handle
 *     buftype           MPI datatype for memory layout
 *                       (Assumes a compatible view has been set on fh)
 *     bufcount          # buftype elements to transfer
 *------------------------------------------------------------------------*/

/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))

void double_buffer(  MPI_File fh, MPI_Datatype buftype, int bufcount)
{

   MPI_Status status;         /* status for MPI calls */
   float *buffer1, *buffer2;  /* buffers to hold results */
   float *compute_buf_ptr;    /* destination  buffer */
                              /*   for computing */
   float *write_buf_ptr;      /* source for writing */
   int done;                  /* determines when to quit */

   /* buffer initialization */
   buffer1 = (float *)
                   malloc(bufcount*sizeof(float)) ;
   buffer2 = (float *)
                   malloc(bufcount*sizeof(float)) ;
   compute_buf_ptr = buffer1 ;   /* initially point to buffer1 */
   write_buf_ptr   = buffer1 ;   /* initially point to buffer1 */


   /* DOUBLE-BUFFER prolog:
    *   compute buffer1; then initiate writing buffer1 to disk
    */
   compute_buffer(compute_buf_ptr, bufcount, &done);
   MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);

   /* DOUBLE-BUFFER steady state:
```

```
     *  Overlap writing old results from buffer pointed to by write_buf_ptr
     *  with computing new results into buffer pointed to by compute_buf_ptr.
     *
     *  There is always one write-buffer and one compute-buffer in use
     *  during steady state.
     */
    while (!done) {
       TOGGLE_PTR(compute_buf_ptr);
       compute_buffer(compute_buf_ptr, bufcount, &done);
       MPI_File_write_all_end(fh, write_buf_ptr, &status);
       TOGGLE_PTR(write_buf_ptr);
       MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
    }

    /* DOUBLE-BUFFER epilog:
     *   wait for final write to complete.
     */
    MPI_File_write_all_end(fh, write_buf_ptr, &status);


    /* buffer cleanup */
    free(buffer1);
    free(buffer2);
 }
```

9.9.2 Subarray Filetype Constructor

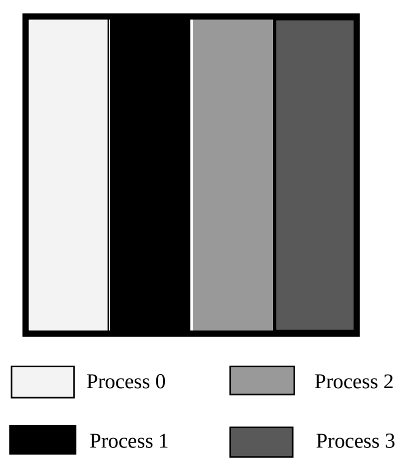

Figure 9.4: Example array file layout

Assume we are writing out a 100x100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0-24, process 1 has columns 25-49, etc.; see Figure 9.4). To create the filetypes for each process one could use the following C program:

```
    double subarray[100][25];
```

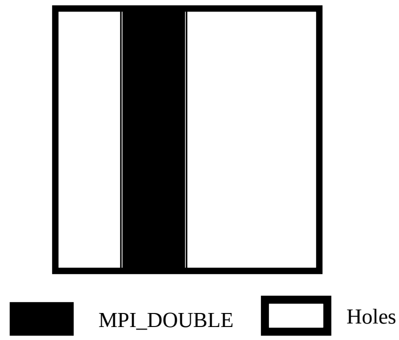

Figure 9.5: Example local array filetype for process 1

```
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &filetype);
```

Or, equivalently in Fortran:

```
double precision subarray(100,25)
integer filetype, rank, ierror
integer sizes(2), subsizes(2), starts(2)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
sizes(1)=100
sizes(2)=100
subsizes(1)=100
subsizes(2)=25
starts(1)=0
starts(2)=rank*subsizes(2)

call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
           MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,       &
           filetype, ierror)
```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 9.5 shows the filetype created for process 1.

Chapter 10

Language Bindings

10.1 C++

10.1.1 Overview

This section presents a complete C++ language interface for MPI. There are some issues specific to C++ that must be considered in the design of this interface that go beyond the simple description of language bindings. In particular, in C++, we must be concerned with the design of objects and their interfaces, rather than just the design of a language-specific functional interface to MPI. Fortunately, the original design of MPI was based on the notion of objects, so a natural set of classes is already part of MPI.

Since the original design of MPI-1 did not include a C++ language interface, a complete list of C++ bindings for MPI-1 functions is provided in Annex B. MPI-2 includes C++ bindings as part of its function specifications. In some cases, MPI-2 provides new names for the C bindings of MPI-1 functions. In this case, the C++ binding matches the new C name — there is no binding for the deprecated name. As such, the C++ binding for the new name appears in Annex A, not Annex B.

10.1.2 Design

The C++ language interface for MPI is designed according to the following criteria:

1. The C++ language interface consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g., communicator, group, etc.).

2. The MPI C++ language bindings provide a semantically correct interface to MPI.

3. To the greatest extent possible, the C++ bindings for MPI functions are member functions of MPI classes.

   Rationale. Providing a lightweight set of MPI objects that correspond to the basic MPI types is the best fit to MPI's implicit object-based design; methods can be supplied for these objects to realize MPI functionality. The existing C bindings can be used in C++ programs, but much of the expressive power of the C++ language is forfeited. On the other hand, while a comprehensive class library would make user programming more elegant, such a library it is not suitable as a language binding for MPI since a binding must provide a direct and unambiguous mapping to the specified functionality of MPI. (*End of rationale.*)

.

10.1.3 C++ Classes for MPI

All MPI classes, constants, and functions are declared within the scope of an `MPI` `namespace`. Thus, instead of the `MPI_` prefix that is used in C and Fortran, MPI functions essentially have an `MPI::` prefix.

> *Advice to implementors.* Although `namespace` is officially part of the draft ANSI C++ standard, as of this writing it not yet widely implemented in C++ compilers. Implementations using compilers without `namespace` may obtain the same scoping through the use of a non-instantiable `MPI` class. (To make the `MPI` class non-instantiable, all constructors must be `private`.) (*End of advice to implementors.*)

The members of the `MPI` namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the `MPI` namespace for MPI-1 and its member classes is as follows:

```
namespace MPI {
  class Comm                             {...};
  class Intracomm : public Comm          {...};
  class Graphcomm : public Intracomm     {...};
  class Cartcomm  : public Intracomm     {...};
  class Intercomm : public Comm          {...};
  class Datatype                         {...};
  class Errhandler                       {...};
  class Exception                        {...};
  class Group                            {...};
  class Op                               {...};
  class Request                          {...};
  class Prequest  : public Request       {...};
  class Status                           {...};
};
```

Additionally, the following classes defined for MPI-2:

```
namespace MPI {
  class File                             {...};
  class Grequest  : public Request       {...};
  class Info                             {...};
  class Win                              {...};
};
```

Note that there are a small number of derived classes, and that virtual inheritance is *not* used.

10.1.4 Class Member Functions for MPI

Besides the member functions which constitute the C++ language bindings for MPI, the C++ language interface has additional functions (as required by the C++ language). In particular, the C++ language interface must provide a constructor and destructor, an assignment operator, and comparison operators.

The complete set of C++ language bindings for MPI-1 is presented in Annex B. The bindings take advantage of some important C++ features, such as references and `const`. Declarations (which apply to all `MPI` member classes) for construction, destruction, copying, assignment, comparison, and mixed-language operability are also provided. To maintain consistency with what has gone before, the binding definitions are given in the same order as given for the C bindings in [6].

Except where indicated, all non-static member functions (except for constructors and the assignment operator) of `MPI` member classes are virtual functions.

> *Rationale.* Providing virtual member functions is an important part of design for inheritance. Virtual functions can be bound at run-time, which allows users of libraries to re-define the behavior of objects already contained in a library. There is a small performance penalty that must be paid (the virtual function must be looked up before it can be called). However, users concerned about this performance penalty can force compile-time function binding. (*End of rationale.*)

Example 10.1 Example showing a derived MPI class.

```
class foo_comm : public MPI::Intracomm {
public:
  void Send(void* buf, int count, const MPI::Datatype& type,
            int dest, int tag) const
  {
    // Class library functionality
    MPI::Intracomm::Send(buf, count, type, dest, tag);
    // More class library functionality
  }
};
```

> *Advice to implementors.* Implementors must be careful to avoid unintended side effects from class libraries that use inheritance, especially in layered implementations. For example, if MPI_BCAST is implemented by repeated calls to MPI_SEND or MPI_RECV, the behavior of MPI_BCAST cannot be changed by derived communicator classes that might redefine MPI_SEND or MPI_RECV. The implementation of MPI_BCAST must explicitly use the MPI_SEND (or MPI_RECV) of the base `MPI::Comm` class. (*End of advice to implementors.*)

10.1.5 Semantics

The semantics of the member functions constituting the C++ language binding for MPI are specified by the MPI function description itself. Here, we specify the semantics for those portions of the C++ language interface that are not part of the language binding. In this subsection, functions are prototyped using the type `MPI::⟨CLASS⟩` rather than listing each function for every MPI class; the word `⟨CLASS⟩` can be replaced with any valid MPI class name (e.g., `Group`), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```
MPI::<CLASS>()
```

```
~MPI::<CLASS>()
```

In terms of construction and destruction, opaque MPI user level objects behave like handles. Default constructors for all MPI objects except MPI::Status create corresponding MPI::*_NULL handles. That is, when an MPI object is instantiated, comparing it with its corresponding MPI::*_NULL object will return `true`. The default constructors do not create new MPI opaque objects. Some classes have a member function `Create()` for this purpose.

Example 10.2 In the following code fragment, the test will return `true` and the message will be sent to `cout`.

```
void foo()
{
  MPI::Intracomm bar;

  if (bar == MPI::COMM_NULL)
    cout << "bar is MPI::COMM_NULL" << endl;
}
```

The destructor for each MPI user level object does *not* invoke the corresponding MPI_*_FREE function (if it exists).

> *Rationale.* MPI_*_FREE functions are not automatically invoked for the following reasons:
>
> 1. Automatic destruction contradicts the shallow-copy semantics of the MPI classes.
> 2. The model put forth in MPI makes memory allocation and deallocation the responsibility of the user, not the implementation.
> 3. Calling MPI_*_FREE upon destruction could have unintended side effects, including triggering collective operations (this also affects the copy, assignment, and construction semantics). In the following example, we would want neither `foo_comm` nor `bar_comm` to automatically invoke MPI_*_FREE upon exit from the function.
>
> ```
> void example_function()
> {
>   MPI::Intracomm foo_comm(MPI::COMM_WORLD), bar_comm;
>   bar_comm = MPI::COMM_WORLD.Dup();
>   // rest of function
> }
> ```
>
> (*End of rationale.*)

Copy / Assignment The copy constructor and assignment operator are prototyped as follows:

```
MPI::<CLASS>(const MPI::<CLASS>& data)

MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI::<CLASS>& data)
```

In terms of copying and assignment, opaque MPI user level objects behave like handles. Copy constructors perform handle-based (shallow) copies. `MPI::Status` objects are exceptions to this rule. These objects perform deep copies for assignment and copy construction.

Advice to implementors. Each MPI user level object is likely to contain, by value or by reference, implementation-dependent state information. The assignment and copying of MPI object handles may simply copy this value (or reference). (*End of advice to implementors.*)

Example 10.3 Example using assignment operator. In this example, `MPI::Intracomm::Dup()` is *not* called for `foo_comm`. The object `foo_comm` is simply an alias for `MPI::COMM_WORLD`. But `bar_comm` is created with a call to `MPI::Intracomm::Dup()` and is therefore a different communicator than `foo_comm` (and thus different from `MPI::COMM_WORLD`). `baz_comm` becomes an alias for `bar_comm`. If one of `bar_comm` or `baz_comm` is freed with MPI_COMM_FREE it will be set to MPI::COMM_NULL. The state of the other handle will be undefined — it will be invalid, but not necessarily set to MPI::COMM_NULL.

```
MPI::Intracomm foo_comm, bar_comm, baz_comm;

foo_comm = MPI::COMM_WORLD;
bar_comm = MPI::COMM_WORLD.Dup();
baz_comm = bar_comm;
```

Comparison The comparison operators are prototyped as follows:

```
bool MPI::<CLASS>::operator==(const MPI::<CLASS>& data) const

bool MPI::<CLASS>::operator!=(const MPI::<CLASS>& data) const
```

The member function `operator==()` returns `true` only when the handles reference the same internal MPI object, `false` otherwise. `operator!=()` returns the boolean complement of `operator==()`. However, since the `Status` class is not a handle to an underlying MPI object, it does not make sense to compare `Status` instances. Therefore, the `operator==()` and `operator!=()` functions are not defined on the `Status` class.

