MPI: A Message-Passing Interface Standard

Version 3.0

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

March 26, 2011

This document describes the Message-Passing Interface (MPI) standard, version 3.0. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined.

Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1 (June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2 and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008), combining the previous documents. Version MPI-2.2 (September 2009) added additional clarifications and seven new routines. This version, MPI-3.0, is an extension of MPI-2.2.

Chapter 1

Introduction to MPI

1.1 Overview and Goals

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

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

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

A complete list of goals follows.

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

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

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [44], Express [12], nCUBE's Vertex [40], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [47, 48], Chimp [16, 17], PVM [4, 14], Chameleon [25], and PICL [24].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message-Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [55]. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process.

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

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

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

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

1.5 Background of MPI-2.2

MPI-2.2 is a minor update to the MPI-2.1 standard. This version addresses additional errors and ambiguities that were not corrected in the MPI-2.1 standard as well as a small number of extensions to MPI-2.1 that met the following criteria:

- Any correct MPI-2.1 program is a correct MPI-2.2 program.
- Any extension must have significant benefit for users.
- Any extension must not require significant implementation effort. To that end, all such changes are accompanied by an open source implementation.

The discussions of MPI-2.2 proceeded concurrently with the MPI-3 discussions; in some cases, extensions were proposed for MPI-2.2 but were later moved to MPI-3.

1.6 Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. Areas of particular interest are the extension of collective operations to include nonblocking and sparse-group routines and more flexible and powerful one-sided operations. This *draft* contains the MPI Forum's current draft of nonblocking collective routines.

A new Fortran `mpi_f08` module is introduced to provide extended compile-time argument checking and buffer handling in nonblocking routines. The existing `mpi` module provides compile-time argument checking on the basis of existing MPI-2.2 routine definitions. The use of `mpif.h` is strongly discouraged.

MPI_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 MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran, and in C++ should be scoped in the `MPI` namespace, MPI::Action_subset.

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

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

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

2.3 Procedure Specification

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

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

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

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

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

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, language dependent bindings follow:

- The ISO C version of the function.
- The Fortran version of the same function used with `USE mpi` or `INCLUDE 'mpif.h'`
- The Fortran version used with `USE mpi_f08`.
- The C++ binding (which is deprecated).

Fortran in this document refers to Fortran 90 and higher; see Section 2.6.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

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

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

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

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

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

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

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

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

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

2.5 Data Types

2.5.1 Opaque Objects

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

In Fortran with USE mpi or INCLUDE 'mpif.h', all handles have type INTEGER. In Fortran with USE mpi_f08, and in C and C++, a different handle type is defined for each category of objects. With Fortran USE mpi_f08, the handles are defined as Fortran sequenced derived types that consist of only one element INTEGER :: MPI_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and in mpif.h. The names are identical to the names in C, except that they are not case sensitive. For example:

```
TYPE MPI_Comm
  SEQUENCE
  INTEGER   :: MPI_VAL
END TYPE MPI_Comm
```

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.*)

> *Rationale.* Due to the sequence attribute in the definition of handles in the mpi_f08 module, the new Fortran handles are associated with one numerical storage unit, i.e., they have the same C binding as the INTEGER handles of the mpi module. Due to the equivalence of the integer values, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle COMM can be converted directly into an exactly equivalent mpi_f08 communicator handle named comm_f08 by comm_f08%MPI_VAL=COMM, and vice versa. (*End of rationale.*)

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. With the `mpi_f08` module, optional arguments through function overloading are used instead of

MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, and MPI_UNWEIGHTED. The constants MPI_ARGV_NULL and MPI_ARGVS_NULL are not substituted by function overloading.

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, but not necessarily in array declarations or as labels in C/C++ `switch` or Fortran `select/case` statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C/C++ and Fortran. 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 handles themselves are constants and can be also used in initialization expressions or assignments.

The constants that are required to be compile-time constants (and can thus be used for array length declarations and labels in C/C++ `switch` and Fortran `case/select` statements) are:

MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)

and their C++ counterparts where appropriate.

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
MPI_UNWEIGHTED
```

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 with the include file mpif.h or the mpi module, the document uses <type> to represent a choice variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran 2008 syntax TYPE(*), DIMENSION(..); for C and C++, we use void *.

Advice to implementors. The implementor can freely choose how to implement choice arguments in the mpi module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi_f08 module. (*End of advice to implementors.*)

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. These types must have the same width and encode address values in the same manner such that address values in one language may be passed directly to another language without conversion. 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. These types must have the same width and encode address values in the same manner such that offset values in one language may be passed directly to another language without conversion.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) The C++ language bindings have been deprecated. 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 or later, though they were originally designed to be usable in Fortran 77 environments. With the `mpi_f08` module, the two Fortran 2008 features *assumed type* and *assumed rank* are also required, see Section 2.5.5 on page 16.

Since the word `PARAMETER` is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will understand the word "argument" (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

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

2.6.1 Deprecated Names and Functions

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

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

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with

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_function
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_FUNCTION
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_FUNCTION
MPI_ATTR_DELETE	MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI_COMM_SET_ATTR

Table 2.1: Deprecated constructs

most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran 90 or later; it means Fortran 2008 and later if the `mpi_f08` module is used.

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. With `USE mpi_f08`, this last argument is declared as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., MPI_NULL_COPY_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

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

Handles are represented in Fortran as `INTEGER`s, or with the `mpi_f08` module as a derived type, see Section 2.5.1 on page 12. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

The MPI Fortran bindings are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 16.2.2.

2.6.3 C Binding Issues

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

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

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

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

Array arguments are indexed from zero.

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

Choice arguments are pointers of type `void *`.

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

2.6.4 C++ Binding Issues

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

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

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

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

C++ functions that create objects or return information usually place the object or information in the return value. Since the language neutral prototypes of MPI functions operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (myrank = 1) receives this message with the **receive** operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

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

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

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

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

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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

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

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

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

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

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

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

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

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag, MPI::Status& status) const`*(binding deprecated, see Section 15.2)* }

{`void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const`*(binding deprecated, see Section 15.2)* }

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

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

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

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

> *Advice to implementors.* Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.
>
> In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

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

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

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify a unique receiver. This matches a "push" communication mechanism, where data transfer is effected by the sender (rather than a "pull" mechanism, where data transfer is effected by the receiver).

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

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

3.2.5 Return Status

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

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

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

With Fortran `USE mpi_f08`, status is defined as the Fortran derived type `TYPE(MPI_Status)`, which contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the derived type may contain additional fields. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code, respectively, of the received message. Additionally, within the `mpi` and the `mpi_f08` module, both, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and the `TYPE(MPI_Status)` is defined to allow with both modules the conversion between both status declarations.

> *Rationale.* It is allowed to have the same name (e.g., MPI_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

> *Advice to implementors.* The Fortran `TYPE(MPI_Status)` may be defined as a sequence derived type to achieve the same data layout as in C. (*End of advice to implementors.*)

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

{`int MPI::Status::Get_source() const` *(binding deprecated, see Section 15.2)* } {void MPI::Status::Set_source(int source) *(binding deprecated, see Section 15.2)* }

{int MPI::Status::Get_tag() const *(binding deprecated, see Section 15.2)* }

{void MPI::Status::Set_tag(int tag) *(binding deprecated, see Section 15.2)* }

{int MPI::Status::Get_error() const *(binding deprecated, see Section 15.2)* }

{void MPI::Status::Set_error(int error) *(binding deprecated, see Section 15.2)* }

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

> *Rationale.* The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

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

MPI_GET_COUNT(status, datatype, count)

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

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

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

```
MPI_Get_count(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{int MPI::Status::Get_count(const MPI::Datatype& datatype) const *(binding deprecated, see Section 15.2)* }

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

> *Rationale.* Some message-passing libraries use INOUT count, tag and source arguments, thus using them both to specify the selection criteria for incoming

messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.

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

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

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

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

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

3.2.6 Passing MPI_STATUS_IGNORE for Status

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

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

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

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

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

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

With the Fortran bindings through the `mpi_f08` module and the C++ bindings, MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE does not exist. To allow an OUT or INOUT `TYPE(MPI_Status)` or `MPI::Status` argument to be ignored, all MPI `mpi_f08` or C++ bindings that have OUT or INOUT `TYPE(MPI_Status)` or `MPI::Status` parameters are overloaded with a second version that omits the OUT or INOUT `TYPE(MPI_Status)` or `MPI::Status` parameter.

Example 3.1 The `mpi_f08` bindings for MPI_PROBE are:

```
SUBROUTINE MPI_Probe(source, tag, comm, status, ierror)
     INTEGER, INTENT(IN) ::  source, tag
     TYPE(MPI_Comm), INTENT(IN) ::  comm
     TYPE(MPI_Status), INTENT(OUT) ::  status
     INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
     END SUBROUTINE
SUBROUTINE MPI_Probe(source, tag, comm, ierror)
     INTEGER, INTENT(IN) ::  source, tag
     TYPE(MPI_Comm), INTENT(IN) ::  comm
     INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
     END SUBROUTINE
```

Example 3.2 The C++ bindings for MPI_PROBE are:

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

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

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

1. Data is pulled out of the send buffer and a message is assembled.

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`MPI::Request MPI::Comm::Irecv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const` *(binding deprecated, see Section 15.2)* }

Start a nonblocking receive.

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization and Temporary Memory Modifications" in Section 16.2.2 on pages 545 and 549. (*End of advice to users.*)

3.7.3 Communication Completion

The functions MPI_WAIT and MPI_TEST are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a synchronous mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A **null** handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return tag

3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

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

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

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

```
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const` *(binding deprecated, see Section 15.2)* }

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

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

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

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

```
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const` *(binding deprecated, see Section 15.2)* }

Creates a persistent communication request for a buffered mode send.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

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

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
```

where $*$ indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization and Temporary Memory Modifications" in Section 16.2.2 on pages 545 and 549. (*End of advice to users.*)

3.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

> *Advice to users.* C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such machines as well. (*End of advice to users.*)

> *Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization and Temporary Memory Modifications" in Section 16.2.2 on pages 545 and 549. (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

```
MPI_Type_size(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`int MPI::Datatype::Get_size() const` *(binding deprecated, see Section 15.2)* }

MPI_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 106. This allows one to define a datatype that has "holes" at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(oldtype, newtype)

IN	type	datatype (handle)
OUT	newtype	copy of oldtype (handle)

```
int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

```
MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{MPI::Datatype MPI::Datatype::Dup() const *(binding deprecated, see Section 15.2)* }

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 6.7.4 on page 275. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype.

4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype , ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

and extent *extent*. (Empty entries of “pseudo-type” MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) The send operation sends $n \cdot$ count

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(user_fn, commute, op)

IN	user_fn	user defined function (function)
IN	commute	true if commutative; false otherwise.
OUT	op	operation (handle)

```
int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)
```

```
MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
    EXTERNAL USER_FN
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

```
MPI_Op_create(user_fn, commute, op, ierror)
    EXTERNAL :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{void MPI::Op::Init(MPI::User_function* user_fn, bool commute) *(binding deprecated, see Section 15.2)* }

MPI_OP_CREATE binds a user-defined reduction operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument user_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len and datatype.

The ISO C prototype for the function is the following.

```
typedef void MPI_User_function(void* invec, void* inoutvec, int* len,
             MPI_Datatype* datatype);
```

The Fortran declaration of the user-defined function appears below.