Constants Constants are singleton objects and are declared `const`. Note that not all globally defined MPI objects are constant. For example, `MPI::COMM_WORLD` and `MPI::COMM_SELF` are not `const`.

10.1.6 C++ Datatypes

Table 10.1 lists all of the C++ predefined MPI datatypes and their corresponding C and C++ datatypes, Table 10.2 lists all of the Fortran predefined MPI datatypes and their corresponding Fortran 77 datatypes. Table 10.3 lists the C++ names for all other MPI datatypes.

MPI::BYTE and MPI::PACKED conform to the same restrictions as MPI_BYTE and MPI_PACKED, listed in Sections 3.2.2 and 3.13 of MPI-1, respectively.

MPI datatype	C datatype	C++ datatype
MPI::CHAR	char	char
MPI::WCHAR	wchar_t	wchar_t
MPI::SHORT	signed short	signed short
MPI::INT	signed int	signed int
MPI::LONG	signed long	signed long
MPI::SIGNED_CHAR	signed char	signed char
MPI::UNSIGNED_CHAR	unsigned char	unsigned char
MPI::UNSIGNED_SHORT	unsigned short	unsigned short
MPI::UNSIGNED	unsigned int	unsigned int
MPI::UNSIGNED_LONG	unsigned long	unsigned long int
MPI::FLOAT	float	float
MPI::DOUBLE	double	double
MPI::LONG_DOUBLE	long double	long double
MPI::BOOL		bool
MPI::COMPLEX		Complex<float>
MPI::DOUBLE_COMPLEX		Complex<double>
MPI::LONG_DOUBLE_COMPLEX		Complex<long double>
MPI::BYTE		
MPI::PACKED		

Table 10.1: C++ names for the MPI C and C++ predefined datatypes, and their corresponding C/C++ datatypes.

The following table defines groups of MPI predefined datatypes:

C integer:	MPI::INT, MPI::LONG, MPI::SHORT, MPI::UNSIGNED_SHORT, MPI::UNSIGNED, MPI::UNSIGNED_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, MPI::BOOL
Complex:	MPI::F_COMPLEX, MPI::COMPLEX, MPI::F_DOUBLE_COMPLEX, MPI::DOUBLE_COMPLEX, MPI::LONG_DOUBLE_COMPLEX
Byte:	MPI::BYTE

Valid datatypes for each reduction operation is specified below in terms of the groups defined above.

MPI datatype	Fortran datatype
MPI::CHARACTER	CHARACTER(1)
MPI::INTEGER	INTEGER
MPI::REAL	REAL
MPI::DOUBLE_PRECISION	DOUBLE PRECISION
MPI::LOGICAL	LOGICAL
MPI::F_COMPLEX	COMPLEX
MPI::BYTE	
MPI::PACKED	

Table 10.2: C++ names for the MPI Fortran predefined datatypes, and their corresponding Fortran 77 datatypes.

MPI datatype	Description
MPI::FLOAT_INT	C/C++ reduction type
MPI::DOUBLE_INT	C/C++ reduction type
MPI::LONG_INT	C/C++ reduction type
MPI::TWOINT	C/C++ reduction type
MPI::SHORT_INT	C/C++ reduction type
MPI::LONG_DOUBLE_INT	C/C++ reduction type
MPI::LONG_LONG	Optional C/C++ type
MPI::UNSIGNED_LONG_LONG	Optional C/C++ type
MPI::TWOREAL	Fortran reduction type
MPI::TWODOUBLE_PRECISION	Fortran reduction type
MPI::TWOINTEGER	Fortran reduction type
MPI::F_DOUBLE_COMPLEX	Optional Fortran type
MPI::INTEGER1	Explicit size type
MPI::INTEGER2	Explicit size type
MPI::INTEGER4	Explicit size type
MPI::INTEGER8	Explicit size type
MPI::REAL4	Explicit size type
MPI::REAL8	Explicit size type
MPI::REAL16	Explicit size type

Table 10.3: C++ names for other MPI datatypes. Implementations may also define other optional types (e.g., MPI::INTEGER8).

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

MPI::MINLOC and MPI::MAXLOC perform just as their C and Fortran counterparts; see Section 4.9.3 in MPI-1.

10.1.7 Communicators

The `MPI::Comm` class hierarchy makes explicit the different kinds of communicators implicitly defined by MPI and allows them to be strongly typed. Since the original design of MPI defined only one type of handle for all types of communicators, the following clarifications are provided for the C++ design.

Types of communicators There are five different types of communicators: `MPI::Comm`, `MPI::Intercomm`, `MPI::Intracomm`, `MPI::Cartcomm`, and `MPI::Graphcomm`. `MPI::Comm` is the abstract base communicator class, encapsulating the functionality common to all MPI communicators. `MPI::Intercomm` and `MPI::Intracomm` are derived from `MPI::Comm`. `MPI::Cartcomm` and `MPI::Graphcomm` are derived from `MPI::Intracomm`.

Advice to users. Initializing a derived class with an instance of a base class is not legal in C++. For instance, it is not legal to initialize a Cartcomm from an Intracomm. Moreover, because MPI::Comm is an abstract base class, it is non-instantiable, so that it is not possible to have an object of class MPI::Comm. However, it is possible to have a reference or a pointer to an MPI::Comm.

Example 10.4 The following code is erroneous.

```
Intracomm intra = MPI::COMM_WORLD.Dup();
Cartcomm cart(intra);         // This is erroneous
```

(*End of advice to users.*)

MPI::COMM_NULL The specific type of MPI::COMM_NULL is implementation dependent. MPI::COMM_NULL must be able to be used in comparisons and initializations with all types of communicators. MPI::COMM_NULL must also be able to be passed to a function that expects a communicator argument in the parameter list (provided that MPI::COMM_NULL is an allowed value for the communicator argument).

Rationale. There are several possibilities for implementation of MPI::COMM_NULL. Specifying its required behavior, rather than its realization, provides maximum flexibility to implementors. (*End of rationale.*)

Example 10.5 The following example demonstrates the behavior of assignment and comparison using MPI::COMM_NULL.

```
MPI::Intercomm comm;
comm = MPI::COMM_NULL;            // assign with COMM_NULL
if (comm == MPI::COMM_NULL)       // true
  cout << "comm is NULL" << endl;
if (MPI::COMM_NULL == comm)       // note -- a different function!
  cout << "comm is still NULL" << endl;
```

`Dup()` is not defined as a member function of `MPI::Comm`, but it is defined for the derived classes of `MPI::Comm`. `Dup()` is not virtual and it returns its OUT/ parameter by value.

`MPI::Comm::Clone()` The C++ language interface for MPI includes a new function `Clone()`. `MPI::Comm::Clone()` is a pure virtual function. For the derived communicator classes, `Clone()` behaves like `Dup()` except that it returns a new object by reference. The `Clone()` functions are prototyped as follows:

`Comm& Comm::Clone() const = 0`

`Intracomm& Intracomm::Clone() const`

`Intercomm& Intercomm::Clone() const`

`Cartcomm& Cartcomm::Clone() const`

`Graphcomm& Graphcomm::Clone() const`

Rationale. `Clone()` provides the "virtual dup" functionality that is expected by C++ programmers and library writers. Since `Clone()` returns a new object by reference, users are responsible for eventually deleting the object. A new name is introduced rather than changing the functionality of `Dup()`. (*End of rationale.*)

Advice to implementors. Within their class declarations, prototypes for `Clone()` and `Dup()` would look like the following:

```
namespace MPI {
  class Comm {
     virtual Comm& Clone() const = 0;
  };
  class Intracomm : public Comm {
     Intracomm Dup() const { ... };
     virtual Intracomm& Clone() const { ... };
  };
  class Intercomm : public Comm {
     Intercomm Dup() const { ... };
     virtual Intercomm& Clone() const { ... };
  };
  // Cartcomm and Graphcomm are similarly defined
};
```

Compilers that do not support the variable return type feature of virtual functions may return a reference to `Comm`. Users can cast to the appropriate type as necessary. (*End of advice to implementors.*)

10.1.8 Exceptions

The C++ language interface for MPI includes the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS for use with the Set_errhandler() member functions. MPI::ERRORS_THROW_EXCEPTIONS can only be set or retrieved by C++ functions. If a non-C++ program causes an error that invokes the MPI::ERRORS_THROW_EXCEPTIONS error handler, the exception will pass up the calling stack until C++ code can catch it. If there is no C++ code to catch it, the behavior is undefined. In a multi-threaded environment or if a non-blocking MPI call throws an exception while making progress in the background, the behavior is implementation dependent.

The error handler MPI::ERRORS_THROW_EXCEPTIONS causes an MPI::Exception to be thrown for any MPI result code other than MPI::SUCCESS. The public interface to MPI::Exception class is defined as follows:

```
namespace MPI {
  class Exception {
  public:

    Exception(int error_code);

    int Get_error_code() const;
    int Get_error_class() const;
    const char *Get_error_string() const;
  };
};
```

Advice to implementors.

The exception will be thrown within the body of MPI::ERRORS_THROW_EXCEPTIONS. It is expected that control will be returned to the user when the exception is thrown. Some MPI functions specify certain return information in their parameters in the case of an error and MPI_ERRORS_RETURN is specified. The same type of return information must be provided when exceptions are thrown.

For example, MPI_WAITALL puts an error code for each request in the corresponding entry in the status array and returns MPI_ERR_IN_STATUS. When using MPI::ERRORS_THROW_EXCEPTIONS, it is expected that the error codes in the status array will be set appropriately before the exception is thrown.

(*End of advice to implementors.*)

10.1.9 Mixed-Language Operability

The C++ language interface provides functions listed below for mixed-language operability. These functions provide for a seamless transition between C and C++. For the case where the C++ class corresponding to <CLASS> has derived classes, functions are also provided for converting between the derived classes and the C MPI_<CLASS>.

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

```
MPI::<CLASS>(const MPI_<CLASS>& data)
```

```
MPI::<CLASS>::operator MPI_<CLASS>() const
```

These functions are discussed in Section 4.12.4.

10.1.10 Profiling

This section specifies the requirements of a C++ profiling interface to MPI.

> *Advice to implementors.* Since the main goal of profiling is to intercept function calls from user code, it is the implementor's decision how to layer the underlying implementation to allow function calls to be intercepted and profiled. If an implementation of the MPI C++ bindings is layered on top of MPI bindings in another language (such as C), or if the C++ bindings are layered on top of a profiling interface in another language, no extra profiling interface is necessary because the underlying MPI implementation already meets the MPI profiling interface requirements.
>
> Native C++ MPI implementations that do not have access to other profiling interfaces must implement an interface that meets the requirements outlined in this section.
>
> High quality implementations can implement the interface outlined in this section in order to promote portable C++ profiling libraries. Implementors may wish to provide an option whether to build the C++ profiling interface or not; C++ implementations that are already layered on top of bindings in another language or another profiling interface will have to insert a third layer to implement the C++ profiling interface. (*End of advice to implementors.*)

To meet the requirements of the C++ MPI profiling interface, an implementation of the MPI functions *must*:

1. Provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix "`MPI::`") should also be accessible with the prefix "`PMPI::`."

2. Ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.

3. Document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that profiler developer knows whether they must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

4. Where the implementation of different language bindings is is done through a layered approach (e.g., the C++ binding is a set of "wrapper" functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

   This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the author of the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. Provide a no-op routine MPI::Pcontrol in the MPI library.

Advice to implementors. There are (at least) two apparent options for implementing the C++ profiling interface: inheritance or caching. An inheritance-based approach may not be attractive because it may require a virtual inheritance implementation of the communicator classes. Thus, it is most likely that implementors still cache `PMPI` objects on their corresponding `MPI` objects. The caching scheme is outlined below.

The "real" entry points to each routine can be provided within a `namespace PMPI`. The non-profiling version can then be provided within a `namespace MPI`.

Caching instances of `PMPI` objects in the `MPI` handles provides the "has a" relationship that is necessary to implement the profiling scheme.

Each instance of an `MPI` object simply "wraps up" an instance of a `PMPI` object. `MPI` objects can then perform profiling actions before invoking the corresponding function in their internal `PMPI` object.

The key to making the profiling work by simply re-linking programs is by having a header file that *declares* all the MPI functions. The functions must be *defined* elsewhere, and compiled into a library. MPI constants should be declared `extern` in the `MPI` namespace. For example, the following is an excerpt from a sample `mpi.h` file:

Example 10.6 Sample `mpi.h` file.

```
namespace PMPI {
  class Comm {
  public:
    int Get_size() const;
  };
  // etc.
};

namespace MPI {
public:
  class Comm {
  public:
    int Get_size() const;

  private:
    PMPI::Comm pmpi_comm;
  };
};
```

Note that all constructors, the assignment operator, and the destructor in the `MPI` class will need to initialize/destroy the internal `PMPI` object as appropriate.