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
```

```
INTEGER LEN, TYPE
```

The C++ declaration of the user-defined function appears below.

{`typedef void MPI::User_function(const void* invec, void* inoutvec, int len, const Datatype& datatype);` *(binding deprecated, see Section 15.2)*}

The datatype argument is a handle to the data type that was passed into the call to MPI_REDUCE. The user reduce function should be written such that the following holds: Let u[0], ... , u[len-1] be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let v[0], ... , v[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let w[0], ... , w[len-1] be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then w[i] = u[i]∘v[i], for i=0 , ... , len-1, where ∘ is the reduce operation that the function computes.

Informally, we can think of invec and inoutvec as arrays of len elements that user_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i] ∘ inoutvec[i], for $\mathsf{i} = 0, \ldots, \mathsf{count} - 1$, where ∘ is the combining operation computed by the function.

Rationale. The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle.

MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)

IN	inbuf	input buffer (choice)
INOUT	inoutbuf	combined input and output buffer (choice)
IN	count	number of elements in inbuf and inoutbuf buffers (non-negative integer)
IN	datatype	data type of elements of inbuf and inoutbuf buffers (handle)
IN	op	operation (handle)

```
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
              MPI_Datatype datatype, MPI_Op op)
```

```
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR
```

```
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(..)  ::  inbuf, inoutbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{void MPI::Op::Reduce_local(const void* inbuf, void* inoutbuf, int count, const MPI::Datatype& datatype) const *(binding deprecated, see Section 15.2)* }

The function applies the operation given by op element-wise to the elements of inbuf and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the same number of elements given by count and the same datatype given by datatype. The MPI_IN_PLACE option is not allowed.

Reduction operations can be queried for their commutativity.

MPI_OP_COMMUTATIVE(op, commute)

IN	op	operation (handle)
OUT	commute	`true` if op is commutative, `false` otherwise (logical)

```
int MPI_Op_commutative(MPI_Op op, int *commute)
```

```
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

```
MPI_Op_commutative(op, commute, ierror)
    TYPE(MPI_Op), INTENT(IN) ::  op
    LOGICAL, INTENT(OUT) ::  commute
```

no pending communication on peer_comm that could interfere with this communication.

> *Advice to users.* We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (*End of advice to users.*)

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN	intercomm	Inter-Communicator (handle)
IN	high	(logical)
OUT	newintracomm	new intra-communicator (handle)

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
             MPI_Comm *newintracomm)
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
    INTEGER NEWINTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
```

```
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
    LOGICAL, INTENT(IN) :: high
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{MPI::Intracomm MPI::Intercomm::Merge(bool high) const*(binding deprecated, see Section 15.2)* }

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the “low” group before the “high” group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

> *Advice to implementors.* The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

- obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoid problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency "hit" inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

IN	comm_copy_attr_fn	copy callback function for comm_keyval (function)
IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
              MPI_Comm_delete_attr_function *comm_delete_attr_fn,
              int *comm_keyval, void *extra_state)
```

```
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
              EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
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, INTENT(OUT) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{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) *(binding deprecated, see Section 15.2)* }

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
             void *attribute_val, void *extra_state);
```

which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.

The Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(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_FUNCTION(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); *(binding deprecated, see Section 15.2)*}

and

{typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm, int comm_keyval, void* attribute_val, void* extra_state); *(binding deprecated, see Section 15.2)*}

The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns flag = 0, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1), the new attribute value is set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_DUP will fail).

The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C, C++, or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

Advice to users. Even though both formal arguments attribute_val_in and attribute_val_out are of type void *, their usage differs. The C copy function is passed by MPI in attribute_val_in the *value* of the attribute, and in attribute_val_out the *address* of the attribute, so as to allow the function to return the (new) attribute value. The use of type void * for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

Advice to implementors. To be able to use the predefined C functions MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN as comm_copy_attr_fn argument and/or MPI_COMM_NULL_DELETE_FN as the comm_delete_attr_fn argument in a call to the C++ routine MPI::Comm::Create_keyval, this routine may be overloaded with 3 additional routines that accept the C functions as the first, the second, or both input arguments (instead of an argument that matches the C++ prototype). (*End of advice to implementors.*)

Advice to users. If a user wants to write a "wrapper" routine that internally calls MPI::Comm::Create_keyval and comm_copy_attr_fn and/or comm_delete_attr_fn are arguments of this wrapper routine, and if this wrapper routine should be callable with both user-defined C++ copy and delete functions and with the predefined C functions, then the same overloading as described above in the advice to implementors may be necessary. (*End of advice to users.*)

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

MPI_Comm_free_keyval(comm_keyval, ierror)
    INTEGER, INTENT(INOUT) ::  comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`static void MPI::Comm::Free_keyval(int& comm_keyval)` *(binding deprecated, see Section 15.2)* }

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explictly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the new communicator-specific creation function. The use of MPI_KEYVAL_FREE is deprecated.

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT	comm	communicator from which attribute will be attached (handle)
IN	comm_keyval	key value (integer)
IN	attribute_val	attribute value

6.7.3 Windows

The new functions for caching on windows are:

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for win_keyval (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
              MPI_Win_delete_attr_function *win_delete_attr_fn,
              int *win_keyval, void *extra_state)
```

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
              EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
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, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{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) *(binding deprecated, see Section 15.2)* }

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_FUNCTION(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_FUNCTION(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); (binding deprecated, see
             Section 15.2)}
```

and

```
{typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
             void* attribute_val, void* extra_state); (binding deprecated, see
             Section 15.2)}
```

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is erroneous.

MPI_WIN_FREE_KEYVAL(win_keyval)

INOUT	win_keyval	key value (integer)

```
int MPI_Win_free_keyval(int *win_keyval)
```

```
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
```

```
MPI_Win_free_keyval(win_keyval, ierror)
    INTEGER, INTENT(INOUT) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
{static void MPI::Win::Free_keyval(int& win_keyval)(binding deprecated, see
             Section 15.2) }
```

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

MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

{void MPI::Win::Delete_attr(int win_keyval) (binding deprecated, see Section 15.2)
             }
```

6.7.4 Datatypes

The new functions for caching on datatypes are:

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

IN	type_copy_attr_fn	copy callback function for type_keyval (function)
IN	type_delete_attr_fn	delete callback function for type_keyval (function)
OUT	type_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
             MPI_Type_delete_attr_function *type_delete_attr_fn,
             int *type_keyval, void *extra_state)

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

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, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

{static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
             type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
```

```
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

```
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const *(binding deprecated, see Section 15.2)* }

MPI_TYPE_DELETE_ATTR(datatype, type_keyval)

INOUT	datatype	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)

```
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
```

```
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
```

```
MPI_Type_delete_attr(datatype, type_keyval, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{void MPI::Datatype::Delete_attr(int type_keyval) *(binding deprecated, see Section 15.2)* }

6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last three are included because keyval is an argument to the copy and delete functions for attributes.

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style 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 (datatype, type_name)

INOUT	datatype	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 datatype, char *type_name)
```

```
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```

```
MPI_Type_set_name(datatype, type_name, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    CHARACTER(LEN=*), INTENT(IN) :: type_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void MPI::Datatype::Set_name(const char* type_name)` *(binding deprecated, see Section 15.2)* }

MPI_TYPE_GET_NAME (datatype, type_name, resultlen)

IN	datatype	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 datatype, char *type_name, int
             *resultlen)

MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER DATATYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME

MPI_Type_get_name(datatype, type_name, resultlen, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=*), INTENT(OUT) ::  type_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`void MPI::Datatype::Get_name(char* type_name, int& resultlen) const` *(binding deprecated, see Section 15.2)* }

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

MPI_Win_set_name(win, win_name, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=*), INTENT(IN) ::  win_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`void MPI::Win::Set_name(const char* win_name)` *(binding deprecated, see Section 15.2)* }

Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and run-time ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C and C++,

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 2
```

in Fortran,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
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(version, subversion, ierror)
```

```
{void MPI::Get_processor_name(char* name, int& resultlen)
```
(binding deprecated, see Section 15.2) }

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.

> *Rationale.* This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

> *Advice to users.* The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead per message buffered by a call to MPI_BSEND (see Section 3.6.1).

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory (see Section 11.4.3.)

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative 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
```

```
MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)` *(binding deprecated, see Section 15.2)* }

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
```

```
MPI_Free_mem(base, ierror)
    USE, INTRINSIC :: ISO_C_BINDING
    TYPE(C_PTR), INTENT(IN) :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void MPI::Free_mem(void *base)` *(binding deprecated, see Section 15.2)* }

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

> *Rationale.* The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the `malloc` and `free` C library calls: a call to MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (*End of rationale.*)

Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C `malloc` and `free` functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes `malloc`, and MPI_FREE_MEM invokes `free`.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

Example 8.1

Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALs, and assume that pointers are address-sized.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran compilers for Intel) do not support this code.

Example 8.2 Same example, in C

```
float  (* f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an **MPI exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.

> *Advice to implementors.* High-quality implementation should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

8.3.1 Error Handlers for Communicators

MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)

IN	comm_errhandler_fn	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
              *comm_errhandler_fn, MPI_Errhandler *errhandler)

MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR

MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
    EXTERNAL :: comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

{static MPI::Errhandler
              MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_function*
              comm_errhandler_fn) (binding deprecated, see Section 15.2) }
```

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_function, which is defined as

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "stdargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. This typedef replaces MPI_Handler_function, whose use is deprecated.

In Fortran, the user routine should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
    INTEGER COMM, ERROR_CODE
```

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

{`typedef void MPI::Comm::Errhandler_function(MPI::Comm &, int *, ...);` *(binding deprecated, see Section 15.2)*}

Rationale. The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT	comm	communicator (handle)
IN	errhandler	new error handler for communicator (handle)

```
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

```
MPI_Comm_set_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler)` *(binding deprecated, see Section 15.2)* }

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_Comm_get_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{MPI::Errhandler MPI::Comm::Get_errhandler() const *(binding deprecated, see Section 15.2)* }

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)

IN	win_errhandler_fn	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_Win_create_errhandler(MPI_Win_errhandler_function
              *win_errhandler_fn, MPI_Errhandler *errhandler)

MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL WIN_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR

MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
    EXTERNAL :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{static MPI::Errhandler
MPI::Win::Create_errhandler(MPI::Win::Errhandler_function* win_errhandler_fn) *(binding deprecated, see Section 15.2)* }

Creates an error handler that can be attached to a window object. The user routine should be, in C, a function of type MPI_Win_errhandler_function which is defined as

```
typedef void MPI_Win_errhandler_function(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_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

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

{`typedef void MPI::Win::Errhandler_function(MPI::Win &, int *, ...);` *(binding deprecated, see Section 15.2)*}

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
```

```
MPI_Win_set_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)` *(binding deprecated, see Section 15.2)* }

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_Win_get_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{MPI::Errhandler MPI::Win::Get_errhandler() const *(binding deprecated, see Section 15.2)* }

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)

IN	file_errhandler_fn	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_File_create_errhandler(MPI_File_errhandler_function
              *file_errhandler_fn, MPI_Errhandler *errhandler)
```

```
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL FILE_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

```
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
    EXTERNAL :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{static MPI::Errhandler
  MPI::File::Create_errhandler(MPI::File::Errhandler_function*
  file_errhandler_fn) *(binding deprecated, see Section 15.2)* }

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_function, which is defined as

```
typedef void MPI_File_errhandler_function(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_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

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

{typedef void MPI::File::Errhandler_function(MPI::File &, int *, ...);
  (binding deprecated, see Section 15.2)}

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)
```

Chapter 9

The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI_Info in C and Fortran with the mpi_f08 module, MPI::Info in C++, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can 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.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

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 nonblocking 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

```
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_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
             array_of_errcodes, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  command, argv(*)
    INTEGER, INTENT(IN) ::  maxprocs, root
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  intercomm
    INTEGER, INTENT(OUT) ::  array_of_errcodes(*) !  optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  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`*(binding deprecated, see Section 15.2)* }

{`MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root) const`*(binding deprecated, see Section 15.2)* }

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 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.*)

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 and Fortran with the `mpi_f08` module, MPI::Info in C++ and `INTEGER` in Fortran with the `mpi` module or the include file `mpif.h`. 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 9 on page 349.