The definitions of the functions must be in separate object files; the `PMPI` class member functions and the non-profiling versions of the `MPI` class member functions can be compiled into `libmpi.a`, while the profiling versions can be compiled into `libpmpi.a`. Note that the `PMPI` class member functions and the `MPI` constants must be in different object files than the non-profiling `MPI` class member functions in the `libmpi.a` library to prevent multiple definitions of `MPI` class member function names when linking both `libmpi.a` and `libpmpi.a`. For example:

Example 10.7 `pmpi.cc`, to be compiled into `libmpi.a`.

```
int PMPI::Comm::Get_size() const
{
  // Implementation of MPI_COMM_SIZE
}
```

Example 10.8 `constants.cc`, to be compiled into `libmpi.a`.

```
const MPI::Intracomm MPI::COMM_WORLD;
```

Example 10.9 `mpi_no_profile.cc`, to be compiled into `libmpi.a`.

```
int MPI::Comm::Get_size() const
{
  return pmpi_comm.Get_size();
}
```

Example 10.10 `mpi_profile.cc`, to be compiled into `libpmpi.a`.

```
int MPI::Comm::Get_size() const
{
  // Do profiling stuff
  int ret = pmpi_comm.Get_size();
  // More profiling stuff
  return ret;
}
```

(*End of advice to implementors.*)

10.2 Fortran Support

10.2.1 Overview

Fortran 90 is the current international Fortran standard. MPI-2 Fortran bindings are Fortran 90 bindings that in most cases are "Fortran 77 friendly." That is, with few exceptions (e.g., `KIND`-parameterized types, and the `mpi` module, both of which can be avoided) Fortran 77 compilers should be able to compile MPI programs.

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. MPI does not (yet) use many of these features because of a number of technical difficulties. (*End of rationale.*)

MPI defines two levels of Fortran support, described in Sections 10.2.3 and 10.2.4. A third level of Fortran support is envisioned, but is deferred to future standardization efforts. In the rest of this section, "Fortran" shall refer to Fortran 90 (or its successor) unless qualified.

1. **Basic Fortran Support** An implementation with this level of Fortran support provides the original Fortran bindings specified in MPI-1, with small additional requirements specified in Section 10.2.3.

2. **Extended Fortran Support** An implementation with this level of Fortran support provides Basic Fortran Support plus additional features that specifically support Fortran 90, as described in Section 10.2.4.

A compliant MPI-2 implementation providing a Fortran interface must provide Extended Fortran Support unless the target compiler does not support modules or KIND-parameterized types.

10.2.2 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail. It supersedes and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran 90, not Fortran 77).

The following MPI features are inconsistent with Fortran 90.

1. An MPI subroutine with a choice argument may be called with different argument types.

2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.

3. 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.

4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls.

5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 for more information.

6. The memory allocation routine MPI_ALLOC_MEM can't be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type MPI_Aint and in Fortran of type INTEGER. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND. A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6
for more information.

Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90 is technically only allowed if the function is overloaded with a different function for each type. In C, the use of void* formal arguments avoids these problems.

The following code fragment is technically illegal and may generate a compile-time error.

```
integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning, though there is concern that Fortran 90 compilers are more likely to return errors.

It is also technically illegal in Fortran to pass a scalar actual argument to an array dummy argument. Thus the following code fragment may generate an error since the buf argument to MPI_SEND is declared as an assumed-size array <type> buf(*).

```
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)
```

> *Advice to users.* In the event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling separately, or by using an MPI implementation with Extended Fortran Support as described in Section 10.2.4. An alternative that will usually work with variables local to a routine but not with arguments to a function or subroutine is to use the EQUIVALENCE statement to create another variable with a type accepted by the compiler. (*End of advice to users.*)

Problems Due to Data Copying and Sequence Association

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran 90, user data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, ... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory. Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.[1]

Because MPI dummy buffer arguments are assumed-size arrays, this leads to a serious problem for a non-blocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

Since the first dummy argument to MPI_IRECV is an assumed-size array (<type> buf(*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a 'simple' section such as A(1:N) of such an array. (We define 'simple' more fully in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

Our formal definition of a 'simple' array section is

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. Examples are

```
A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
```

Because of Fortran's column-major ordering, where the first index varies fastest, a simple section of a contiguous array will also be contiguous.[2]

The same problem can occur with a scalar argument. Some compilers, even for Fortran 77, make a copy of some scalar dummy arguments within a called procedure. That this can cause a problem is illustrated by the example

```
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a
```

[1] Technically, the Fortran standards are worded to allow non-contiguous storage of any array data.

[2] To keep the definition of 'simple' simple, we have chosen to require all but one of the section subscripts to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader that the whole of the dimension is selected. It would have been possible to allow cases where the whole dimension is selected with one or two bounds, but this means for the reader that the array declaration or most recent allocation has to be consulted and for the compiler that a run-time check may be required.

```
subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end
```

If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.

Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., `A(1:n:2)`), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.

If there is a compiler option that inhibits copying of arguments, in either the calling or called procedure, this should be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, simple array sections of such arrays, or scalars, and if there is no compiler option to inhibit this, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any non-blocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, including MPI_BOTTOM, MPI_STATUS_IGNORE, MPI_IN_PLACE, MPI_STATUSES_IGNORE and MPI_ERRCODES_IGNORE. In C, these are implemented as constant pointers, usually as `NULL` and are used where the function prototype calls for a pointer to a variable, not the variable itself.

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).

Fortran 90 Derived Types

MPI does not explicitly support passing Fortran 90 derived types to choice dummy arguments. Indeed, for MPI implementations that provide explicit interfaces through the `mpi` module a compiler will reject derived type actual arguments at compile time. Even when no explicit interfaces are given, users should be aware that Fortran 90 provides no guarantee of sequence association for derived types or arrays of derived types. For instance, an array of a derived type consisting of two elements may be implemented as an array of the first elements followed by an array of the second. Use of the `SEQUENCE` attribute may help here, somewhat.

The following code fragment shows one possible way to send a derived type in Fortran. The example assumes that all data is passed by address.

```
    type mytype
       integer i
       real x
       double precision d
    end type mytype

    type(mytype) foo
    integer blocklen(3), type(3)
    integer(MPI_ADDRESS_KIND) disp(3), base

    call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
    call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
    call MPI_GET_ADDRESS(foo%d, disp(3), ierr)

    base = disp(1)
    disp(1) = disp(1) - base
    disp(2) = disp(2) - base
    disp(3) = disp(3) - base

    blocklen(1) = 1
    blocklen(2) = 1
    blocklen(3) = 1

    type(1) = MPI_INTEGER
    type(2) = MPI_REAL
    type(3) = MPI_DOUBLE_PRECISION

    call MPI_TYPE_CREATE_STRUCT(3, blocklen, disp, type, newtype, ierr)
    call MPI_TYPE_COMMIT(newtype, ierr)

! unpleasant to send foo%i instead of foo, but it works for scalar
! entities of type mytype
    call MPI_SEND(foo%i, 1, newtype, ...)
```

A Problem with Register Optimization

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. This section discusses register optimization pitfalls.

When a variable is local to a Fortran subroutine (i.e., not in a module or `COMMON block`), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

Normally users are not afflicted with this. But the user should pay attention to this section if in his/her program a buffer argument to an MPI_SEND, MPI_RECV etc., uses a name which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one mentioned in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

The following example shows what Fortran compilers are allowed to do.

```
This source ...                         can be compiled as:

call MPI_GET_ADDRESS(buf,bufaddr,       call MPI_GET_ADDRESS(buf,...)
               ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,        call MPI_TYPE_CREATE_STRUCT(...)
               bufaddr,
               MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)       call MPI_TYPE_COMMIT(...)
val_old = buf                           register = buf
                                        val_old = register
call MPI_RECV(MPI_BOTTOM,1,type,...)    call MPI_RECV(MPI_BOTTOM,...)
val_new = buf                           val_new = register
```

The compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access of buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

The next example shows extreme, but allowed, possibilities.

```
Source                      compiled as                 or compiled as

call MPI_IRECV(buf,..req)   call MPI_IRECV(buf,..req)   call MPI_IRECV(buf,..req)
                            register = buf              b1 = buf
call MPI_WAIT(req,..)       call MPI_WAIT(req,..)       call MPI_WAIT(req,..)
b1 = buf                    b1 := register
```

MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_IRECV and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has returned, and may schedule the load of buf earlier than typed in the source. It has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as in the case on the right.

To prevent instruction reordering or the allocation of a buffer in a register there are two possibilities in portable Fortran code:

- The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. Note that if the intent is declared in the external subroutine, it must be `OUT` or `INOUT`. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, the above call of MPI_RECV might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

with the separately compiled

```
subroutine DD(buf)
  integer buf
end
```

(assuming that buf has type INTEGER). The compiler may be similarly prevented from moving a reference to a variable across a call to an MPI subroutine.

In the case of a non-blocking call, as in the above call of MPI_WAIT, no reference to the buffer is permitted until it has been verified that the transfer has been completed. Therefore, in this case, the extra call ahead of the MPI call is not necessary, i.e., the call of MPI_WAIT in the example might be replaced by

```
call MPI_WAIT(req,..)
call DD(buf)
```

- An alternative is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure (MPI_RECV in the above example) may alter the buffer or variable, provided that the compiler cannot analyze that the MPI procedure does not reference the module or common block.

In the longer term, the attribute VOLATILE is under consideration for Fortran 2000 and would give the buffer or variable the properties needed, but it would inhibit optimization of any code containing the buffer or variable.

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe.

10.2.3 Basic Fortran Support

Because Fortran 90 is (for all practical purposes) a superset of Fortran 77, Fortran 90 (and future) programs can use the original Fortran interface. The following additional requirements are added:

1. Implementations are required to provide the file mpif.h, as described in the original MPI-1 specification.

2. `mpif.h` must be valid and equivalent for both fixed- and free- source form.

   Advice to implementors. To make `mpif.h` compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended that requirement two be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` contains only declarations, and because common block declarations can be split among several lines. To support Fortran 77 as well as Fortran 90, it may be necessary to eliminate all comments from `mpif.h`. (*End of advice to implementors.*)

10.2.4 Extended Fortran Support

Implementations with Extended Fortran support must provide:

1. An `mpi` module

2. A new set of functions to provide additional support for Fortran intrinsic numeric types, including parameterized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. Parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters. These routines are described in detail in Section 10.2.5.

Additionally, high quality implementations should provide a mechanism to prevent fatal type mismatch errors for MPI routines with choice arguments.

The `mpi` Module

An MPI implementation must provide a module named `mpi` that can be `USE`d in a Fortran 90 program. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.

An MPI implementation may provide in the `mpi` module other features that enhance the usability of MPI while maintaining adherence to the standard. For example, it may:

- Provide interfaces for all or for a subset of MPI routines.
- Provide `INTENT` information in these interface blocks.

  Advice to implementors. The appropriate `INTENT` may be different from what is given in the MPI generic interface. Implementations must choose `INTENT` so that the function adheres to the MPI standard. (*End of advice to implementors.*)

  Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran `INTENT`. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy `OUT` argument. Moreover, “constants” such as MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran, but “special addresses” used in a nonstandard way. Finally, the MPI-1 generic intent is changed in several places by MPI-2. For instance, MPI_IN_PLACE changes the sense of an `OUT` argument to be `INOUT`. (*End of rationale.*)

Applications may use either the `mpi` module or the `mpif.h` include file. An implementation may require use of the module to prevent type mismatch errors (see below).

Advice to users. It is recommended to use the `mpi` module even if it is not necessary to use it to avoid type mismatch errors on a particular system. Using a module provides several potential advantages over using an include file. (*End of advice to users.*)

It must be possible to link together routines some of which `USE mpi` and others of which `INCLUDE mpif.h`.

No Type Mismatch Problems for Subroutines with Choice Arguments

A high quality MPI implementation should provide a mechanism to ensure that MPI choice arguments do not cause fatal compile-time or run-time errors due to type mismatch. An MPI implementation may require applications to use the `mpi` module, or require that it be compiled with a particular compiler flag, in order to avoid type mismatch problems.

Advice to implementors. In the case where the compiler does not generate errors, nothing needs to be done to the existing interface. In the case where the compiler may generate errors, a set of overloaded functions may be used. See the paper of M. Hennecke [8]. Even if the compiler does not generate errors, explicit interfaces for all routines would be useful for detecting errors in the argument list. Also, explicit interfaces which give `INTENT` information can reduce the amount of copying for `BUF(*)` arguments. (*End of advice to implementors.*)

10.2.5 Additional Support for Fortran Numeric Intrinsic Types

The routines in this section are part of Extended Fortran Support described in Section 10.2.4.

MPI-1 provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called `KIND`-parameterized types. These types are declared using an intrinsic type (one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER`) with an optional integer `KIND` parameter that selects from among one or more variants. The specific meaning of different `KIND` values themselves are implementation dependent and not specified by the language. Fortran provides the `KIND` selection functions `selected_real_kind` for `REAL` and `COMPLEX` types, and `selected_int_kind` for `INTEGER` types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare `KIND`-parameterized `REAL`, `COMPLEX` and `INTEGER` variables in Fortran. This scheme is backward compatible with Fortran 77. `REAL` and `INTEGER` Fortran variables have a default `KIND` if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type `REAL` with a non-default `KIND`. The following two declarations are equivalent:

```
double precision x
real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods to communicate using numeric intrinsic types. The first method can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method gives the user complete control over communication by exposing machine representations.

Parameterized Datatypes with Specified Precision and Exponent Range

MPI-1 provides named datatypes corresponding to standard Fortran 77 numeric types — MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this MPI-1 model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using `selected_real_kind(p, r)` to determine the KIND parameter, where `p` is decimal digits of precision and `r` is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes `D(p, r)`. `D(p, r)` is defined for each value of `(p, r)` supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index `(p, r)` not supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX datatypes. For integers, there is a similar implicit array related to `selected_int_kind` and indexed by the requested number of digits `r`. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but a new set.

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

Advice to users. `selected_real_kind()` maps a large number of `(p,r)` pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same `(p,r)` value (REAL and COMPLEX) or `r` value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (*End of advice to users.*)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	p	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

```
static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r)
```

This function returns a predefined MPI datatype that matches a `REAL` variable of `KIND` `selected_real_kind(p, r)`. In the model described above it returns a handle for the element `D(p, r)`. Either `p` or `r` may be omitted from calls to `selected_real_kind(p, r)` (but not both). Analogously, either `p` or `r` may be set to MPI_UNDEFINED. In communication, an MPI datatype `A` returned by MPI_TYPE_CREATE_F90_REAL matches a datatype `B` if and only if `B` was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for `p` and `r` or `B` is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	p	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

```
static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r)
```

This function returns a predefined MPI datatype that matches a `COMPLEX` variable of `KIND` `selected_real_kind(p, r)`. Either `p` or `r` may be omitted from calls to `selected_real_kind(p, r)` (but not both). Analogously, either `p` or `r` may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

MPI_TYPE_CREATE_F90_INTEGER(r, newtype)

IN	r	decimal exponent range, i.e., number of decimal digits (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR
```

```
static MPI::Datatype MPI::Datatype::Create_f90_integer(int r)
```

This function returns a predefined MPI datatype that matches a `INTEGER` variable of

KIND selected_int_kind(r). Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given.

It is erroneous to supply a value for r that is not supported by the compiler.

Example:

```
integer       longtype, quadtype
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
...

call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x,  10, quadtype, ...)
```

Advice to users. The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction operations. There are two situations in which they behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_ routines.

If a variable was declared specifying a non-default KIND value that was not obtained with selected_real_kind() or selected_int_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(*End of advice to users.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 9.5.2) or user-defined (Section 9.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 9.5.2.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double" and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with

15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.
For MPI_TYPE_CREATE_F90_REAL:

```
if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
else if (p > 15) or (r >  307) then  external32_size = 16
else if (p >  6) or (r >   37) then  external32_size =  8
else                                 external32_size =  4
```

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.
For MPI_TYPE_CREATE_F90_INTEGER:

```
if      (r > 38) then  external32 representation is undefined
else if (r > 18) then  external32_size =  16
else if (r >  9) then  external32_size =  8
else if (r >  4) then  external32_size =  4
else if (r >  2) then  external32_size =  2
else                   external32_size =  1
```

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI-1 provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran and of the form MPI::<TYPE>n in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and **n** is the length in bytes of the machine representation. This datatype locally matches all variables of type (**typeclass**, **n**). The list of names for such types includes:

```
MPI_REAL4
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
```

```
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16
```

In MPI-1 these datatypes are all optional and correspond to the optional, nonstandard declarations supported by many Fortran compilers. In MPI-2, one datatype is required for each representation supported by the compiler. To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations `REAL*n`, `INTEGER*n`, always create a variable whose representation is of size **n**. All these datatypes are predefined.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

MPI_SIZEOF(x, size)

IN	x	a Fortran variable of numeric intrinsic type (choice)
OUT	size	size of machine representation of that type (integer)

```
MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR
```

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C and C++ *sizeof* operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (*End of advice to users.*)

Rationale. This function is not available in other languages because it would not be useful. (*End of rationale.*)

MPI_TYPE_MATCH_SIZE(typeclass, size, type)

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	type	datatype with correct type, size (handle)

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)
```

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)
    INTEGER TYPECLASS, SIZE, TYPE, IERROR
```

```
static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size)
```

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C and C++, one can use the C function sizeof(), instead of MPI_SIZEOF. In addition, for variables of default kind the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the `typeclass` is known. It is erroneous to specify a size not supported by the compiler.

Rationale. This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{
  switch(typeclass) {
      case MPI_TYPECLASS_REAL: switch(size) {
        case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
        case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
        default: error(...);
      }
      case MPI_TYPECLASS_INTEGER: switch(size) {
         case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
         case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
         default: error(...);       }
     ... etc ...
   }
}
```

(*End of advice to implementors.*)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype MPI_<TYPE>n can be received with this same datatype on another process. Most modern computers use 2's complement for integers and IEEE format for floating point. Thus, communication using these size-specific datatypes will not entail loss of precision or truncation errors.

Advice to users. Care is required when communicating in a heterogeneous environment. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
```

```
else if (myrank .eq. 1) then
    call MPI_RECV(x, xtype, 100, 0, ...)
endif
```

This may not work in a heterogeneous environment if the value of size is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second is to use selected_real_kind or selected_int_kind and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., selected_real_kind(12) on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communication. This may require explicit conversion to a variable of size that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)

if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',               &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,     &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif

call MPI_BARRIER(MPI_COMM_WORLD, ierror)

if (myrank .eq. 1) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,  &
                 MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

Bibliography

[1] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[2] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[4] C++ Forum. Working paper for draft proposed international standard for information systems — programming language C++. Technical report, American National Standards Institute, 1995.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. `http://www.mpi-forum.org`.

[7] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[8] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from `http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90`.

[9] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985.

[10] International Organization for Standardization, Geneva. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987.

[11] International Organization for Standardization, Geneva. *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996.

[12] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993.

[13] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[14] S. J. Leffler, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at `http://www.netbsd.org/Documentation/lite2/psd/`.

[15] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[16] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.

[17] *4.4BSD Programmer's Supplementary Documents (PSD)*. O'Reilly and Associates, 1994.

[18] Perry Partow and Dennis Cottel. Scalable Programming Environment. Technical Report 1672, Naval Command Control and Ocean Surveillance Center (NRAD), September 1994.

[19] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[20] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. `http://www.erc.msstate.edu/mpi/mpix.html`.

[21] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In Gregory V. Wilson, editor, *Parallel Programming Using C++*, Engineering Computation Series. MIT Press, July 1996. ISBN 0-262-73118-5.

[22] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[23] *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996. ISBN 0-201-48345-9.

Annex A

Language Binding

A.1 Introduction

This annex summarizes the specific bindings for Fortran, C, and C++. First the constants, error codes, info keys, and info values are presented. Second, the MPI-1.2 bindings are given. Third, the MPI-2 bindings are given.

A.2 Defined Values and Handles

A.2.1 Defined Constants

The C and Fortran name is listed in the left column and the C++ name is listed in the right column.

Return Codes

MPI_ERR_ACCESS	MPI::ERR_ACCESS
MPI_ERR_AMODE	MPI::ERR_AMODE
MPI_ERR_ASSERT	MPI::ERR_ASSERT
MPI_ERR_BAD_FILE	MPI::ERR_BAD_FILE
MPI_ERR_BASE	MPI::ERR_BASE
MPI_ERR_CONVERSION	MPI::ERR_CONVERSION
MPI_ERR_DISP	MPI::ERR_DISP
MPI_ERR_DUP_DATAREP	MPI::ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS	MPI::ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE	MPI::ERR_FILE_IN_USE
MPI_ERR_FILE	MPI::ERR_FILE
MPI_ERR_INFO_KEY	MPI::ERR_INFO_VALUE
MPI_ERR_INFO_NOKEY	MPI::ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE	MPI::ERR_INFO_KEY
MPI_ERR_INFO	MPI::ERR_INFO
MPI_ERR_IO	MPI::ERR_IO
MPI_ERR_KEYVAL	MPI::ERR_KEYVAL
MPI_ERR_LOCKTYPE	MPI::ERR_LOCKTYPE
MPI_ERR_NAME	MPI::ERR_NAME
MPI_ERR_NO_MEM	MPI::ERR_NO_MEM
MPI_ERR_NOT_SAME	MPI::ERR_NOT_SAME
MPI_ERR_NO_SPACE	MPI::ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE	MPI::ERR_NO_SUCH_FILE
MPI_ERR_PORT	MPI::ERR_PORT
MPI_ERR_QUOTA	MPI::ERR_QUOTA
MPI_ERR_READ_ONLY	MPI::ERR_READ_ONLY
MPI_ERR_RMA_CONFLICT	MPI::ERR_RMA_CONFLICT
MPI_ERR_RMA_SYNC	MPI::ERR_RMA_SYNC
MPI_ERR_SERVICE	MPI::ERR_SERVICE
MPI_ERR_SIZE	MPI::ERR_SIZE
MPI_ERR_SPAWN	MPI::ERR_SPAWN
MPI_ERR_UNSUPPORTED_DATAREP	MPI::ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_OPERATION	MPI::ERR_UNSUPPORTED_OPERATION
MPI_ERR_WIN	MPI::ERR_WIN

Assorted Constants

MPI_IN_PLACE	MPI::IN_PLACE
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

Variable Address Size (Fortran only)

MPI_ADDRESS_KIND	Not defined for C++
MPI_INTEGER_KIND	Not defined for C++
MPI_OFFSET_KIND	Not defined for C++

Maximum Sizes for Strings

MPI_MAX_DATAREP_STRING	MPI::MAX_DATAREP_STRING
MPI_MAX_INFO_KEY	MPI::MAX_INFO_KEY
MPI_MAX_INFO_VAL	MPI::MAX_INFO_VAL
MPI_MAX_OBJECT_NAME	MPI::MAX_OBJECT_NAME
MPI_MAX_PORT_NAME	MPI::MAX_PORT_NAME

Named Predefined Datatypes

MPI_WCHAR	MPI::WCHAR

C and C++ (no Fortran) Named Predefined Datatypes

MPI_Fint	MPI::Fint

Optional C and C++ (no Fortran) Named Predefined Datatypes

MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR

Predefined Attribute Keys

MPI_APPNUM	MPI::APPNUM
MPI_LASTUSEDCODE	MPI::LASTUSEDCODE
MPI_UNIVERSE_SIZE	MPI::UNIVERSE_SIZE
MPI_WIN_BASE	MPI::WIN_BASE
MPI_WIN_DISP_UNIT	MPI::WIN_DISP_UNIT
MPI_WIN_SIZE	MPI::WIN_SIZE

Collective Operations

MPI_REPLACE	MPI::REPLACE

Null Handles

MPI_FILE_NULL	MPI::FILE_NULL
MPI_INFO_NULL	MPI::INFO_NULL
MPI_WIN_NULL	MPI::WIN_NULL

Mode Constants

MPI_MODE_APPEND	MPI::MODE_APPEND
MPI_MODE_CREATE	MPI::MODE_CREATE
MPI_MODE_DELETE_ON_CLOSE	MPI::MODE_DELETE_ON_CLOSE
MPI_MODE_EXCL	MPI::MODE_EXCL
MPI_MODE_NOCHECK	MPI::MODE_NOCHECK
MPI_MODE_NOPRECEDE	MPI::MODE_NOPRECEDE
MPI_MODE_NOPUT	MPI::MODE_NOPUT
MPI_MODE_NOSTORE	MPI::MODE_NOSTORE
MPI_MODE_NOSUCCEED	MPI::MODE_NOSUCCEED
MPI_MODE_RDONLY	MPI::MODE_RDONLY
MPI_MODE_RDWR	MPI::MODE_RDWR
MPI_MODE_SEQUENTIAL	MPI::MODE_SEQUENTIAL
MPI_MODE_UNIQUE_OPEN	MPI::MODE_UNIQUE_OPEN
MPI_MODE_WRONLY	MPI::MODE_WRONLY

Datatype Decoding Constants

MPI_COMBINER_CONTIGUOUS	MPI::COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY	MPI::COMBINER_DARRAY
MPI_COMBINER_DUP	MPI::COMBINER_DUP
MPI_COMBINER_F90_COMPLEX	MPI::COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI::COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL	MPI::COMBINER_F90_REAL
MPI_COMBINER_HINDEXED_INTEGER	MPI::COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HINDEXED	MPI::COMBINER_HINDEXED
MPI_COMBINER_HVECTOR_INTEGER	MPI::COMBINER_HVECTOR_INTEGER
MPI_COMBINER_HVECTOR	MPI::COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK	MPI::COMBINER_INDEXED_BLOCK
MPI_COMBINER_INDEXED	MPI::COMBINER_INDEXED
MPI_COMBINER_NAMED	MPI::COMBINER_NAMED
MPI_COMBINER_RESIZED	MPI::COMBINER_RESIZED
MPI_COMBINER_STRUCT_INTEGER	MPI::COMBINER_STRUCT_INTEGER
MPI_COMBINER_STRUCT	MPI::COMBINER_STRUCT
MPI_COMBINER_SUBARRAY	MPI::COMBINER_SUBARRAY
MPI_COMBINER_VECTOR	MPI::COMBINER_VECTOR

Threads Constants

MPI_THREAD_FUNNELED	MPI::THREAD_FUNNELED
MPI_THREAD_MULTIPLE	MPI::THREAD_MULTIPLE
MPI_THREAD_SERIALIZED	MPI::THREAD_SERIALIZED
MPI_THREAD_SINGLE	MPI::THREAD_SINGLE

File Operation Constants

MPI_DISPLACEMENT_CURRENT	MPI::DISPLACEMENT_CURRENT
MPI_DISTRIBUTE_BLOCK	MPI::DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG
MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE
MPI_ORDER_C	MPI::ORDER_C
MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN
MPI_SEEK_CUR	MPI::SEEK_CUR
MPI_SEEK_END	MPI::SEEK_END
MPI_SEEK_SET	MPI::SEEK_SET

F90 Datatype Matching Constants

MPI_TYPECLASS_COMPLEX	MPI::TYPECLASS_COMPLEX
MPI_TYPECLASS_INTEGER	MPI::TYPECLASS_INTEGER
MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL

Handles to Assorted Structures in C and C++ (no Fortran)

MPI_File	MPI::File
MPI_Info	MPI::Info
MPI_Win	MPI::Win

Constants Specifying Empty or Ignored Input

MPI_ARGVS_NULL	MPI::ARGVS_NULL
MPI_ARGV_NULL	MPI::ARGV_NULL
MPI_ERRCODES_IGNORE	Not defined for C++
MPI_STATUSES_IGNORE	Not defined for C++
MPI_STATUS_IGNORE	Not defined for C++

C Constants Specifying Ignored Input (no C++ or Fortran)

MPI_F_STATUSES_IGNORE	Not defined for C++
MPI_F_STATUS_IGNORE	Not defined for C++

C and C++ cpp Constants and Fortran Parameters

MPI_SUBVERSION
MPI_VERSION

A.2.2 Info Keys

access_style
appnum
arch
cb_block_size

cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename
file
host
io_node_list
ip_address
ip_port
nb_proc
no_locks
num_io_nodes
path
soft
striping_factor
striping_unit
wdir

A.2.3 Info Values

false
random
read_mostly
read_once
reverse_sequential
sequential
true
write_mostly
write_once

A.3 MPI-1.2 C Bindings