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 10.3.4 on page 366). 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 ($0 \leq m <$ 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 in the Fortran `mpi` module or `mpif.h` include file, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. Inthe Fortran `mpi_f08` module or in C++ this constant does not exist, and the array_of_errcodes argument may be omitted from the argument list.

Advice to implementors. In the Fortran mpi module or mpif.h include file, MPI_ERRCODES_IGNORE is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4 on page 15. In the Fortran `mpi_f08` module, the optional argument has to be implemented through function overloading. See the discussion in Section 2.5.2 on page 14. (*End of advice to implementors.*)

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.*)

11.7.3 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 16.2.2.

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 in modules or `COMMON` blocks, or to variables that were declared `VOLATILE` (but this attribute may inhibit optimization of any code containing the RMA window). Further details and additional solutions are discussed in Section 16.2.2, "A Problem with Register Optimization and Temporary Memory Modifications," on page 549. See also, "Problems Due to Data Copying and Sequence Association," on page 545, for additional Fortran problems.

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
```

```
MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
             ierror)
    EXTERNAL :: query_fn, free_fn, cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
{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) (binding deprecated, see Section 15.2) }
```

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; extra_state 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);` *(binding deprecated, see Section 15.2)*}

The 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).

The 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 or has omitted the status argument (with the `mpi_f08` Fortran module or C++), 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);` *(binding deprecated, see Section 15.2)*}

The free_fn function is invoked to clean up user-allocated resources when the generalized request is freed.

The 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.

The 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 in which 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); *(binding deprecated, see Section 15.2)*}

The cancel_fn function is invoked to start the cancelation of a generalized request. It is called by MPI_CANCEL(request). MPI passes complete=true to the callback function 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 an 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 or the status argument was omitted, 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

MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

{`void MPI::Grequest::Complete()` *(binding deprecated, see Section 15.2)* }

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.*)

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 13.4.5, page 480).

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 13.6.4, page 502, 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 communication functions; see Section 3.2.2 on page 29 and Section 4.1.11 on page 111. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 456). 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 and Temporary Memory Modifications" in Section 16.2.2, pages 545 and 549. (*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 with the Fortran mpi module or mpif.h include file) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. With the Fortran mpi_f08 module or 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).

13.4.2 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
```

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- 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 and Temporary Memory Modifications," Section 16.2.2, page 549.
- 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)

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.*)

Table 13.2 specifies the sizes of predefined datatypes in "external32" format.

13.5.3 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
```

```
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
             dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    EXTERNAL :: read_conversion_fn, write_conversion_fn,
    dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  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) (binding deprecated, see Section 15.2) }
```

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 13.7, page 508). 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 void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
             MPI::Aint& file_extent, void* extra_state); (binding deprecated,
             see Section 15.2)}
```

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 void MPI::Datarep_conversion_function(void* userbuf,
              MPI::Datatype& datatype, int count, void* filebuf,
              MPI::Offset position, void* extra_state);
```
(binding deprecated, see Section 15.2)}

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 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, 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, and the userbuf pointer will be unchanged.

> *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.*)

13.6.7 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 with kind parameter MPI_OFFSET_KIND, which is defined in the `mpi_f08` module, the `mpi` module and the `mpif.h` include file.

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 16.3, page 569).

13.6.8 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 13.2.8, page 453).

13.6.9 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 13.6.1, page 498, 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.

MPI_TYPE_UB(datatype, displacement)

IN	datatype	datatype (handle)
OUT	displacement	displacement of upper bound from origin, in bytes (integer)

```
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

```
MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_CREATE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 16.3.7 on page 579. The language bindings are modified.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
             *delete_fn, int *keyval, void* extra_state)
```

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR
```

The copy_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)
```

A Fortran declaration for such a function is as follows:

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

copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_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. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.

Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows:

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

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

delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN; MPI_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.

The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_KEYVAL_FREE(keyval)

INOUT	keyval	Frees the integer key value (integer)

```
int MPI_Keyval_free(int *keyval)
```

```
MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_PUT(comm, keyval, attribute_val)

INOUT	comm	communicator to which attribute will be attached (handle)
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)
IN	attribute_val	attribute value

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

```
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_GET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

IN	comm	communicator to which attribute is attached (handle)
IN	keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	`true` if an attribute value was extracted; `false` if no attribute is associated with the key

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

```
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG
```

The following function is deprecated and is superseded by MPI_COMM_DELETE_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_DELETE(comm, keyval)

INOUT	comm	communicator to which attribute is attached (handle)
IN	keyval	The key value of the deleted attribute (integer)

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_CREATE_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ERRHANDLER_CREATE(handler_fn, errhandler)

IN	handler_fn	user defined error handling procedure
OUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_create(MPI_Handler_function *handler_fn,
             MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_CREATE(HANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL HANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

Register the user routine handler_fn for use as an MPI exception handler. Returns in errhandler a handle to the registered exception handler.

In the C language, the user routine should be a C function of type MPI_Handler_function, which is defined as:

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use, the second is the error code to be returned.

In the Fortran language, the user routine should be of the form:

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

The following function is deprecated and is superseded by MPI_COMM_SET_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ERRHANDLER_SET(comm, errhandler)

INOUT	comm	communicator to set the error handler for (handle)
IN	errhandler	new MPI error handler for communicator (handle)

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

Associates the new error handler errorhandler with communicator comm at the calling process. Note that an error handler is always associated with the communicator.

The following function is deprecated and is superseded by MPI_COMM_GET_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ERRHANDLER_GET(comm, errhandler)

IN	comm	communicator to get the error handler from (handle)
OUT	errhandler	MPI error handler currently associated with communicator (handle)

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

Returns in errhandler (a handle to) the error handler that is currently associated with communicator comm.

15.2 Deprecated since MPI-2.2

The entire set of C++ language bindings have been deprecated.

Example 16.10 `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.*)

16.2 Fortran Support

16.2.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard (and later).

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. In Fortran 2008, the only new language features used, are of assumed type and assumed rank. They were defined to allow the definition of choice arguments as part of the Fortran language. (*End of rationale.*)

MPI defines three methods of Fortran support:

1. **INCULDE 'mpif.h'** This method is described in Section 16.2.3. The use of the include file `mpif.h` is strongly discouraged since MPI-3.0.
2. **USE mpi** This method is described in Section 16.2.4 and requires compile-time argument checking.
3. **USE mpi_f08** This method is described in Section 16.2.5 and requires compile-time argument checking that includes also unique handle types.

A compliant MPI-3 implementation providing a Fortran interface must provide all three Fortran support methods.

Application subroutines and functions may use either one of the modules or the `mpif.h` include file. An implementation may require use of the module to prevent type mismatch errors.

Advice to users. It is recommended to use one of the MPI modules 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.*)

In a single application, it must be possible to link together routines some of which `USE mpi` and others of which `USE mpi_f08` or `INCLUDE mpif.h`.

The `INTEGER` compile-time constant MPI_SUBARRAYS is MPI_SUBARRAYS_SUPPORTED if all choice arguments are defined in explicit interfaces with standardized assumed type and assumed rank, otherwise it equals MPI_SUBARRAYS_UNSUPPORTED. This constant exists with each Fortran support method, but not in the C/C++ header files. The value may be different for each Fortran support method.

Section 16.2.2 gives an overview on known problems when using Fortran together with MPI. Section 16.2.6 and Section 16.2.7 describe additional functionality that is part of the Fortran support. funcMPI_F_SYNC_REG is needed for one of the methods to prevent register optimization problems. A set of functions provides additional support for Fortran intrinsic numeric types, including parameterized types. The functions are: 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.

16.2.2 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.

The following MPI features are inconsistent with Fortran 90.

1. An MPI subroutine with a choice argument may be called with different argument types. Using the `mpi_f08` module, this problem is resolved.
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_UNWEIGHTED, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 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.

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

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, `INTEGER*8` or `INTEGER` should be used instead.

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 on page 17 and Section 4.1.1 on page 87 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. Similar to C, with Fortran 2008 and later together with the `mpi_f08` module, the problem is avoided by declaring choice arguments with `TYPE(*), DIMENSION(..)`, i.e., as assumed type and assumed rank dummy arguments.

Using `INCLUDE mpif.h`, the following code fragment might technically be invalid 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. Using the `mpi_f08` or `mpi` module, the problem is usually resolved through the standardized assume-type and assume-rank declarations of the dummy arguments, or with non-standard Fortran options preventing type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument. Thus, when using the `mpi_f08` or `mpi` module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays `INTEGER, DIMS(*)` and `LOGICAL, PERIODS(*)`.

```
USE mpi_f08
INTEGER size
CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
```

Using INCLUDE 'mpif.h', compiler warnings are not expected except if this include file also uses Fortran explicit interfaces.

Problems Due to Data Copying and Sequence Association

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_SUPPORTED:

  Choice buffer arguments are declared as TYPE(*), DIMENSION(..). For example, considering the following code fragment:

  ```
  REAL s(100), r(100)
  CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
  CALL MPI_Wait(rq, status, ierror)
  CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
  CALL MPI_Wait(rq, status, ierror)
  ```

  In this case, the individual elements s(1), s(6), s(11), etc. are sent between the start of MPI_ISEND and the end of MPI_WAIT even though the compiled code may not copy s(1:100:5) to a contiguous temporary scratch buffer. Instead, the compiled code may pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6), s(11), ..., s(96).

  All nonblocking MPI communication functions behave as if the user-specified elements of choice buffers are copied to a contiguous scratch buffer in the MPI runtime environment. All datatype descriptions (in the example above, "3, MPI_REAL") read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a corresponding MPI_Request returns), it is as if the received data has been copied from the virtual contiguous scratch buffer back to the non-contiguous application buffer. In the example above, r(1), r(6), and r(11) will be filled with the received data when MPI_WAIT returns.

  Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION(..) arguments contains enough information that the MPI library can make a real contiguous copy of non-contiguous user buffers. Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

  Rationale. If MPI_SUBARRAYS equals MPI_SUBARRAYS_SUPPORTED, non-contiguous buffers are handled inside of the MPI library instead of by the compiled user code. Therefore the scope of scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. If MPI_SUBARRAYS equals MPI_SUBARRAYS_UNSUPPORTED, such scratch buffers can be organized only by the compiler for the duration of the nonblocking call, which is too short for implementing the whole MPI operation. (*End of rationale.*)

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_UNSUPPORTED:

  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 nonblocking 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

[1] Technically, the Fortran standards are worded to allow non-contiguous storage of any array data.

```
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

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 nonblocking 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, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4 on page 15. 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

[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.

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). With `USE mpi_f08`, the attributes `INTENT(IN)`, `INTENT(OUT)`, and `INTENT(INOUT)` are used in the Fortran interface. In most cases `INTENT(IN)` is used if the C interface uses call-by-value. For all buffer arguments and for `OUT` dummy arguments that allow one of these special constants as input, an `INTENT(...)` is not specified.

Fortran Derived Types

MPI does explicitly support passing Fortran derived types to choice dummy arguments, but does not support Fortran non-sequence derived types.

The following code fragment shows one possible way to send a sequence derived type in Fortran. The example assumes that all data is passed by address.