```
int MPI_Get_version(int *version, int *subversion)
```

A.4 MPI-1.2 Fortran Bindings

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

A.5 MPI-1.2 C++ Bindings

See Section B.11.

A.6 MPI-2 C Bindings

A.6.1 Miscellany

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

MPI_Fint MPI_Comm_c2f(MPI_Comm comm)

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function,
             MPI_Errhandler *errhandler)

MPI_Comm MPI_Comm_f2c(MPI_Fint comm)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_Fint MPI_File_c2f(MPI_File file)

int MPI_File_create_errhandler(MPI_File_errhandler_fn *function,
             MPI_Errhandler *errhandler)

MPI_File MPI_File_f2c(MPI_Fint file)

int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

int MPI_Finalized(int *flag)

int MPI_Free_mem(void *base)

int MPI_Get_address(void *location, MPI_Aint *address)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Info_c2f(MPI_Info info)

int MPI_Info_create(MPI_Info *info)

int MPI_Info_delete(MPI_Info info, char *key)

int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)

MPI_Info MPI_Info_f2c(MPI_Fint info)

int MPI_Info_free(MPI_Info *info)

int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,
             int *flag)

int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)

int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,
             int *flag)

int MPI_Info_set(MPI_Info info, char *key, char *value)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Op MPI_Op_f2c(MPI_Fint op)

int MPI_Pack_external(char *datarep, void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)

int MPI_Pack_external_size(char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_Request MPI_Request_f2c(MPI_Fint request)

int MPI_Request_get_status(MPI_Request request, int *flag,
             MPI_Status *status)

int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)

int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

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)

int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_indexed_block(int count, int blocklength,
             int array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
             extent, MPI_Datatype *newtype)

int MPI_Type_create_struct(int count, int array_of_blocklengths[],
             MPI_Aint array_of_displacements[],
             MPI_Datatype array_of_types[], MPI_Datatype *newtype)

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_Datatype MPI_Type_f2c(MPI_Fint datatype)

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
            MPI_Aint *extent)

int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
            MPI_Aint *true_extent)

int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
            MPI_Aint *position, void *outbuf, int outcount,
            MPI_Datatype datatype)

MPI_Fint MPI_Win_c2f(MPI_Win win)

int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function, MPI_Errhandler
            *errhandler)

MPI_Win MPI_Win_f2c(MPI_Fint win)

int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

A.6.2 Process Creation and Management

```
int MPI_Close_port(char *port_name)

int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm,
            MPI_Comm *newcomm)

int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
            MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_disconnect(MPI_Comm *comm)

int MPI_Comm_get_parent(MPI_Comm *parent)

int MPI_Comm_join(int fd, MPI_Comm *intercomm)

int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info,
            int root, MPI_Comm comm, MPI_Comm *intercomm,
            int array_of_errcodes[])

int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
            char **array_of_argv[], int array_of_maxprocs[],
            MPI_Info array_of_info[], int root, MPI_Comm comm,
            MPI_Comm *intercomm, int array_of_errcodes[])

int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)

int MPI_Open_port(MPI_Info info, char *port_name)

int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)

int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)
```

A.6.3 One-Sided Communications

```
int MPI_Accumulate(void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target_rank, MPI_Aint target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
             origin_datatype, int target_rank, MPI_Aint target_disp, int
             target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Win_complete(MPI_Win win)

int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, MPI_Win *win)

int MPI_Win_fence(int assert, MPI_Win win)

int MPI_Win_free(MPI_Win *win)

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_test(MPI_Win win, int *flag)

int MPI_Win_unlock(int rank, MPI_Win win)

int MPI_Win_wait(MPI_Win win)
```

A.6.4 Extended Collective Operations

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
             MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
             int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)

int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

A.6.5 External Interfaces

```
int MPI_Add_error_class(int *errorclass)

int MPI_Add_error_code(int errorclass, int *errorcode)

int MPI_Add_error_string(int errorcode, char *string)

int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
```

```
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)

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

int MPI_Comm_free_keyval(int *comm_keyval)

int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
            int *flag)

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)

int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)

int MPI_File_call_errhandler(MPI_File fh, int errorcode)

int MPI_Grequest_complete(MPI_Request request)

int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
            MPI_Grequest_free_function *free_fn,
            MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
            MPI_Request *request)

int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
            int *provided)

int MPI_Is_thread_main(int *flag)

int MPI_Query_thread(int *provided)

int MPI_Status_set_cancelled(MPI_Status *status, int flag)

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

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)

int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

int MPI_Type_free_keyval(int *type_keyval)

int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void
            *attribute_val, int *flag)

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[])

int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
```

```
            int *num_addresses, int *num_datatypes, int *combiner)

int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)

int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
            void *attribute_val)

int MPI_Type_set_name(MPI_Datatype type, char *type_name)

int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

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)

int MPI_Win_delete_attr(MPI_Win win, int win_keyval)

int MPI_Win_free_keyval(int *win_keyval)

int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
            int *flag)

int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)

int MPI_Win_set_name(MPI_Win win, char *win_name)
```

A.6.6 I/O

```
int MPI_File_close(MPI_File *fh)

int MPI_File_delete(char *filename, MPI_Info info)

int MPI_File_get_amode(MPI_File fh, int *amode)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
            MPI_Offset *disp)

int MPI_File_get_group(MPI_File fh, MPI_Group *group)

int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)

int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)

int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
            MPI_Aint *extent)

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
            MPI_Datatype *filetype, char *datarep)
```

```
int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Request *request)

int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_File_iwrite(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Request *request)

int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
            MPI_File *fh)

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)

int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Status *status)

int MPI_File_read_all(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype)

int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype)

int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype)

int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_read_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)

int MPI_File_set_atomicity(MPI_File fh, int flag)

int MPI_File_set_info(MPI_File fh, MPI_Info info)

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
            MPI_Datatype filetype, char *datarep, MPI_Info info)

int MPI_File_sync(MPI_File fh)

int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
            MPI_Status *status)

int MPI_File_write_all(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype)

int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
            int count, MPI_Datatype datatype)

int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype)

int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)

int MPI_File_write_shared(MPI_File fh, void *buf, int count,
            MPI_Datatype datatype, MPI_Status *status)

int MPI_Register_datarep(char *datarep,
            MPI_Datarep_conversion_function *read_conversion_fn,
            MPI_Datarep_conversion_function *write_conversion_fn,
            MPI_Datarep_extent_function *dtype_file_extent_fn,
            void *extra_state)
```

A.6.7 Language Bindings

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
```

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)

int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)
```

A.6.8 User Defined Functions

```
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);

typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
             void *attribute_val, void *extra_state);

typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);

typedef int MPI_Datarep_conversion_function(void *userbuf,
             MPI_Datatype datatype, int count, void *filebuf,
             MPI_Offset position, void *extra_state);

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
             MPI_Aint *file_extent, void *extra_state);

typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);

typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

typedef int MPI_Grequest_free_function(void *extra_state);

typedef int MPI_Grequest_query_function(void *extra_state,
             MPI_Status *status);

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);

typedef int MPI_Type_delete_attr_function(MPI_Datatype type, int type_keyval,
             void *attribute_val, void *extra_state);

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);

typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);

typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);
```

A.7 MPI-2 Fortran Bindings

A.7.1 Miscellany

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
```

```
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR

MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR

MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS

MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR

MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY

MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR

MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR

MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG

MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
```

```
    INTEGER INFO, NKEYS, IERROR

MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
    INTEGER INFO, N, IERROR
    CHARACTER*(*) KEY

MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    LOGICAL FLAG
    CHARACTER*(*) KEY

MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY, VALUE

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(*)

MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP

MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

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_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_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE

MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR

MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

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(*)

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_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT

MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT

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(*)

MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR

MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR

MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

A.7.2 Process Creation and Management

```
MPI_CLOSE_PORT(PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERROR

MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_DISCONNECT(COMM, IERROR)
```

```
    INTEGER COMM, IERROR

MPI_COMM_GET_PARENT(PARENT, IERROR)
    INTEGER PARENT, IERROR

MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
    INTEGER FD, INTERCOMM, IERROR

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
              ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
              ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
              ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)

MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR

MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR

MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME

MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

A.7.3 One-Sided Communications

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR

MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
              TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
```

```
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR

MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR

MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR

MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR

MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG

MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR

MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

A.7.4 Extended Collective Operations

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

A.7.5 External Interfaces

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR

MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR

MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

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

MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR

MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR

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

MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME

MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME

MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR

MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
```

```
            IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR

MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

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

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

MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR

MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR

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

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(*)

MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
            COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

```
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER TYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME

MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
    INTEGER TYPE, IERROR
    CHARACTER*(*) TYPE_NAME

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

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

MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR

MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR

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

MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME

MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

A.7.6 I/O

```
MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR

MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR
```

```
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG

MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR

MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR

MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP, INTEGER(KIND=MPI_OFFSET_KIND) DISP

MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR

MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
```

```
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG

MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR

MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP

MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR

MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
```

```
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
              DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

A.7.7 Language Bindings

```
MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR
```

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR

MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR

MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)
    INTEGER TYPECLASS, SIZE, TYPE, IERROR
```

A.7.8 User Defined Subroutines

```
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

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

SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...)
    INTEGER COMM, ERROR_CODE

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
             POSITION, EXTRA_STATE, IERROR)
    <TYPE> USERBUF(*), FILEBUF(*)
    INTEGER COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) POSITION
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)
    INTEGER FILE, ERROR_CODE

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
```

```
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

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

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

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

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

SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...)
    INTEGER WIN, ERROR_CODE
```

A.8 MPI-2 C++ Bindings

A.8.1 Miscellany