```
type mytype
   SEQUENCE
   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)
```

```
call MPI_SEND(foo%i, 1, newtype, ...)
```

A Problem with Register Optimization and Temporary Memory Modifications

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 and problems with temporary memory modifications. These problems are independent of the Fortran support method, i.e., they occur with the `mpi_f08` module, the `mpi` module, and the `mif.h` include file.

This section shows four problematic usage areas (the abbrevations in parentheses are used in the table below):

- Usage of nonblocking routines *(Nonbl.)*.
- Usage of one-sided routines *(1-sided)*.
- Usage of MPI parallel file I/O split collective operations *(Split)*.
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between to variables in such datatypes *(Bottom)*.

The compiler is allowed to cause two optimization problems

- Register optimization problems and code movements *(Regis.)*.
- Temporary memory modifications *(Memory)*.

The optimization problems can occur not in all usage areas:

	Nonbl.	1-sided	Split	Bottom
Register	occurs	occurs	-not-	occurs
Memory	occurs	occurs	occurs	-not-

The application writer has several methods to circumvent parts of these problems with special declarations for the used send and receive buffers:

- Usage of the Fortran `ASYNCHRONOUS` attribute.
- Usage of the Fortran `TARGET` attribute.
- Usage of the helper routine MPI_F_SYNC_REG or a user-written dummy routine `DD(buf)`.
- Declaring the buffer as Fortran module data or within a Fortran common block.
- Usage of the Fortran `VOLATILE` attribute.

Each of these methods may solve only a subset of the problems, may have more or less performance drawbacks, and may be usable not in each application context. The following table shows the usability of each method:

	Nonbl. Regis.	Nonbl. Memory	1-sided Regis.	1-sided Memory	Split Memo.	Bottom Regis.	overhead may be
Examples	16.11, 16.12	16.13	Section 11.7.3			16.14, 16.15	
`ASYNCHRONOUS`	solved	solved	may be	may be	solved	may be	medium
`TARGET`	solved	NOT s.	solved	NOT s.	NOT s.	solved	low-med
MPI_F_SYNC_REG	solved	NOT s.	solved	NOT s.	NOT s.	solved	low
Module Data	solved	NOT s.	solved	NOT s.	NOT s.	solved	low-med
`VOLATILE`	solved	solved	solved	solved	solved	solved	high

The next paragraphs describe the problems in detail.

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.

Example 16.11 shows extreme, but allowed, possibilities.

Example 16.11 Fortran 90 register optimization – extreme.

Source	compiled as	or compiled as
`REAL :: buf, b1`	`REAL :: buf, b1`	`REAL :: buf, b1`
`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.

Due to allowed code movement, the content of buf may be already overwritten when sending of the content of buf is executed. The code movement is permitted, because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

Note, that code movement can also be executed across subroutine boundaries when subroutines or functions are inlined.

This register optimization / code movement problem does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as actual argument.

Example 16.12 Similar example with MPI_ISEND

Source	compiled as	or compiled as
`REAL :: buf, copy`	`REAL :: buf, copy`	`REAL :: buf, copy`
`buf = val`	`buf = val`	`buf = val`
`call MPI_ISEND(buf,..req)`	`call MPI_ISEND(buf,..req)`	`addr = &buf`
`copy = buf`	`copy=buf`	`copy = val`
	`buf = val_overwrite`	`buf = val_overwrite`
`call MPI_WAIT(req,..)`	`call MPI_WAIT(req,..)`	`send(*addr)`
`buf = val_overwrite`		

Nonblocking operations and temporary memory modifications. The compiler is allowed to modify temporarily data in the memory. Normally, this problem may occur only if overlapping communication and computation. Example 16.13 shows a possibility.

Example 16.13 Overlapping Communication and Computation

```
USE mpi_f08
REAL :: buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
DO j=1,100
  DO i=2,100
    buf(i,j)=....
  END DO
END DO
CALL MPI_Wait(req,...)
```

The compiler may substitute the nested loops through loop fusion by

```
EQUIVALENCE (buf(1,1), buf_1dim(1))
DO h=1,100
  tmp(h)=buf(1,h)
END DO
DO j=1,10000
  buf_1dim(h)=...
END DO
DO h=1,100
  buf(1,h)=tmp(h)
END DO
```

In the substitution of Example 16.13, buf_1dim(10000) is the 1-dimensional equivalence of buf(100,100). The nonblocking receive may receive the data in the boundary buf(1,1:100) while the fused loop is using temporarily this part of the buffer. When the tmp data is written back to buf, the old data is restored and the received data is lost.

Note, that this problem occurs also

- with one-sided communication with the local buffer at the origin process between an RMA call and the ensuing synchronization call

- and with the window buffer at the target process between two ensuing synchronization calls,
- and also with MPI parallel file I/O split collective operations with the local buffer between the ..._BEGIN and ..._END call.

This type of compiler optimization can be prevented when buf is declared with the Fortran attribute ASYNCHRONOUS:

```
REAL, ASYNCHRONOUS :: buf(100,100)
```

One-sided communication. An example with instruction reordering due to register optimization can be found in Section 11.7.3 on page 425.

One-sided communication. 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.

Example 16.14 shows what Fortran compilers are allowed to do.

Example 16.14 Fortran 90 register optimization.

This source ...	can be compiled as:
`call MPI_GET_ADDRESS(buf,bufaddr, ierror)`	`call MPI_GET_ADDRESS(buf,...)`
`call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr, MPI_REAL,type,ierror)`	`call MPI_TYPE_CREATE_STRUCT(...)`
`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.

Several successive assigments to the same variable can be combined in this way, but only the last assignment is executed. Successive means that no interfering read access to this variable is in between. The compiler cannot detect that the call to MPI_SEND statement is interfering, because the read access to buf is hidden by the usage of MPI_BOTTOM.

Example 16.15 Similar example with MPI_SEND

This source ...	can be compiled as:
`! buf contains val_old`	`! buf contains val_old`
`buf = val_new`	`! dead code:`
	`!   buf=val_new is removed`
`call MPI_SEND(MPI_BOTTOM,1,type,...)`	`call MPI_SEND(...)`
`! with buf as a displacement in type`	`! i.e. val_old is sent`
`buf = val_overwrite`	`buf = val_overwrite`

Solutions. The following paragraphs show in detail how these problems can be solved in portabel way. Several solutions are presented, because all of these solutions have different implication on the performance. Only one solution (with VOLATILE) solves all problems, but it may have the most negative impact on the performance.

Fortran ASYNCHRONOUS attribute. Declaring a buffer with the Fortran ASYNCHRONOUS attribute in a scoping unit (or BLOCK) tells the compiler that any statement of the scoping unit may be executed while the buffer is affected by a pending asynchronous input/output operation. Each library call (e.g., to an MPI routine) within the scoping unit may contain a Fortran asynchronous I/O statement, e.g., the Fortran WAIT statement.

- In the case of nonblocking MPI communication, the send and receive buffers should be declared with the Fortran ASYNCHRONOUS attribute within each scoping unit (or BLOCK) where the buffers are declared and statements are executed between the start (e.g., MPI_IRECV) and completion (e.g., MPI_WAIT) of the nonblocking communication. Declaring REAL, ASYNCHRONOUS :: buf in Examples 16.11 and 16.12, and REAL, ASYNCHRONOUS :: buf(100,100) in Examples 16.13 solves the register optimization and temporary memory modification problem.

  Rationale. A combination of a nonblocking MPI communication call with a buffer in the argument list together with a subsequent call to MPI_WAIT or MPI_TEST is similar to the combination a of Fortran asynchronous read or write together with the matching Fortran wait statement. To prevent incorrect register optimizations or code movement, the Fortran standard requires in the case of Fortran IO, that the ASYNCHRONOUS attribute is defined for the buffer. The ASYNCHONOUS attribute also works with the asynchronous MPI routines because the compiler must expect that inside of the MPI routines such Fortran asynchronous read, write, or wait routines may be called. (*End of rationale.*)

- In the Examples 16.14 and 16.15 and also in the example in Section 11.7.3 on page 425, the ASYNCHRONOUS attribute may also help but the help is not guaranteed because there is not an IO counterpart to the MPI usage.

  Rationale. In case of using MPI_BOTTOM or one-sided synchronizations (e.g., MPI_WIN_FENCE), the buffer is not specified, i.e., those calls can include only a Fortran WAIT statement (or another routine that finishes an asynchronous IO).

Additionally, with Fortran asynchronous IO, it is a clear and forbidden race-condition when storing new data into the buffer while an asynchronous IO is active. Exactly this storing of data into the buffer is done in Example 16.14 when there would have been an initialization `buf=val_init` prior to the call to MPI_RECV, or in Example 16.15, the statement `buf=val_new`. (*End of rationale.*)

Fortran TARGET attribute. Declaring a buffer with the Fortran `TARGET` attribute in a scoping unit (or `BLOCK`) tells the compiler that any statement of the scoping unit may be executed while some pointer to the buffer exist. Calling a library routine (e.g., an MPI routine) may imply that such a pointer is used to modify the buffer.

- The `TARGET` attribute solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles. Declaring `REAL, TARGET` :: buf solves the register optimization problem in Examples 16.11, 16.12, 16.14, and 16.15
- Unfortunately, the TARGET attribute has **not** any impact on problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication, i.e., problems through temporary memory modifications are not solved. Example 16.13 can **not** be solved with the `TARGET` attribute.

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. The MPI library provides MPI_F_SYNC_REG for this purpose, see Section 16.2.6 on page 560.

- Examples 16.11 and 16.12 can be solved by calling MPI_F_SYNC_REG(buf) once directly after MPI_WAIT.

First example	Second example
`call MPI_IRECV(buf,..req)`	`buf = val`
	`call MPI_ISEND(buf,..req)`
	`copy = buf`
`call MPI_WAIT(req,..)`	`call MPI_WAIT(req,..)`
`call MPI_F_SYNC_REG(buf)`	`call MPI_F_SYNC_REG(buf)`
`b1 = buf`	`buf = val_overwrite`

The call MPI_F_SYNC_REG(buf) prevents moving the last line before the MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed, because it is still correct if the additional read access `copy=buf` is moved behind MPI_WAIT and before `buf=val_overwrite`.

- Examples 16.14 and 16.15 can be solved with two additional `call MPI_F_SYNC_REG(buf)`, one directly before MPI_RECV/MPI_SEND, and one directly after this communication operation.

First example	Second example
`call MPI_F_SYNC_REG(buf)`	`call MPI_F_SYNC_REG(buf)`
`call MPI_RECV(MPI_BOTTOM,...)`	`call MPI_SEND(MPI_BOTTOM,...)`
`call MPI_F_SYNC_REG(buf)`	`call MPI_F_SYNC_REG(buf)`

The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND, and the second call is needed to assure that the subsequent access to buf are not moved before MPI_RECV/SEND.

- In the example in Section 16.2.6 on page 560, two asynchronous accesses must be protected: In Process 1, the access to bbbb must be protected similar to Example 16.11, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer, i.e., before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

Source of Process 1	**Source of Process 2**
`bbbb = 777`	`buff = 999`
	`call MPI_F_SYNC_REG(buff)`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`
`call MPI_PUT(bbbb`	
`into buff of process 2)`	
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`
`call MPI_F_SYNC_REG(bbbb)`	`call MPI_F_SYNC_REG(buff)`
	`ccc = buff`

- The temporary memory modification problem, i.e., Example 16.13, can **not** be solved with this method.

A user defined DD instead of MPI_F_SYNC_REG. Instead of MPI_F_SYNC_REG, one can use also a user defined external subroutine, which is separately compiled:

```
subroutine DD(buf)
  integer buf
end
```

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)
```

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.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method has **not** any impact on problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication, i.e., problems through temporary memory modifications are not solved.

The VOLATILE attribute, gives the buffer or variable the properties needed, but it may 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.

16.2.3 Fortran Support Through the mpif.h Include File

The use of the mpif.h include file is strongly discouraged.

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 Fortran bindings are compatible with Fortran 77 implicit-style interfaces in most cases. The include file mpif.h must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as INTEGER. This is reflected in the first of the two Fortran interfaces in each MPI function definition.
- Be valid and equivalent for both fixed- and free- source form.

For each MPI routine, an implementation can choose to use an implicit or explicit interface.