```
void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)

static MPI::Errhandler
             MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_fn*
             function)

MPI::Errhandler MPI::Comm::Get_errhandler() const

void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler)

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::Datatype MPI::Datatype::Create_hindexed(int count,
             const int array_of_blocklengths[],
             const MPI::Aint array_of_displacements[]) const
```

```
MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
            MPI::Aint stride) const

MPI::Datatype MPI::Datatype::Create_indexed_block( int count,
            int blocklength, const int array_of_displacements[]) const

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[])

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

void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const

void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
            MPI::Aint& true_extent) const

void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
            int incount, void* outbuf, MPI::Aint outsize,
            MPI::Aint& position) const

MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
            int incount) const

MPI::Datatype MPI::Datatype::Resized(const MPI::Aint lb,
            const MPI::Aint extent) const

void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
            MPI::Aint insize, MPI::Aint& position, void* outbuf,
            int outcount) const

static MPI::Errhandler
            MPI::File::Create_errhandler(MPI::File::Errhandler_fn*
            function)

MPI::Errhandler MPI::File::Get_errhandler() const

void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)

void MPI::Free_mem(void *base)

MPI::Aint MPI::Get_address(void* location)

static MPI::Info MPI::Info::Create()

void MPI::Info::Delete(const char* key)

MPI::Info MPI::Info::Dup() const

void MPI::Info::Free()

bool MPI::Info::Get(const char* key, int valuelen, char* value) const

int MPI::Info::Get_nkeys() const
```

```
void MPI::Info::Get_nthkey(int n, char* key) const

bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const

void MPI::Info::Set(const char* key, const char* value)

bool MPI::Is_finalized()

bool MPI::Request::Get_status() const

bool MPI::Request::Get_status(MPI::Status& status) const

static MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn*
            function)

MPI::Errhandler MPI::Win::Get_errhandler() const

void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)
```

A.8.2 Process Creation and Management

```
void MPI::Close_port(const char* port_name)

void MPI::Comm::Disconnect()

static MPI::Intercomm MPI::Comm::Get_parent()

static MPI::Intercomm MPI::Comm::Join(const int fd)

MPI::Intercomm MPI::Intracomm::Accept(const char* port_name,
            const MPI::Info& info, int root) const

MPI::Intercomm MPI::Intracomm::Connect(const char* port_name,
            const MPI::Info& info, int root) const

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
            const char* argv[], int maxprocs, const MPI::Info& info,
            int root) const

MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
            const char* argv[], int maxprocs, const MPI::Info& info,
            int root, int array_of_errcodes[]) const

MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
            const char* array_of_commands[], const char** array_of_argv[],
            const int array_of_maxprocs[], const MPI::Info array_of_info[],
            int root)

MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
            const char* array_of_commands[], const char** array_of_argv[],
            const int array_of_maxprocs[], const MPI::Info array_of_info[],
            int root, int array_of_errcodes[])

void MPI::Lookup_name(const char* service_name, const MPI::Info& info,
            char* port_name)
```

```
void MPI::Open_port(const MPI::Info& info, char* port_name)

void MPI::Publish_name(const char* service_name, const MPI::Info& info,
            const char* port_name)

void MPI::Unpublish_name(const char* service_name, const MPI::Info& info,
            const char* port_name)
```

A.8.3 One-Sided Communications

```
void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const
            MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
            target_disp, int target_count, const MPI::Datatype&
            target_datatype, const MPI::Op& op) const

void MPI::Win::Complete() const

static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int
            disp_unit, const MPI::Info& info, const MPI::Intracomm& comm)

void MPI::Win::Fence(int assert) const

void MPI::Win::Free()

void MPI::Win::Get(const void *origin_addr, int origin_count, const
            MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
            target_disp, int target_count, const MPI::Datatype&
            target_datatype) const

MPI::Group MPI::Win::Get_group() const

void MPI::Win::Lock(int lock_type, int rank, int assert) const

void MPI::Win::Post(const MPI::Group& group, int assert) const

void MPI::Win::Put(const void* origin_addr, int origin_count, const
            MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
            target_disp, int target_count, const MPI::Datatype&
            target_datatype) const

void MPI::Win::Start(const MPI::Group& group, int assert) const

bool MPI::Win::Test() const

void MPI::Win::Unlock(int rank) const

void MPI::Win::Wait() const
```

A.8.4 Extended Collective Operations

```
void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
            MPI::Datatype& sendtype, void* recvbuf, int recvcount,
            const MPI::Datatype& recvtype) const = 0
```

```
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

void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const = 0

void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0

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

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

void MPI::Comm::Barrier() const = 0

void MPI::Comm::Bcast(void* buffer, int count,
             const MPI::Datatype& datatype, int root) const = 0

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

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

void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op, int root)
             const = 0

void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
             int recvcounts[], const MPI::Datatype& datatype,
             const MPI::Op& op) const = 0

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

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::Intercomm MPI::Intercomm::Create(const Group& group) const
```

```
MPI::Intercomm MPI::Intercomm::Split(int color, int key) const

MPI::Intracomm MPI::Intracomm::Create(const Group& group) const

void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const

MPI::Intracomm MPI::Intracomm::Split(int color, int key) const
```

A.8.5 External Interfaces

```
int MPI::Add_error_class()

int MPI::Add_error_code(int errorclass)

void MPI::Add_error_string(int errorcode, const char* string)

void MPI::Comm::Call_errhandler(int errorcode) const

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)

void MPI::Comm::Delete_attr(int comm_keyval)

static void MPI::Comm::Free_keyval(int& comm_keyval)

bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const

void MPI::Comm::Get_name(char* comm_name, int& resultlen) const

void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const

void MPI::Comm::Set_name(const char* comm_name)

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)

void MPI::Datatype::Delete_attr(int type_keyval)

MPI::Datatype MPI::Datatype::Dup() const

static void MPI::Datatype::Free_keyval(int& type_keyval)

bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const

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

void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const

void MPI::Datatype::Get_name(char* type_name, int& resultlen) const
```

```
void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val)

void MPI::Datatype::Set_name(const char* type_name)

void MPI::File::Call_errhandler(int errorcode) const

void MPI::Grequest::Complete()

static MPI::Grequest
             MPI::Grequest::Start(const MPI::Grequest::Query_function
             query_fn, const MPI::Grequest::Free_function free_fn,
             const MPI::Grequest::Cancel_function cancel_fn,
             void *extra_state)

int MPI::Init_thread(int required)

int MPI::Init_thread(int& argc, char**& argv, int required)

bool MPI::Is_thread_main()

int MPI::Query_thread()

void MPI::Status::Set_cancelled(bool flag)

void MPI::Status::Set_elements(const MPI::Datatype& datatype, int count)

void MPI::Win::Call_errhandler(int errorcode) const

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)

void MPI::Win::Delete_attr(int win_keyval)

static void MPI::Win::Free_keyval(int& win_keyval)

bool MPI::Win::Get_attr(const MPI::Win& win, int win_keyval,
             void* attribute_val) const

void MPI::Win::Get_name(char* win_name, int& resultlen) const

void MPI::Win::Set_attr(int win_keyval, const void* attribute_val)

void MPI::Win::Set_name(const char* win_name)
```

A.8.6 I/O

```
void MPI::File::Close()

static void MPI::File::Delete(const char* filename, const MPI::Info& info)

int MPI::File::Get_amode() const

bool MPI::File::Get_atomicity() const

MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const
```

```
MPI::Group MPI::File::Get_group() const

MPI::Info MPI::File::Get_info() const

MPI::Offset MPI::File::Get_position() const

MPI::Offset MPI::File::Get_position_shared() const

MPI::Offset MPI::File::Get_size() const

MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const

void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype& etype,
             MPI::Datatype& filetype, char* datarep) const

MPI::Request MPI::File::Iread(void* buf, int count,
             const MPI::Datatype& datatype)

MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype)

MPI::Request MPI::File::Iread_shared(void* buf, int count,
             const MPI::Datatype& datatype)

MPI::Request MPI::File::Iwrite(const void* buf, int count,
             const MPI::Datatype& datatype)

MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype)

MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
             const MPI::Datatype& datatype)

static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
             const char* filename, int amode, const MPI::Info& info)

void MPI::File::Preallocate(MPI::Offset size)

void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype)

void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
             MPI::Status& status)

void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Read_all_begin(void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Read_all_end(void* buf)

void MPI::File::Read_all_end(void* buf, MPI::Status& status)

void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype)
```

```
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Read_at_all_end(void* buf)

void MPI::File::Read_at_all_end(void* buf, MPI::Status& status)

void MPI::File::Read_ordered(void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Read_ordered(void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Read_ordered_begin(void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Read_ordered_end(void* buf)

void MPI::File::Read_ordered_end(void* buf, MPI::Status& status)

void MPI::File::Read_shared(void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Read_shared(void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Seek(MPI::Offset offset, int whence)

void MPI::File::Seek_shared(MPI::Offset offset, int whence)

void MPI::File::Set_atomicity(bool flag)

void MPI::File::Set_info(const MPI::Info& info)

void MPI::File::Set_size(MPI::Offset size)

void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
            const MPI::Datatype& filetype, const char* datarep,
            const MPI::Info& info)

void MPI::File::Sync()

void MPI::File::Write(const void* buf, int count,
            const MPI::Datatype& datatype)

void MPI::File::Write(const void* buf, int count,
            const MPI::Datatype& datatype, MPI::Status& status)
```

```
void MPI::File::Write_all(const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_all(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_all_begin(const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_all_end(const void* buf)

void MPI::File::Write_all_end(const void* buf, MPI::Status& status)

void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype)

void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype)

void MPI::File::Write_at_all_end(const void* buf)

void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status)

void MPI::File::Write_ordered(const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_ordered(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_ordered_begin(const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_ordered_end(const void* buf)

void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status)

void MPI::File::Write_shared(const void* buf, int count,
             const MPI::Datatype& datatype)

void MPI::File::Write_shared(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status)

void MPI::Register_datarep(const char* datarep,
             MPI::Datarep_conversion_function* read_conversion_fn,
             MPI::Datarep_conversion_function* write_conversion_fn,
             MPI::Datarep_extent_function* dtype_file_extent_fn,
             void* extra_state)
```

A.8.7 Language Bindings

```
static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r)

static MPI::Datatype MPI::Datatype::Create_f90_integer(int r)

static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r)

static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size)
```

A.8.8 User Defined Functions

```
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);

typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
             int comm_keyval, void* attribute_val, void* extra_state);

typedef void MPI::Comm::Errhandler_fn(MPI::Comm &, int *, ...);

typedef MPI::Datarep_conversion_function(void* userbuf,
             MPI::Datatype& datatype, int count, void* filebuf,
             MPI::Offset position, void* extra_state);

typedef MPI::Datarep_extent_function(const MPI::Datatype& datatype,
             MPI::Aint& file_extent, void* extra_state);

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);

typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& type,
             int type_keyval, void* attribute_val, void* extra_state);

typedef void MPI::File::Errhandler_fn(MPI::File &, int *, ...);

typedef int MPI::Grequest::Cancel_function(void* extra_state,
             bool complete);

typedef int MPI::Grequest::Free_function(void* extra_state);

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

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);

typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
             void* attribute_val, void* extra_state);

typedef void MPI::Win::Errhandler_fn(MPI::Win &, int *, ...);
```

Annex B

MPI-1 C++ Language Binding

B.1 C++ Classes

The following are the classes provided with the C++ MPI-1 language bindings:

```
namespace MPI {
  class Comm                             {...};
  class Intracomm : public Comm          {...};
  class Graphcomm : public Intracomm     {...};
  class Cartcomm  : public Intracomm     {...};
  class Intercomm : public Comm          {...};
  class Datatype                         {...};
  class Errhandler                       {...};
  class Exception                        {...};
  class Group                            {...};
  class Op                               {...};
  class Request                          {...};
  class Prequest  : public Request       {...};
  class Status                           {...};
};
```

Note that several MPI-1 functions, constants, and typedefs have been deprecated and therefore do not have corresponding C++ bindings. All deprecated names have corresponding new names in MPI-2 (albeit probably with different semantics). See the table in Section 2.6.1 for a list of the deprecated names and their corresponding new names. The bindings for the new names are listed in Annex A.

B.2 Defined Constants

These are required constants, defined in the file `mpi.h`. For brevity, the types of the constants are defined below are defined in the comments.