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.*)

16.2.4 Fortran Support Through the `mpi` Module

An MPI implementation must provide a module named `mpi` that can be `used` in a Fortran program. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces for all MPI routines, i.e., this module guarantees compile-time argument checking, and allows positional and keyword-based argument lists.
- Define all handles as `INTEGER`. This is reflected in the first of the two Fortran interfaces in each MPI function definition.
- Define also all the named handle types and MPI_Status that are used in the `mpi_f08` module. They are needed only when the application needs to convert and old-style `INTEGER` handle into a new-style handle with a named type.

An MPI implementation may provide other features in the `mpi` module that enhance the usability of MPI while maintaining adherence to the standard. For example, it mayprovide `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.*)

Advice to implementors. In the `mpi` module with some compilers, a choice argument can be implemented with the following explicit interface:

```
!DEC$ ATTRIBUTES NO_ARG_CHECK :: BUF
!$PRAGMA IGNORE_TKR BUF
REAL, DIMENSION(*) :: BUF
```

In this case, the compile-time constant MPI_SUBARRAYS equals MPI_SUBARRAYS_UNSUPPORTED. It is explicitly allowed that the choice arguments are implemented in the same way as with the `mpi_f08` module. In the case where the compiler does not provide such functionality, a set of overloaded functions may be used. See the paper of M. Hennecke [26]. (*End of advice to implementors.*)

16.2.5 Fortran Support Through the `mpi_f08` Module

An MPI implementation must provide a module named `mpi_f08` that can be used in a Fortran program. With this module, new Fortran definitions are added for each MPI routine, except for routines that are deprecated. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces for all MPI routines, i.e., this module guarantees compile-time argument checking.
- Define all handles with uniquely named handle types (instead of INTEGER handles in the `mpi` module). This is reflected in the second of the two Fortran interfaces in each MPI function definition.
- Set the `INTEGER` compile-time constant MPI_SUBARRAYS to MPI_SUBARRAYS_SUPPORTED and declare choice buffers with the Fortran 2008 feature assumed-type and assumed-rank `TYPE(*), DIMENSION(..)` if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays are valid also in nonblocking routines.
- Set the MPI_SUBARRAYS compile-time constant to MPI_SUBARRAYS_UNSUPPORTED and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be restricted as with the `mpi` module.

Advice to implementors. In this case, the choice argument may be implemented with an explicit interface with compiler directives, for example:

```
!DEC$ ATTRIBUTES NO_ARG_CHECK :: BUF
!$PRAGMA IGNORE_TKR BUF
REAL, DIMENSION(*) :: BUF
```

(*End of advice to implementors.*)

- Declare each argument with an `INTENT=IN`, `OUT`, or `INOUT` as appropriate.
- Declare all status and array_of_statuses output arguments as optional through function overloading, instead of using MPI_STATUS_IGNORE.
- Declare all array_of_errcodes output arguments as optional through function overloading, instead of using MPI_ERRCODES_IGNORE.
- Declare all ierror output arguments as `OPTIONAL`, except for user-defined callback functions (e.g., ftypeCOMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_NULL_COPY_FN).

Rationale. For user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., MPI_NULL_COPY_FN), the ierror argument is not optional, i.e., these user-defined functions need not to check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the `mpi_f08` module are designed based on the Fortran 2008 standard [34] together with the Technical Report (TR) on Further Interoperability with C [35] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The TR on further interoperability with C was defined by WG5 to support the MPI-3.0 standardization. "It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations."[3]

This TR contains language features that are needed for the MPI bindings in the `mpi_f08` module: assumed type and assumed rank. Here, it is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., `REAL`, `REAL*4`, `CHARACTER*5`, `CHARACTERR*(*)`, derived types.

Furthermore, the implementors of the MPI Fortran bindings can freely choose whether all bindings are defined as `BIND(C)`. This is important to implement the Fortran `mpi_f08` interface with only **one** set of **portable** wrapper routines written in C. For this implementation goal, the following additional features are used: `BIND(C)` together

[3] [35] page iv, sentence 7.

with `OPTIONAL`, and with standard Fortran types like `INTEGER` and `CHARACTER` (i.e., not only with `INTEGER(C_INT)`).

The MPI Forum wants to acknoledge this important effort by the Fortran WG5 committee. (*End of rationale.*)

16.2.6 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 16.2.2 on page 549, a dummy call is needed to tell the compiler that registers are to be flushed for a given buffer. It is a generic Fortran routine and has only a Fortran binding.

MPI_F_SYNC_REG(buf)

INOUT	buf	initial address of buffer (choice)

```
MPI_F_SYNC_REG(buf)
    <type> buf(*)
```

```
MPI_F_sync_reg(buf)
    TYPE(*), DIMENSION(..)  ::  buf
```

This routine has no operation associated with. It must be compiled in the MPI library in the way that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to tell the compiler that a cached register value of a variable or buffer should be flushed, i.e., stored back to the memory (when necessary) or invalidated.

Rationale. This function is not available in other languages because it would not be useful. This routine has not an ierror return argument because there isn't any operation that can detect an error. (*End of rationale.*)

Advice to implementors. It is recommended to bind this routine to a C routine to minimize the risk that the fortran compiler can learn that this routine is empty, i.e., that the compiler can learn that a call to this routine can be removed as part of the automated optimization. (*End of advice to implementors.*)

16.2.7 Additional Support for Fortran Numeric Intrinsic Types

The routines in this section are part of Extended Fortran Support described in Section 16.2.4.

MPI 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

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

```
MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) :: typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

{`static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size)` *(binding deprecated, see Section 15.2)* }

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.*)

16.3.4 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. With the Fortran `mpi` module or the `mpif.h` include file, a Fortran handle is a Fortran `INTEGER` value that can be used in the following conversion functions. With the Fortran `mpi_f08` module, a Fortran handle is a derived tpye that contains the Fortran `INTEGER` field MPI_VAL, which contains the the INTEGER value that can be used in the following conversion functions.

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. See also Section 2.6.5 on page 22.

```
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)
```

```
   //   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 with 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`.

16.3.5 Status

The following two procedures are provided in C to convert from a Fortran (with the `mpi` module or `mpif.h`) 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.*)

Using the `mpi_f08` Fortran module, a status is declared as `TYPE(MPI_Status)`. The C datatype `MPI_F_status` can be used to hand over a Fortran `TYPE(MPI_Status)` argument into a C routine.

```
int MPI_Status_f082c(MPI_F_status *f08_status, MPI_Status *c_status)
```

This C routine converts a Fortran `mpi_f08` f08_status into a C c_status.

```
int MPI_Status_c2f08(MPI_Status *c_status, MPI_F_status *f08_status)
```

This C routine converts a C c_status into a Fortran `mpi_f08` f08_status.

Conversion between the two Fortran versions of a status can be done with:

MPI_STATUS_F2F08(f_status, f08_status)

IN	f_status	status object declared as array
OUT	f08_status	status object declared as named type

```
int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F_status *f08_status)
```

```
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
    INTEGER F_STATUS(MPI_STATUS_SIZE)
    TYPE(MPI_Status) ::  F08_STATUS
    INTEGER IERROR
```

```
MPI_Status_f2f08(f_status, f08_status, ierror)
    INTEGER f_status(MPI_STATUS_SIZE)
    TYPE(MPI_Status) ::  f08_status
    INTEGER, OPTIONAL ::  ierror
```

This routine converts a Fortran `mpi` module status into a Fortran `mpi_f08` f08_status.

MPI_STATUS_F082F(f08_status, f_status)

IN	f08_status	status object declared as named type
OUT	f_status	status object declared as array

```
int MPI_Status_f082f(MPI_F_status *f08_status, MPI_Fint *f_status)
```

```
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
```

```
    TYPE(MPI_Status) :: F08_STATUS
    INTEGER F_STATUS(MPI_STATUS_SIZE)
    INTEGER IERROR
```

```
MPI_Status_f082f(f08_status, f_status, ierror)
    TYPE(MPI_Status) :: f08_status
    INTEGER :: f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL :: ierror
```

This routine converts a Fortran `mpi_f08` module f08_status into a Fortran `mpi` status.

16.3.6 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 16.3.9, page 583).

Example 16.19

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) AODISP(1)

! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN,AODISP,AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
```

Assorted Constants

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_PROC_NULL	MPI::PROC_NULL
MPI_ANY_SOURCE	MPI::ANY_SOURCE
MPI_ANY_TAG	MPI::ANY_TAG
MPI_UNDEFINED	MPI::UNDEFINED
MPI_BSEND_OVERHEAD	MPI::BSEND_OVERHEAD
MPI_KEYVAL_INVALID	MPI::KEYVAL_INVALID
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

Status size and reserved index values (Fortran only)

Fortran type: `INTEGER`	
MPI_STATUS_SIZE	Not defined for C++
MPI_SOURCE	Not defined for C++
MPI_TAG	Not defined for C++
MPI_ERROR	Not defined for C++

Variable Address Size (Fortran only)

Fortran type: `INTEGER`	
MPI_ADDRESS_KIND	Not defined for C++
MPI_INTEGER_KIND	Not defined for C++
MPI_OFFSET_KIND	Not defined for C++

Error-handling specifiers

C type: `MPI_Errhandler` Fortran type: `INTEGER` or `TYPE(MPI_Errhandler)`	C++ type: `MPI::Errhandler`
MPI_ERRORS_ARE_FATAL	MPI::ERRORS_ARE_FATAL
MPI_ERRORS_RETURN	MPI::ERRORS_RETURN
	MPI::ERRORS_THROW_EXCEPTIONS

Maximum Sizes for Strings

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_MAX_PROCESSOR_NAME	MPI::MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING	MPI::MAX_ERROR_STRING
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		C/C++ types
C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype	
MPI_CHAR	MPI::CHAR	char (treated as printable character)
MPI_SHORT	MPI::SHORT	signed short int
MPI_INT	MPI::INT	signed int
MPI_LONG	MPI::LONG	signed long
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	signed long long
MPI_LONG_LONG	MPI::LONG_LONG	long long (synonym)
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	MPI::UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	MPI::UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	MPI::FLOAT	float
MPI_DOUBLE	MPI::DOUBLE	double
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	long double
MPI_WCHAR	MPI::WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	(use C datatype handle)	_Bool
MPI_INT8_T	(use C datatype handle)	int8_t
MPI_INT16_T	(use C datatype handle)	int16_t
MPI_INT32_T	(use C datatype handle)	int32_t
MPI_INT64_T	(use C datatype handle)	int64_t
MPI_UINT8_T	(use C datatype handle)	uint8_t
MPI_UINT16_T	(use C datatype handle)	uint16_t
MPI_UINT32_T	(use C datatype handle)	uint32_t
MPI_UINT64_T	(use C datatype handle)	uint64_t
MPI_AINT	(use C datatype handle)	MPI_Aint
MPI_OFFSET	(use C datatype handle)	MPI_Offset
MPI_C_COMPLEX	(use C datatype handle)	float _Complex
MPI_C_FLOAT_COMPLEX	(use C datatype handle)	float _Complex
MPI_C_DOUBLE_COMPLEX	(use C datatype handle)	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	(use C datatype handle)	long double _Complex
MPI_BYTE	MPI::BYTE	(any C/C++ type)
MPI_PACKED	MPI::PACKED	(any C/C++ type)

Named Predefined Datatypes		Fortran types
C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype	
MPI_INTEGER	MPI::INTEGER	INTEGER
MPI_REAL	MPI::REAL	REAL
MPI_DOUBLE_PRECISION	MPI::DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	MPI::F_COMPLEX	COMPLEX
MPI_LOGICAL	MPI::LOGICAL	LOGICAL
MPI_CHARACTER	MPI::CHARACTER	CHARACTER(1)
MPI_AINT	(use C datatype handle)	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	(use C datatype handle)	INTEGER (KIND=MPI_OFFSET_KIND)
MPI_BYTE	MPI::BYTE	(any Fortran type)
MPI_PACKED	MPI::PACKED	(any Fortran type)

C++-Only Named Predefined Datatypes	C++ types
C++ type: MPI::Datatype	
MPI::BOOL	bool
MPI::COMPLEX	Complex<float>
MPI::DOUBLE_COMPLEX	Complex<double>
MPI::LONG_DOUBLE_COMPLEX	Complex<long double>

Optional datatypes (Fortran)		Fortran types
C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype	
MPI_DOUBLE_COMPLEX	MPI::F_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_INTEGER1	MPI::INTEGER1	INTEGER*1
MPI_INTEGER2	MPI::INTEGER2	INTEGER*8
MPI_INTEGER4	MPI::INTEGER4	INTEGER*4
MPI_INTEGER8	MPI::INTEGER8	INTEGER*8
MPI_INTEGER16		INTEGER*16
MPI_REAL2	MPI::REAL2	REAL*2
MPI_REAL4	MPI::REAL4	REAL*4
MPI_REAL8	MPI::REAL8	REAL*8
MPI_REAL16		REAL*16
MPI_COMPLEX4		COMPLEX*4
MPI_COMPLEX8		COMPLEX*8
MPI_COMPLEX16		COMPLEX*16
MPI_COMPLEX32		COMPLEX*32

Datatypes for reduction functions (C and C++)

C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype
MPI_FLOAT_INT	MPI::FLOAT_INT
MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_LONG_INT	MPI::LONG_INT
MPI_2INT	MPI::TWOINT
MPI_SHORT_INT	MPI::SHORT_INT
MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT

Datatypes for reduction functions (Fortran)

C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype
MPI_2REAL	MPI::TWOREAL
MPI_2DOUBLE_PRECISION	MPI::TWODOUBLE_PRECISION
MPI_2INTEGER	MPI::TWOINTEGER

Special datatypes for constructing derived datatypes

C type: MPI_Datatype Fortran type: INTEGER or TYPE(MPI_Datatype)	C++ type: MPI::Datatype
MPI_UB	MPI::UB
MPI_LB	MPI::LB

Reserved communicators

C type: MPI_Comm Fortran type: INTEGER or TYPE(MPI_Comm)	C++ type: MPI::Intracomm
MPI_COMM_WORLD	MPI::COMM_WORLD
MPI_COMM_SELF	MPI::COMM_SELF

Results of communicator and group comparisons

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_IDENT	MPI::IDENT
MPI_CONGRUENT	MPI::CONGRUENT
MPI_SIMILAR	MPI::SIMILAR
MPI_UNEQUAL	MPI::UNEQUAL

Environmental inquiry keys

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_TAG_UB	MPI::TAG_UB
MPI_IO	MPI::IO
MPI_HOST	MPI::HOST
MPI_WTIME_IS_GLOBAL	MPI::WTIME_IS_GLOBAL

Collective Operations

C type: `MPI_Op` Fortran type: `INTEGER` or `TYPE(MPI_Op)`	C++ type: `const MPI::Op`
MPI_MAX	MPI::MAX
MPI_MIN	MPI::MIN
MPI_SUM	MPI::SUM
MPI_PROD	MPI::PROD
MPI_MAXLOC	MPI::MAXLOC
MPI_MINLOC	MPI::MINLOC
MPI_BAND	MPI::BAND
MPI_BOR	MPI::BOR
MPI_BXOR	MPI::BXOR
MPI_LAND	MPI::LAND
MPI_LOR	MPI::LOR
MPI_LXOR	MPI::LXOR
MPI_REPLACE	MPI::REPLACE

Null Handles

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_GROUP_NULL MPI_Group / INTEGER or TYPE(MPI_Group)	MPI::GROUP_NULL const MPI::Group
MPI_COMM_NULL MPI_Comm / INTEGER or TYPE(MPI_Comm)	MPI::COMM_NULL [1])
MPI_DATATYPE_NULL MPI_Datatype / INTEGER or TYPE(MPI_Datatype)	MPI::DATATYPE_NULL const MPI::Datatype
MPI_REQUEST_NULL MPI_Request / INTEGER or TYPE(MPI_Request)	MPI::REQUEST_NULL const MPI::Request
MPI_OP_NULL MPI_Op / INTEGER or TYPE(MPI_Op)	MPI::OP_NULL const MPI::Op
MPI_ERRHANDLER_NULL MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)	MPI::ERRHANDLER_NULL const MPI::Errhandler
MPI_FILE_NULL MPI_File / INTEGER or TYPE(MPI_File)	MPI::FILE_NULL
MPI_INFO_NULL MPI_Info / INTEGER or TYPE(MPI_Info)	MPI::INFO_NULL const MPI::Info
MPI_WIN_NULL MPI_Win / INTEGER or TYPE(MPI_Win)	MPI::WIN_NULL

[1]) C++ type: See Section 16.1.7 on page 536 regarding class hierarchy and the specific type of MPI::COMM_NULL

Empty group

C type: MPI_Group Fortran type: INTEGER or TYPE(MPI_Group)	C++ type: const MPI::Group
MPI_GROUP_EMPTY	MPI::GROUP_EMPTY

Topologies

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_GRAPH	MPI::GRAPH
MPI_CART	MPI::CART
MPI_DIST_GRAPH	MPI::DIST_GRAPH

Predefined functions

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_COMM_NULL_COPY_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FUNCTION	MPI_COMM_NULL_COPY_FN same as in C [1])
MPI_COMM_DUP_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FUNCTION	MPI_COMM_DUP_FN same as in C [1])
MPI_COMM_NULL_DELETE_FN MPI_Comm_delete_attr_function / COMM_DELETE_ATTR_FUNCTION	MPI_COMM_NULL_DELETE_FN same as in C [1])
MPI_WIN_NULL_COPY_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FUNCTION	MPI_WIN_NULL_COPY_FN same as in C [1])
MPI_WIN_DUP_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FUNCTION	MPI_WIN_DUP_FN same as in C [1])
MPI_WIN_NULL_DELETE_FN MPI_Win_delete_attr_function / WIN_DELETE_ATTR_FUNCTION	MPI_WIN_NULL_DELETE_FN same as in C [1])
MPI_TYPE_NULL_COPY_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FUNCTION	MPI_TYPE_NULL_COPY_FN same as in C [1])
MPI_TYPE_DUP_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FUNCTION	MPI_TYPE_DUP_FN same as in C [1])
MPI_TYPE_NULL_DELETE_FN MPI_Type_delete_attr_function / TYPE_DELETE_ATTR_FUNCTION	MPI_TYPE_NULL_DELETE_FN same as in C [1])

[1] See the advice to implementors on MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2 on page 266

Deprecated predefined functions

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_NULL_COPY_FN MPI_Copy_function / COPY_FUNCTION	MPI::NULL_COPY_FN MPI::Copy_function
MPI_DUP_FN MPI_Copy_function / COPY_FUNCTION	MPI::DUP_FN MPI::Copy_function
MPI_NULL_DELETE_FN MPI_Delete_function / DELETE_FUNCTION	MPI::NULL_DELETE_FN MPI::Delete_function

File Operation Constants, Part 2

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
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

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_TYPECLASS_COMPLEX	MPI::TYPECLASS_COMPLEX
MPI_TYPECLASS_INTEGER	MPI::TYPECLASS_INTEGER
MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL

Constants Specifying Empty or Ignored Input

C/Fortran name C type / Fortran type with `mpi` module / Fortran type with `mpi_f08` module	C++ name C++ type
MPI_ARGVS_NULL `char***` / 2-dim. array of `CHARACTER*(*)` / 2-dim. array of `CHARACTER*(*)`	MPI::ARGVS_NULL `const char ***`
MPI_ARGV_NULL `char**` / array of `CHARACTER*(*)` / array of `CHARACTER*(*)`	MPI::ARGV_NULL `const char **`
MPI_ERRCODES_IGNORE `int*` / `INTEGER` array / not defined	Not defined for C++
MPI_STATUSES_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE,*)` / not defined	Not defined for C++
MPI_STATUS_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE)` / not defined	Not defined for C++
MPI_UNWEIGHTED `int*` / `INTEGER` array / not defined	Not defined for C++

MPI::File
MPI::Group
MPI::Info
MPI::Op
MPI::Request
MPI::Prequest
MPI::Grequest
MPI::Win

The following are defined Fortran type definitions, included in the `mpi_f08` and `mpi` module.

```
!  Fortran opaque types in the mpi_f08 and mpi module
TYPE(MPI_Status)

!  Fortran handles in the mpi_f08 and mpi module
TYPE(MPI_Comm)
TYPE(MPI_Datatype)
TYPE(MPI_Errhandler)
TYPE(MPI_File)
TYPE(MPI_Group)
TYPE(MPI_Info)
TYPE(MPI_Op)
TYPE(MPI_Request)
TYPE(MPI_Win)
```

A.1.3 Prototype Definitions

The following are defined C typedefs for user-defined functions, also included in the file `mpi.h`.

```
/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
              MPI_Datatype *datatype);

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 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 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 void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);

typedef int MPI_Grequest_query_function(void *extra_state,
            MPI_Status *status);
typedef int MPI_Grequest_free_function(void *extra_state);
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
            MPI_Aint *file_extent, void *extra_state);
typedef int MPI_Datarep_conversion_function(void *userbuf,
            MPI_Datatype datatype, int count, void *filebuf,
            MPI_Offset position, void *extra_state);
```

For Fortran, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
   <type> INVEC(LEN), INOUTVEC(LEN)
   INTEGER LEN, TYPE
```

The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(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_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
   INTEGER COMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE WIN_COPY_ATTR_FUNCTION(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_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
                EXTRA_STATE, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(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_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                 EXTRA_STATE, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
    INTEGER COMM, ERROR_CODE
```

The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
```

```
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
```

The extend and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:

```
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, 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
```

The following are defined C++ typedefs, also included in the file `mpi.h`.

```
namespace MPI {
  typedef void User_function(const void* invec, void *inoutvec,
              int len, const Datatype& datatype);

  typedef int Comm::Copy_attr_function(const Comm& oldcomm,
              int comm_keyval, void* extra_state, void* attribute_val_in,
              void* attribute_val_out, bool& flag);
  typedef int Comm::Delete_attr_function(Comm& comm, int
              comm_keyval, void* attribute_val, void* extra_state);

  typedef int Win::Copy_attr_function(const Win& oldwin,
              int win_keyval, void* extra_state, void* attribute_val_in,
              void* attribute_val_out, bool& flag);
  typedef int Win::Delete_attr_function(Win& win, int
              win_keyval, void* attribute_val, void* extra_state);

  typedef int Datatype::Copy_attr_function(const Datatype& oldtype,
              int type_keyval, void* extra_state,
              const void* attribute_val_in, void* attribute_val_out,
              bool& flag);
  typedef int Datatype::Delete_attr_function(Datatype& type,
              int type_keyval, void* attribute_val, void* extra_state);

  typedef void Comm::Errhandler_function(Comm &, int *, ...);
  typedef void Win::Errhandler_function(Win &, int *, ...);
```

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)

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)

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)

int MPI_Type_commit(MPI_Datatype *datatype)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

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)

int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_free(MPI_Datatype *datatype)