```
// return codes
// Type: const int (or unnamed enum)
MPI::SUCCESS
MPI::ERR_BUFFER
```

```
MPI::ERR_COUNT
MPI::ERR_TYPE
MPI::ERR_TAG
MPI::ERR_COMM
MPI::ERR_RANK
MPI::ERR_REQUEST
MPI::ERR_ROOT
MPI::ERR_GROUP
MPI::ERR_OP
MPI::ERR_TOPOLOGY
MPI::ERR_DIMS
MPI::ERR_ARG
MPI::ERR_UNKNOWN
MPI::ERR_TRUNCATE
MPI::ERR_OTHER
MPI::ERR_INTERN
MPI::ERR_PENDING
MPI::ERR_IN_STATUS
MPI::ERR_LASTCODE

// assorted constants
// Type: const void *
MPI::BOTTOM
// Type: const int (or unnamed enum)
MPI::PROC_NULL
MPI::ANY_SOURCE
MPI::ANY_TAG
MPI::UNDEFINED
MPI::BSEND_OVERHEAD
MPI::KEYVAL_INVALID

// Error-handling specifiers
// Type: MPI::Errhandler (see below)
MPI::ERRORS_ARE_FATAL
MPI::ERRORS_RETURN
MPI::ERRORS_THROW_EXCEPTIONS

// Maximum sizes for strings
// Type: const int
MPI::MAX_PROCESSOR_NAME
MPI::MAX_ERROR_STRING

// elementary datatypes (C / C++)
// Type: const MPI::Datatype
MPI::CHAR
MPI::SHORT
MPI::INT
MPI::LONG
```

```
MPI::SIGNED_CHAR
MPI::UNSIGNED_CHAR
MPI::UNSIGNED_SHORT
MPI::UNSIGNED
MPI::UNSIGNED_LONG
MPI::FLOAT
MPI::DOUBLE
MPI::LONG_DOUBLE
MPI::BYTE
MPI::PACKED

// elementary datatypes (Fortran)
// Type: const MPI::Datatype
MPI::INTEGER
MPI::REAL
MPI::DOUBLE_PRECISION
MPI::F_COMPLEX
MPI::F_DOUBLE_COMPLEX
MPI::LOGICAL
MPI::CHARACTER

// datatypes for reduction functions (C / C++)
// Type: const MPI::Datatype
MPI::FLOAT_INT
MPI::DOUBLE_INT
MPI::LONG_INT
MPI::TWOINT
MPI::SHORT_INT
MPI::LONG_DOUBLE_INT

// datatype for reduction functions (Fortran)
// Type const MPI::Datatype
MPI::TWOREAL
MPI::TWODOUBLE_PRECISION
MPI::TWOINTEGER

// optional datatypes (Fortran)
// Type: const MPI::Datatype
MPI::INTEGER1
MPI::INTEGER2
MPI::INTEGER4
MPI::REAL2
MPI::REAL4
MPI::REAL8

// optional datatypes (C / C++)
// Type: const MPI::Datatype
```

```
MPI::LONG_LONG
MPI::UNSIGNED_LONG_LONG


// special datatypes for construction derived datatypes
// Type: const MPI::Datatype
MPI::UB
MPI::LB

// C++ datatypes
// Type: const MPI::Datatype
MPI::BOOL
MPI::COMPLEX
MPI::DOUBLE_COMPLEX
MPI::LONG_DOUBLE_COMPLEX

// reserved communicators
// Type: MPI::Intracomm
MPI::COMM_WORLD
MPI::COMM_SELF

// results of communicator and group comparisons
// Type: const int (or unnamed enum)
MPI::IDENT
MPI::CONGRUENT
MPI::SIMILAR
MPI::UNEQUAL

// environmental inquiry keys
// Type: const int (or unnamed enum)
MPI::TAG_UB
MPI::IO
MPI::HOST
MPI::WTIME_IS_GLOBAL

// collective operations
// Type: const MPI::Op
MPI::MAX
MPI::MIN
MPI::SUM
MPI::PROD
MPI::MAXLOC
MPI::MINLOC
MPI::BAND
MPI::BOR
MPI::BXOR
MPI::LAND
MPI::LOR
```

```
MPI::LXOR

// Null handles
// Type: const MPI::Group
MPI::GROUP_NULL

// Type:  See Section 10.1.7 regarding the MPI::Comm class hierarchy and
// the specific type of MPI::COMM_NULL.

MPI::COMM_NULL
// Type: const MPI::Datatype
MPI::DATATYPE_NULL
// Type: const MPI::Request
MPI::REQUEST_NULL
// Type: const MPI::Op
MPI::OP_NULL
// Type: MPI::Errhandler
MPI::ERRHANDLER_NULL

// Empty group
// Type: const MPI::Group
MPI::GROUP_EMPTY

// Topologies
// Type: const int (or unnamed enum)
MPI::GRAPH
MPI::CART

// Predefined functions
// Type: MPI::Copy_function
MPI::NULL_COPY_FN
MPI::DUP_FN
// Type: MPI::Delete_function
MPI::NULL_DELETE_FN
```

B.3 Typedefs

The following are defined C++ types, also included in the file `mpi.h`.

```
// Typedef
MPI::Aint
```

The rest of this annex uses the `namespace` notation because all the functions listed below are prototypes. The `namespace` notation is not used previously because the lists of constants and types above are not actual declarations.

```
// prototypes for user-defined functions
namespace MPI {
  typedef void User_function(const void *invec, void* inoutvec, int len,
```

```
                          const Datatype& datatype);
};
```

B.4 C++ Bindings for Point-to-Point Communication

Except where specifically noted, all non-`static` member functions in this annex are `virtual`. For brevity, the keyword `virtual` is omitted.

```
namespace MPI {

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

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

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

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

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

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

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

  void Attach_buffer(void* buffer, int size)

  int Detach_buffer(void*& buffer)

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

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

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

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

  Request Comm::Irecv(void* buf, int count, const Datatype& datatype,
            int source, int tag) const

  void Request::Wait(Status& status)

  void Request::Wait()

  bool Request::Test(Status& status)

  bool Request::Test()
```

```
void Request::Free()

static int Request::Waitany(int count, Request array_of_requests[],
          Status& status)

static int Request::Waitany(int count, Request array_of_requests[])

static bool Request::Testany(int count, Request array_of_requests[],
          int& index, Status& status)

static bool Request::Testany(int count, Request array_of_requests[],
          int& index)

static void Request::Waitall(int count, Request array_of_requests[],
          Status array_of_statuses[])

static void Request::Waitall(int count, Request array_of_requests[])

static bool Request::Testall(int count, Request array_of_requests[],
          Status array_of_statuses[])

static bool Request::Testall(int count, Request array_of_requests[])

static int Request::Waitsome(int incount, Request array_of_requests[],
          int array_of_indices[], Status array_of_statuses[])

static int Request::Waitsome(int incount, Request array_of_requests[],
          int array_of_indices[])

static int Request::Testsome(int incount, Request array_of_requests[],
          int array_of_indices[], Status array_of_statuses[])

static int Request::Testsome(int incount, Request array_of_requests[],
          int array_of_indices[])

bool Comm::Iprobe(int source, int tag, Status& status) const

bool Comm::Iprobe(int source, int tag) const

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

void Comm::Probe(int source, int tag) const

void Request::Cancel() const

bool Status::Is_cancelled() const

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

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

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

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

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

  void Prequest::Start()

  static void Prequest::Startall(int count, Prequest array_of_requests[])

  void Comm::Sendrecv(const void *sendbuf, int sendcount, const
            Datatype& sendtype, int dest, int sendtag, void *recvbuf,
            int recvcount, const Datatype& recvtype, int source,
            int recvtag, Status& status) const

  void Comm::Sendrecv(const void *sendbuf, int sendcount, const
            Datatype& sendtype, int dest, int sendtag, void *recvbuf,
            int recvcount, const Datatype& recvtype, int source,
            int recvtag) const

  void Comm::Sendrecv_replace(void* buf, int count, const
            Datatype& datatype, int dest, int sendtag, int source,
            int recvtag, Status& status) const

  void Comm::Sendrecv_replace(void* buf, int count, const
            Datatype& datatype, int dest, int sendtag, int source,
            int recvtag) const

  Datatype Datatype::Create_contiguous(int count) const

  Datatype Datatype::Create_vector(int count, int blocklength, int stride)
            const

  Datatype Datatype::Create_indexed(int count,
            const int array_of_blocklengths[],
            const int array_of_displacements[]) const

  int Datatype::Get_size() const

  void Datatype::Commit()

  void Datatype::Free()

  int Status::Get_elements(const Datatype& datatype) const

  void Datatype::Pack(const void* inbuf, int incount, void *outbuf,
            int outsize, int& position, const Comm &comm) const

  void Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
            int outcount, int& position, const Comm& comm) const

  int Datatype::Pack_size(int incount, const Comm& comm) const

};
```

B.5 C++ Bindings for Collective Communication

```
namespace MPI {
```

```
void Intracomm::Barrier() const

void Intracomm::Bcast(void* buffer, int count, const Datatype& datatype,
          int root) const

void Intracomm::Gather(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, int recvcount,
          const Datatype& recvtype, int root) const

void Intracomm::Gatherv(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, const int recvcounts[],
          const int displs[], const Datatype& recvtype, int root) const

void Intracomm::Scatter(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, int recvcount,
          const Datatype& recvtype, int root) const

void Intracomm::Scatterv(const void* sendbuf, const int sendcounts[],
          const int displs[], const Datatype& sendtype, void* recvbuf,
          int recvcount, const Datatype& recvtype, int root) const

void Intracomm::Allgather(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, int recvcount,
          const Datatype& recvtype) const

void Intracomm::Allgatherv(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, const int recvcounts[],
          const int displs[], const Datatype& recvtype) const

void Intracomm::Alltoall(const void* sendbuf, int sendcount, const
          Datatype& sendtype, void* recvbuf, int recvcount,
          const Datatype& recvtype) const

void Intracomm::Alltoallv(const void* sendbuf, const int sendcounts[],
          const int sdispls[], const Datatype& sendtype, void* recvbuf,
          const int recvcounts[], const int rdispls[],
          const Datatype& recvtype) const

void Intracomm::Reduce(const void* sendbuf, void* recvbuf, int count,
          const Datatype& datatype, const Op& op, int root) const

void Op::Init(User_function* function, bool commute)

void Op::Free()

void Intracomm::Allreduce(const void* sendbuf, void* recvbuf, int count,
          const Datatype& datatype, const Op& op) const

void Intracomm::Reduce_scatter(const void* sendbuf, void* recvbuf,
          int recvcounts[], const Datatype& datatype, const Op& op)
          const

void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
          const Datatype& datatype, const Op& op) const
```

```
};
```

B.6 C++ Bindings for Groups, Contexts, and Communicators

For both syntactic and semantic reasons, the `Dup()` functions listed below are not virtual. Syntactically, they must each have a different return type. Dup() and Clone are discussed in Section 10.1.7.

```
namespace MPI {

  int Group::Get_size() const

  int Group::Get_rank() const

  static void Group::Translate_ranks (const Group& group1, int n,
            const int ranks1[], const Group& group2, int ranks2[])

  static int Group::Compare(const Group& group1, const Group& group2)

  Group Comm::Get_group() const

  static Group Group::Union(const Group& group1, const Group& group2)

  static Group Group::Intersect(const Group& group1, const Group& group2)

  static Group Group::Difference(const Group& group1, const Group& group2)

  Group Group::Incl(int n, const int ranks[]) const

  Group Group::Excl(int n, const int ranks[]) const

  Group Group::Range_incl(int n, const int ranges[][3]) const

  Group Group::Range_excl(int n, const int ranges[][3]) const

  void Group::Free()

  int Comm::Get_size() const

  int Comm::Get_rank() const

  static int Comm::Compare(const Comm& comm1, const Comm& comm2)

  Intracomm Intracomm::Dup() const

  Intercomm Intercomm::Dup() const

  Cartcomm Cartcomm::Dup() const

  Graphcomm Graphcomm::Dup() const

  Comm& Comm::Clone() const = 0

  Intracomm& Intracomm::Clone() const

  Intercomm& Intercomm::Clone() const

  Cartcomm& Cartcomm::Clone() const
```

```
  Graphcomm& Graphcomm::Clone() const

  Intracomm Intracomm::Create(const Group& group) const

  Intracomm Intracomm::Split(int color, int key) const

  void Comm::Free()

  bool Comm::Is_inter() const

  int Intercomm::Get_remote_size() const

  Group Intercomm::Get_remote_group() const

  Intercomm Intracomm::Create_intercomm(int local_leader, const
            Comm& peer_comm, int remote_leader, int tag) const

  Intracomm Intercomm::Merge(bool high) const

};
```

B.7 C++ Bindings for Process Topologies

```
namespace MPI {

  Cartcomm Intracomm::Create_cart(int ndims, const int dims[],
            const bool periods[], bool reorder) const

  void Compute_dims(int nnodes, int ndims, int dims[])

  Graphcomm Intracomm::Create_graph(int nnodes, const int index[],
            const int edges[], bool reorder) const

  int Comm::Get_topology() const

  void Graphcomm::Get_dims(int nnodes[], int nedges[]) const

  void Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
            int edges[]) const

  int Cartcomm::Get_dim() const

  void Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
            int coords[]) const

  int Cartcomm::Get_cart_rank(const int coords[]) const

  void Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const

  int Graphcomm::Get_neighbors_count(int rank) const

  void Graphcomm::Get_neighbors(int rank, int maxneighbors, int
            neighbors[]) const

  void Cartcomm::Shift(int direction, int disp, int& rank_source,
            int& rank_dest) const
```

```
  Cartcomm Cartcomm::Sub(const bool remain_dims[]) const

  int Cartcomm::Map(int ndims, const int dims[], const bool periods[])
             const

  int Graphcomm::Map(int nnodes, const int index[], const int edges[])
             const

};
```

B.8 C++ Bindings for Environmental Inquiry

```
namespace MPI {

  void Get_processor_name(char* name, int& resultlen)

  void Errhandler::Free()

  void Get_error_string(int errorcode, char* name, int& resultlen)

  int Get_error_class(int errorcode)

  double Wtime()

  double Wtick()

  void Init(int& argc, char**& argv)

  void Init()

  void Finalize()

  bool Is_initialized()

  void Comm::Abort(int errorcode)

};
```

B.9 C++ Bindings for Profiling

```
namespace MPI {

  void Pcontrol(const int level, ...)

};
```

B.10 C++ Bindings for Status Access

```
namespace MPI {
```

```
  int Status::Get_source() const

  void Status::Set_source(int source)

  int Status::Get_tag() const

  void Status::Set_tag(int tag)

  int Status::Get_error() const

  void Status::Set_error(int error)

};
```

B.11 C++ Bindings for New 1.2 Functions

```
namespace MPI {

  void Get_version(int& version, int& subversion);

};
```

B.12 C++ Bindings for Exceptions

```
namespace MPI {

  Exception::Exception(int error_code);

  int Exception::Get_error_code() const;

  int Exception::Get_error_class() const;

  const char* Exception::Get_error_string() const;

};
```

B.13 C++ Bindings on all MPI Classes

The C++ language requires all classes to have four special functions: a default constructor, a copy constructor, a destructor, and an assignment operator. The bindings for these functions are listed below; their semantics are discussed in Section 10.1.5. The two constructors are *not* `virtual`. The bindings prototype functions using the type ⟨`CLASS`⟩ rather than listing each function for every MPI class; the token ⟨`CLASS`⟩ can be replaced with valid MPI-2 class names, such as `Group`, `Datatype`, etc., except when noted. In addition, bindings are provided for comparison and inter-language operability from Sections 10.1.5 and 10.1.9.

B.13.1 Construction / Destruction

```
namespace MPI {

  ⟨CLASS⟩::⟨CLASS⟩()

  ⟨CLASS⟩::~⟨CLASS⟩()

};
```

B.13.2 Copy / Assignment

```
namespace MPI {

  ⟨CLASS⟩::⟨CLASS⟩(const ⟨CLASS⟩& data)

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const ⟨CLASS⟩& data)

};
```

B.13.3 Comparison

Since `Status` instances are not handles to underlying MPI objects, the `operator==()` and `operator!=()` functions are not defined on the `Status` class.

```
namespace MPI {

  bool ⟨CLASS⟩::operator==(const ⟨CLASS⟩& data) const

  bool ⟨CLASS⟩::operator!=(const ⟨CLASS⟩& data) const

};
```

B.13.4 Inter-language Operability

Since there are no C++ `MPI::STATUS_IGNORE` and `MPI::STATUSES_IGNORE` objects, the results of promoting the C or Fortran handles (MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE) to C++ is undefined.

```
namespace MPI {

  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩::⟨CLASS⟩(const MPI_⟨CLASS⟩& data)

  ⟨CLASS⟩::operator MPI_⟨CLASS⟩() const

};
```

B.13.5 Function Name Cross Reference

Since some of the C++ bindings have slightly different names than their C and Fortran counterparts, this section maps each language neutral MPI-1 name to its corresponding C++ binding.

For brevity, the "`MPI::`" prefix is assumed for all C++ class names.

Where MPI-1 names have been deprecated, the <`none`> keyword is used in the "Member name" column to indicate that this function is supported with a new name (see Annex A).

Where non-`void` values are listed in the "Return value" column, the given name is that of the corresponding parameter in the language neutral specification.

MPI Function	C++ class	Member name	Return value
MPI_ABORT	Comm	Abort	void
MPI_ADDRESS		<none>	
MPI_ALLGATHERV	Intracomm	Allgatherv	void
MPI_ALLGATHER	Intracomm	Allgather	void
MPI_ALLREDUCE	Intracomm	Allreduce	void
MPI_ALLTOALLV	Intracomm	Alltoallv	void
MPI_ALLTOALL	Intracomm	Alltoall	void
MPI_ATTR_DELETE		<none>	
MPI_ATTR_GET		<none>	
MPI_ATTR_PUT		<none>	
MPI_BARRIER	Intracomm	Barrier	void
MPI_BCAST	Intracomm	Bcast	void
MPI_BSEND_INIT	Comm	Bsend_init	Prequest request
MPI_BSEND	Comm	Bsend	void
MPI_BUFFER_ATTACH		Attach_buffer	void
MPI_BUFFER_DETACH		Detach_buffer	void* buffer
MPI_CANCEL	Request	Cancel	void
MPI_CARTDIM_GET	Cartcomm	Get_dim	int ndims
MPI_CART_COORDS	Cartcomm	Get_coords	void
MPI_CART_CREATE	Intracomm	Create_cart	Cartcomm newcomm
MPI_CART_GET	Cartcomm	Get_topo	void
MPI_CART_MAP	Cartcomm	Map	int newrank
MPI_CART_RANK	Cartcomm	Get_rank	int rank
MPI_CART_SHIFT	Cartcomm	Shift	void
MPI_CART_SUB	Cartcomm	Sub	Cartcomm newcomm
MPI_COMM_COMPARE	Comm	static Compare	int result
MPI_COMM_CREATE	Intracomm	Create	Intracomm newcomm
MPI_COMM_DUP	Intracomm	Dup	Intracomm newcomm
	Cartcomm	Dup	Cartcomm newcomm
	Graphcomm	Dup	Graphcomm newcomm
	Intercomm	Dup	Intercomm newcomm
	Comm	Clone	Comm& newcomm
	Intracomm	Clone	Intracomm& newcomm
	Cartcomm	Clone	Cartcomm& newcomm
	Graphcomm	Clone	Graphcomm& newcomm
	Intercomm	Clone	Intercomm& newcomm
MPI_COMM_FREE	Comm	Free	void
MPI_COMM_GROUP	Comm	Get_group	Group group
MPI_COMM_RANK	Comm	Get_rank	int rank
MPI_COMM_REMOTE_GROUP	Intercomm	Get_remote_group	Group group
MPI_COMM_REMOTE_SIZE	Intercomm	Get_remote_size	int size
MPI_COMM_SIZE	Comm	Get_size	int size
MPI_COMM_SPLIT	Intracomm	Split	Intracomm newcomm
MPI_COMM_TEST_INTER	Comm	Is_inter	bool flag
MPI_DIMS_CREATE		Compute_dims	void

MPI Function	C++ class	Member name	Return value
MPI_ERRHANDLER_CREATE		<none>	
MPI_ERRHANDLER_FREE	Errhandler	Free	void
MPI_ERRHANDLER_GET		<none>	
MPI_ERRHANDLER_SET		<none>	
MPI_ERROR_CLASS		Get_error_class	int errorclass
MPI_ERROR_STRING		Get_error_string	void
MPI_FINALIZE		Finalize	void
MPI_GATHERV	Intracomm	Gatherv	void
MPI_GATHER	Intracomm	Gather	void
MPI_GET_COUNT	Status	Get_count	int count
MPI_GET_ELEMENTS	Status	Get_elements	int count
MPI_GET_PROCESSOR_NAME		Get_processor_name	void
MPI_GRAPHDIMS_GET	Graphcomm	Get_dims	void
MPI_GRAPH_CREATE	Intracomm	Create_graph	Graphcomm newcomm
MPI_GRAPH_GET	Graphcomm	Get_topo	void
MPI_GRAPH_MAP	Graphcomm	Map	int newrank
MPI_GRAPH_NEIGHBORS_COUNT	Graphcomm	Get_neighbors_count	int nneighbors
MPI_GRAPH_NEIGHBORS	Graphcomm	Get_neighbors	void
MPI_GROUP_COMPARE	Group	static Compare	int result
MPI_GROUP_DIFFERENCE	Group	static Difference	Group newgroup
MPI_GROUP_EXCL	Group	Excl	Group newgroup
MPI_GROUP_FREE	Group	Free	void
MPI_GROUP_INCL	Group	Incl	Group newgroup
MPI_GROUP_INTERSECTION	Group	static Intersect	Group newgroup
MPI_GROUP_RANGE_EXCL	Group	Range_excl	Group newgroup
MPI_GROUP_RANGE_INCL	Group	Range_incl	Group newgroup
MPI_GROUP_RANK	Group	Get_rank	int rank
MPI_GROUP_SIZE	Group	Get_size	int size
MPI_GROUP_TRANSLATE_RANKS	Group	static Translate_ranks	void
MPI_GROUP_UNION	Group	static Union	Group newgroup
MPI_IBSEND	Comm	Ibsend	Request request
MPI_INITIALIZED		Is_initialized	bool flag
MPI_INIT		Init	void
MPI_INTERCOMM_CREATE	Intracomm	Create_intercomm	Intercomm newcomm
MPI_INTERCOMM_MERGE	Intercomm	Merge	Intracomm newcomm
MPI_IPROBE	Comm	Iprobe	bool flag
MPI_IRECV	Comm	Irecv	Request request
MPI_IRSEND	Comm	Irsend	Request request
MPI_ISEND	Comm	Isend	Request request
MPI_ISSEND	Comm	Issend	Request request
MPI_KEYVAL_CREATE		<none>	
MPI_KEYVAL_FREE		<none>	
MPI_OP_CREATE	Op	Init	void
MPI_OP_FREE	Op	Free	void
MPI_PACK_SIZE	Datatype	Pack_size	int size
MPI_PACK	Datatype	Pack	void

MPI Function	C++ class	Member name	Return value
MPI_PCONTROL		Pcontrol	void
MPI_PROBE	Comm	Probe	void
MPI_RECV_INIT	Comm	Recv_init	Prequest request
MPI_RECV	Comm	Recv	void
MPI_REDUCE_SCATTER	Intracomm	Reduce_scatter	void
MPI_REDUCE	Intracomm	Reduce	void
MPI_REQUEST_FREE	Request	Free	void
MPI_RSEND_INIT	Comm	Rsend_init	Prequest request
MPI_RSEND	Comm	Rsend	void
MPI_SCAN	Intracomm	Scan	void
MPI_SCATTERV	Intracomm	Scatterv	void
MPI_SCATTER	Intracomm	Scatter	void
MPI_SENDRECV_REPLACE	Comm	Sendrecv_replace	void
MPI_SENDRECV	Comm	Sendrecv	void
MPI_SEND_INIT	Comm	Send_init	Prequest request
MPI_SEND	Comm	Send	void
MPI_SSEND_INIT	Comm	Ssend_init	Prequest request
MPI_SSEND	Comm	Ssend	void
MPI_STARTALL	Prequest	static Startall	void
MPI_START	Prequest	Start	void
MPI_TESTALL	Request	static Testall	bool flag
MPI_TESTANY	Request	static Testany	bool flag
MPI_TESTSOME	Request	static Testsome	int outcount
MPI_TEST_CANCELLED	Status	Is_cancelled	bool flag
MPI_TEST	Request	Test	bool flag
MPI_TOPO_TEST	Comm	Get_topo	int status
MPI_TYPE_COMMIT	Datatype	Commit	void
MPI_TYPE_CONTIGUOUS	Datatype	Create_contiguous	Datatype
MPI_TYPE_EXTENT		<none>	
MPI_TYPE_FREE	Datatype	Free	void
MPI_TYPE_HINDEXED		<none>	
MPI_TYPE_HVECTOR		<none>	
MPI_TYPE_INDEXED	Datatype	Create_indexed	Datatype
MPI_TYPE_LB		<none>	
MPI_TYPE_SIZE	Datatype	Get_size	int
MPI_TYPE_STRUCT		<none>	
MPI_TYPE_UB		<none>	
MPI_TYPE_VECTOR	Datatype	Create_vector	Datatype
MPI_UNPACK	Datatype	Unpack	void
MPI_WAITALL	Request	static Waitall	void
MPI_WAITANY	Request	static Waitany	int index
MPI_WAITSOME	Request	static Waitsome	int outcount
MPI_WAIT	Request	Wait	void
MPI_WTICK		Wtick	double wtick
MPI_WTIME		Wtime	double wtime

MPI Function Index