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 MPI_Ireduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
             MPI_Request* request)

int MPI_Ireduce_scatter(void* sendbuf, void* recvbuf, int recvcounts[],
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request* request)

int MPI_Ireduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request* request)

int MPI_Iscan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request* request)

int MPI_Iscatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm, MPI_Request* request)

int MPI_Iscatterv(void* sendbuf, int sendcounts[], int displs[],
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm,
             MPI_Request* request)

int MPI_Op_commutative(MPI_Op op, int *commute)

int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
             MPI_Datatype datatype, MPI_Op op)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int recvcounts[],
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm)

int MPI_Scatterv(void* sendbuf, int sendcounts[], int displs[],
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_op_free(MPI_Op *op)
```

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)

int MPI_Group_rank(MPI_Group group, int *rank)

int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
             MPI_Group group2, int *ranks2)

int MPI_Group_union(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
             MPI_Comm peer_comm, int remote_leader, int tag,
             MPI_Comm *newintercomm)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
             MPI_Comm *newintracomm)

int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag)

int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag)

int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void
             *attribute_val, void *extra_state)

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 datatype, int type_keyval)

int MPI_Type_free_keyval(int *type_keyval)

int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void
             *attribute_val, int *flag)

int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int
             *resultlen)

int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
             void *attribute_val)
```

```
int MPI_Type_set_name(MPI_Datatype datatype, char *type_name)

int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
              void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
              void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void
              *attribute_val, void *extra_state)

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.2.5 Process Topologies C Bindings

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
              int reorder, MPI_Comm *comm_cart)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
              int *coords)

int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
              int *newrank)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
              int *rank_source, int *rank_dest)

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

int MPI_Dims_create(int nnodes, int ndims, int *dims)

int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
              int degrees[], int destinations[], int weights[],
              MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
             int sources[], int sourceweights[], int outdegree,
             int destinations[], int destweights[], MPI_Info info,
             int reorder, MPI_Comm *comm_dist_graph)

int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
             int sourceweights[], int maxoutdegree, int destinations[],
             int destweights[])

int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
             int *outdegree, int *weighted)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
             int reorder, MPI_Comm *comm_graph)

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
             int *edges)

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
             int *newrank)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
             int *neighbors)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

int MPI_Topo_test(MPI_Comm comm, int *status)
```

A.2.6 MPI Environmental Management C Bindings

```
double MPI_Wtick(void)

double MPI_Wtime(void)

int MPI_Abort(MPI_Comm comm, int errorcode)

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_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
             *comm_errhandler_fn, MPI_Errhandler *errhandler)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

```
int MPI_Error_class(int errorcode, int *errorclass)

int MPI_Error_string(int errorcode, char *string, int *resultlen)

int MPI_File_call_errhandler(MPI_File fh, int errorcode)

int MPI_File_create_errhandler(MPI_File_errhandler_function
             *file_errhandler_fn, MPI_Errhandler *errhandler)

int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

int MPI_Finalize(void)

int MPI_Finalized(int *flag)

int MPI_Free_mem(void *base)

int MPI_Get_processor_name(char *name, int *resultlen)

int MPI_Get_version(int *version, int *subversion)

int MPI_Init(int *argc, char ***argv)

int MPI_Initialized(int *flag)

int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

int MPI_Win_create_errhandler(MPI_Win_errhandler_function
             *win_errhandler_fn, MPI_Errhandler *errhandler)

int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

A.2.7 The Info Object C Bindings

```
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)

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)
```

```
              void *extra_state)
```

A.2.12 Language Bindings C Bindings

```
int MPI_Status_f082f(MPI_F_status *f08_status, MPI_Fint *f_status)

int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F_status *f08_status)

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 *datatype)

MPI_Fint MPI_Comm_c2f(MPI_Comm comm)

MPI_Comm MPI_Comm_f2c(MPI_Fint comm)

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)

MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)

MPI_Fint MPI_File_c2f(MPI_File file)

MPI_File MPI_File_f2c(MPI_Fint file)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Info_c2f(MPI_Info info)

MPI_Info MPI_Info_f2c(MPI_Fint info)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_Request MPI_Request_f2c(MPI_Fint request)

int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)

int MPI_Status_c2f08(MPI_Status *c_status, MPI_F_status *f08_status)

int MPI_Status_f082c(MPI_F_status *f08_status, MPI_Status *c_status)

int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Win MPI_Win_f2c(MPI_Fint win)
```

A.2.13 Profiling Interface C Bindings

```
int MPI_Pcontrol(const int level, ...)
```

A.2.14 Deprecated C Bindings

```
int MPI_Address(void* location, MPI_Aint *address)

int MPI_Attr_delete(MPI_Comm comm, int keyval)

int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)

int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
             void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_Errhandler_create(MPI_Handler_function *handler_fn,
             MPI_Errhandler *errhandler)

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
             *delete_fn, int *keyval, void* extra_state)

int MPI_Keyval_free(int *keyval)

int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
             void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,
             void *extra_state)

int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)

int MPI_Type_struct(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements,
             MPI_Datatype *array_of_types, MPI_Datatype *newtype)

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

A.3 Fortran Bindings with mpif.h or the mpi Module

A.3.1 Point-to-Point Communication Fortran Bindings

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

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

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

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

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

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

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

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

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

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

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

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

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

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

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

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR

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

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

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

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, 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_DUP(OLDTYPE, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

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_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_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
             DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
```

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR

MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, REQUEST, IERROR

MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR

MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR

MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR

MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, REQUEST, IERROR

MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
    EXTERNAL USER_FN
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

A.3.4 Groups, Contexts, Communicators, and Caching Fortran Bindings

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, 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_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
             TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
    INTEGER NEWINTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
```

```
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(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR

MPI_TYPE_DUP_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

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR

MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER DATATYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME

MPI_TYPE_NULL_COPY_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

MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
             IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME

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 COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

A.3.6 MPI Environmental Management Fortran Bindings

```
DOUBLE PRECISION MPI_WTICK()

DOUBLE PRECISION MPI_WTIME()

MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

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_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    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_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR

MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

```
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING

MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR

MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL FILE_ERRHANDLER_FN
    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_FINALIZE(IERROR)
    INTEGER IERROR

MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR

MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR

MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR

MPI_INIT(IERROR)
    INTEGER IERROR

MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL WIN_ERRHANDLER_FN
    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
```

```
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.3.12 Language Bindings Fortran Bindings

```
MPI_F_SYNC_REG(buf)
    <type> buf(*)

MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR

MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
    TYPE(MPI_Status) ::  F08_STATUS
    INTEGER F_STATUS(MPI_STATUS_SIZE)
    INTEGER IERROR

MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
    INTEGER F_STATUS(MPI_STATUS_SIZE)
    TYPE(MPI_Status) ::  F08_STATUS
    INTEGER 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, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

A.3.13 Profiling Interface Fortran Bindings

```
MPI_PCONTROL(LEVEL)
    INTEGER LEVEL
```

A.3.14 Deprecated Fortran Bindings

```
MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG

MPI_ERRHANDLER_CREATE(HANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL HANDLER_FN
    INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR

MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG

MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
```

A.4 Fortran 2008 Bindings with the mpi_f08 Module

A.4.1 Point-to-Point Communication Fortran 2008 Bindings

```
MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer
    INTEGER, INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Buffer_detach(buffer_addr, size, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer_addr
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Cancel(request, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Get_count(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Iprobe(source, tag, comm, flag, status, ierror)
    INTEGER, INTENT(IN) ::  source, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    LOGICAL, INTENT(OUT) ::  flag
```

```
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Probe(source, tag, comm, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
```

```
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Request_free(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Request_get_status( request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  sendbuf, recvbuf
    INTEGER, INTENT(IN) ::  sendcount, dest, sendtag, recvcount, source,
    recvtag
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
             comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, sendtag, source, recvtag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Start(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Startall(count, array_of_requests, ierror)
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Test(request, flag, status, ierror)
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Test_cancelled(status, flag, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
    INTEGER, INTENT(IN) ::  count
```

```
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(*)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status), INTENT(OUT) :: array_of_statuses(*) ! optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testany(count, array_of_requests, index, flag, status, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(*)
    INTEGER, INTENT(OUT) :: index
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(*)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status), INTENT(OUT) :: array_of_statuses(*) ! optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Wait(request, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
    INTEGER, INTENT(INOUT) :: array_of_requests(*)
    TYPE(MPI_Status), INTENT(OUT) :: array_of_statuses(*) ! optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitany(count, array_of_requests, index, status, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(*)
    INTEGER, INTENT(OUT) :: index
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(*)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status), INTENT(OUT) :: array_of_statuses(*) ! optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.2 Datatypes Fortran 2008 Bindings

```
MPI_Get_address(location, address, ierror)
    TYPE(*), DIMENSION(..) :: location
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: count
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
    TYPE(*), DIMENSION(..) :: inbuf, outbuf
    INTEGER, INTENT(IN) :: incount, outsize
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(INOUT) :: position
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
             position, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(..) :: inbuf, outbuf
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: incount
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_size(incount, datatype, comm, size, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_commit(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_contiguous(count, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
             array_of_distribs, array_of_dargs, array_of_psizes, order,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  size, rank, ndims, array_of_gsizes(*),
    array_of_distribs(*), array_of_dargs(*), array_of_psizes(*), order
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_hindexed(count, array_of_blocklengths,
             array_of_displacements, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
             ierror)
    INTEGER, INTENT(IN) ::  count, blocklength
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  stride
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, blocklength, array_of_displacements(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  lb, extent
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_struct(count, array_of_blocklengths,
             array_of_displacements, array_of_types, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  array_of_types(*)
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
             array_of_starts, order, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  ndims, array_of_sizes(*), array_of_subsizes(*),
    array_of_starts(*), order
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_free(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
             array_of_integers, array_of_addresses, array_of_datatypes,
             ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  max_integers, max_addresses, max_datatypes
    INTEGER, INTENT(OUT) ::  array_of_integers(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  array_of_addresses(*)
    TYPE(MPI_Datatype), INTENT(OUT) ::  array_of_datatypes(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
             combiner, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  num_integers, num_addresses, num_datatypes,
    combiner
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_extent(datatype, lb, extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  lb, extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  true_lb, true_extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(*),
    array_of_displacements(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
```

```
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_size(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, blocklength, stride
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
             ierror)
    TYPE(*), DIMENSION(..) :: inbuf, outbuf
    INTEGER, INTENT(IN) :: insize, outcount
    INTEGER, INTENT(INOUT) :: position
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
             datatype, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(..) :: inbuf, outbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
    INTEGER, INTENT(IN) :: outcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.3 Collective Communication Fortran 2008 Bindings

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
             rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
             rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
    TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Barrier(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Bcast(buffer, count, datatype, root, comm, ierror)
    TYPE(*), DIMENSION(..) :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  sendbuf, recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  sendbuf, recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype
    TYPE(MPI_Datatype), INTENT(IN) ::  recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  sendbuf, recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*), root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype
    TYPE(MPI_Datatype), INTENT(IN) ::  recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Iallgather( -- Fortran 2008 -- to be done -- )

MPI_Iallgatherv( -- Fortran 2008 -- to be done -- )

MPI_Iallreduce( -- Fortran 2008 -- to be done -- )

MPI_Ialltoall( -- Fortran 2008 -- to be done -- )

MPI_Ialltoallv( -- Fortran 2008 -- to be done -- )

MPI_Ialltoallw( -- Fortran 2008 -- to be done -- )

MPI_Ibarrier( -- Fortran 2008 -- to be done -- )

MPI_Ibcast( -- Fortran 2008 -- to be done -- )

MPI_Iexscan( -- Fortran 2008 -- to be done -- )

MPI_Igather( -- Fortran 2008 -- to be done -- )

MPI_Igatherv( -- Fortran 2008 -- to be done -- )

MPI_Ireduce( -- Fortran 2008 -- to be done -- )

MPI_Ireduce_scatter( -- Fortran 2008 -- to be done -- )

MPI_Ireduce_scatter_block( -- Fortran 2008 -- to be done -- )

MPI_Iscan( -- Fortran 2008 -- to be done -- )
```

```
MPI_Iscatter( -- Fortran 2008 -- to be done -- )

MPI_Iscatterv( -- Fortran 2008 -- to be done -- )

MPI_Op_commutative(op, commute, ierror)
    TYPE(MPI_Op), INTENT(IN) :: op
    LOGICAL, INTENT(OUT) :: commute
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Op_create(user_fn, commute, op, ierror)
    EXTERNAL :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Op_free(op, ierror)
    TYPE(MPI_Op), INTENT(INOUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(..) :: inbuf, inoutbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..) :: sendbuf, recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype
    TYPE(MPI_Datatype), INTENT(IN) :: recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings

```
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: oldcomm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: oldcomm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
```

```
            ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val,
    extra_state
    INTEGER, INTENT(OUT) ::  ierror

MPI_Comm_compare(comm1, comm2, result, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm1
    TYPE(MPI_Comm), INTENT(IN) ::  comm2
    INTEGER, INTENT(OUT) ::  result
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_create(comm, group, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  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, INTENT(OUT) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_delete_attr(comm, comm_keyval, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_dup(comm, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_free(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_free_keyval(comm_keyval, ierror)
    INTEGER, INTENT(INOUT) ::  comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=*), INTENT(OUT) :: comm_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_rank(comm, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_remote_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_remote_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_set_name(comm, comm_name, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=*), INTENT(IN) :: comm_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_split(comm, color, key, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: color, key
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_test_inter(comm, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Group_compare(group1, group2, result, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    INTEGER, INTENT(OUT) :: result
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_difference(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_excl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: n, ranks(*)
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_free(group, ierror)
    TYPE(MPI_Group), INTENT(INOUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_incl(group, n, ranks, newgroup, ierror)
    INTEGER, INTENT(IN) :: n, ranks(*)
    TYPE(MPI_Group), INTENT(IN) :: group
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_intersection(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: n, ranges(3,*)
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: n, ranges(3,*)
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_rank(group, rank, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_size(group, size, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    INTEGER, INTENT(IN) :: n, ranks1(*)
    INTEGER, INTENT(OUT) :: ranks2(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_union(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
             tag, newintercomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
    LOGICAL, INTENT(IN) :: high
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
             ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
    extra_state
```

```
    INTEGER, INTENT(OUT) ::  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, INTENT(OUT) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_delete_attr(datatype, type_keyval, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) ::  type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_name(datatype, type_name, resultlen, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=*), INTENT(OUT) ::  type_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_set_name(datatype, type_name, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=*), INTENT(IN) ::  type_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    INTEGER, INTENT(IN) ::  oldwin, win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val_out
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, INTENT(OUT) ::  ierror

MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
```

```
          attribute_val_out, flag, ierror)
    INTEGER, INTENT(IN) :: oldwin, win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
    extra_state
    INTEGER, INTENT(OUT) :: 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, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_free_keyval(win_keyval, ierror)
    INTEGER, INTENT(INOUT) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_name(win, win_name, resultlen, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    CHARACTER(LEN=*), INTENT(OUT) :: win_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_name(win, win_name, ierror)
```

```
    TYPE(MPI_Win), INTENT(IN) :: win
    CHARACTER(LEN=*), INTENT(IN) :: win_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.5 Process Topologies Fortran 2008 Bindings

```
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank, maxdims
    INTEGER, INTENT(OUT) :: coords(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: ndims, dims(*)
    LOGICAL, INTENT(IN) :: periods(*), reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: maxdims
    INTEGER, INTENT(OUT) :: dims(*), coords(*)
    LOGICAL, INTENT(OUT) :: periods(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: ndims, dims(*)
    LOGICAL, INTENT(IN) :: periods(*)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_rank(comm, coords, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: coords(*)
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: direction, disp
    INTEGER, INTENT(OUT) :: rank_source, rank_dest
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(IN) :: remain_dims(*)
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  ndims
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dims_create(nnodes, ndims, dims, ierror)
    INTEGER, INTENT(IN) ::  nnodes, ndims
    INTEGER, INTENT(INOUT) ::  dims(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
            info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  n, sources(*), degrees(*), destinations(*)
    INTEGER, INTENT(IN) ::  weights(*) !  optional by overloading
    TYPE(MPI_Info), INTENT(IN) ::  info
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
            outdegree, destinations, destweights, info, reorder,
            comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  indegree, sources(*), outdegree,
    destinations(*)
    INTEGER, INTENT(IN) ::  sourceweights(*), destweights(*) !  optional by
    overloading
    TYPE(MPI_Info), INTENT(IN) ::  info
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
            maxoutdegree, destinations, destweights, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  maxindegree, maxoutdegree
    INTEGER, INTENT(OUT) ::  sources(*), destinations(*)
    INTEGER ::  sourceweights(*), destweights(*) !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  indegree, outdegree
    LOGICAL, INTENT(OUT) ::  weighted
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
            ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
```

```
    INTEGER, INTENT(IN) :: nnodes, index(*), edges(*)
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: maxindex, maxedges
    INTEGER, INTENT(OUT) :: index(*), edges(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: nnodes, index(*), edges(*)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank, maxneighbors
    INTEGER, INTENT(OUT) :: neighbors(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank
    INTEGER, INTENT(OUT) :: nneighbors
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Graphdims_get(comm, nnodes, nedges, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: nnodes, nedges
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Topo_test(comm, status, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.6 MPI Environmental Management Fortran 2008 Bindings

```
DOUBLE PRECISION MPI_Wtick()

DOUBLE PRECISION MPI_Wtime()

MPI_Abort(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_class(errorclass, ierror)
    INTEGER, INTENT(OUT) :: errorclass
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_code(errorclass, errorcode, ierror)
    INTEGER, INTENT(IN) :: errorclass
    INTEGER, INTENT(OUT) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_string(errorcode, string, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=*), INTENT(IN) :: string
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_call_errhandler(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
    EXTERNAL :: comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_set_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Errhandler_free(errhandler, ierror)
    TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Error_class(errorcode, errorclass, ierror)
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Error_string(errorcode, string, resultlen, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=*), INTENT(OUT) :: string
    INTEGER, INTENT(OUT) :: resultlen
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_call_errhandler(fh, errorcode, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
    EXTERNAL :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_set_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Finalize(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Finalized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Free_mem(base, ierror)
    USE, INTRINSIC :: ISO_C_BINDING
    TYPE(C_PTR), INTENT(IN) :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_processor_name( name, resultlen, ierror)
    CHARACTER(LEN=*), INTENT(OUT) :: name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_version(version, subversion, ierror)
    INTEGER, INTENT(OUT) :: version, subversion
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Init(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Initialized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_call_errhandler(win, errorcode, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: errorcode
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
    EXTERNAL :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.7 The Info Object Fortran 2008 Bindings

```
MPI_Info_create(info, ierror)
    TYPE(MPI_Info), INTENT(OUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_delete(info, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_dup(info, newinfo, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Info), INTENT(OUT) :: newinfo
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_free(info, ierror)
    TYPE(MPI_Info), INTENT(INOUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get(info, key, valuelen, value, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(IN) :: valuelen
    CHARACTER(LEN=*), INTENT(OUT) :: value
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nkeys(info, nkeys, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(OUT) :: nkeys
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nthkey(info, n, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
```

```
    INTEGER, INTENT(IN) :: n
    CHARACTER(LEN=*), INTENT(OUT) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(OUT) :: valuelen
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_set(info, key, value, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key, value
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.8 Process Creation and Management Fortran 2008 Bindings

```
MPI_Close_port(port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_disconnect(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm), INTENT(OUT) :: parent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_join(fd, intercomm, ierror)
    INTEGER, INTENT(IN) :: fd
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
             array_of_errcodes, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  command, argv(*)
    INTEGER, INTENT(IN) ::  maxprocs, root
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  intercomm
    INTEGER, INTENT(OUT) ::  array_of_errcodes(*) !  optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  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, INTENT(IN) ::  count, array_of_maxprocs(*), root
    CHARACTER(LEN=*), INTENT(IN) ::  array_of_commands(*),
    array_of_argv(count, *)
    TYPE(MPI_Info), INTENT(IN) ::  array_of_info(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  intercomm
    INTEGER, INTENT(OUT) ::  array_of_errcodes(*) !  optional by
    overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Lookup_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  service_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    CHARACTER(LEN=*), INTENT(OUT) ::  port_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Open_port(info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) ::  info
    CHARACTER(LEN=*), INTENT(OUT) ::  port_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Publish_name(service_name, info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) ::  info
    CHARACTER(LEN=*), INTENT(IN) ::  service_name, port_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Unpublish_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  service_name, port_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

A.4.9 One-Sided Communications Fortran 2008 Bindings

```
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..)  ::  origin_addr
```

```
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Datatype), INTENT(IN) :: target_datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..) :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Datatype), INTENT(IN) :: target_datatype
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..) :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Datatype), INTENT(IN) :: target_datatype
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_complete(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
    TYPE(*), DIMENSION(..) :: base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_free(win, ierror)
    TYPE(MPI_Win), INTENT(INOUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_group(win, group, ierror)
```

```
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_lock(lock_type, rank, assert, win, ierror)
    INTEGER, INTENT(IN) :: lock_type, rank, assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_post(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_start(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_test(win, flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_unlock(rank, win, ierror)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_wait(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.10 External Interfaces Fortran 2008 Bindings

```
MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
             ierror)
    EXTERNAL :: query_fn, free_fn, cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Init_thread(required, provided, ierror)
    INTEGER, INTENT(IN) :: required
    INTEGER, INTENT(OUT) :: provided
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Is_thread_main(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Query_thread(provided, ierror)
    INTEGER, INTENT(OUT) :: provided
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_set_cancelled(status, flag, ierror)
    TYPE(MPI_Status), INTENT(INOUT) :: status
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_set_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: count
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.11 I/O Fortran 2008 Bindings

```
MPI_File_close(fh, ierror)
    TYPE(MPI_File), INTENT(INOUT) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_delete(filename, info, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: filename
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_amode(fh, amode, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(OUT) :: amode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_byte_offset(fh, offset, disp, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_group(fh, group, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_File_get_info(fh, info_used, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_position(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_position_shared(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_type_extent(fh, datatype, extent, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
    TYPE(MPI_Datatype), INTENT(OUT) :: etype
    TYPE(MPI_Datatype), INTENT(OUT) :: filetype
    CHARACTER(LEN=*), INTENT(OUT) :: datarep
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
    TYPE(*), DIMENSION(..) :: buf
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_open(comm, filename, amode, info, fh, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=*), INTENT(IN) :: filename
    INTEGER, INTENT(IN) :: amode
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_File), INTENT(OUT) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_preallocate(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status), INTENT(OUT) ::  status !  optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_File_read_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_seek(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER, INTENT(IN) :: whence
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_seek_shared(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER, INTENT(IN) :: whence
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_set_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    LOGICAL, INTENT(IN) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_File_set_info(fh, info, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_set_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
    TYPE(MPI_Datatype), INTENT(IN) :: etype
    TYPE(MPI_Datatype), INTENT(IN) :: filetype
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_sync(fh, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
```

```
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status), INTENT(OUT) :: status ! optional by overloading
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
             dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    EXTERNAL :: read_conversion_fn, write_conversion_fn,
    dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.12 Language Bindings Fortran 2008 Bindings

```
MPI_F_sync_reg(buf)
    TYPE(*), DIMENSION(..) :: buf

MPI_Sizeof(x, size, ierror)
    TYPE(*) :: x
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_f082f(f08_status, f_status, ierror)
    TYPE(MPI_Status) :: f08_status
    INTEGER :: f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL :: ierror

MPI_Status_f2f08(f_status, f08_status, ierror)
    INTEGER f_status(MPI_STATUS_SIZE)
    TYPE(MPI_Status) :: f08_status
    INTEGER, OPTIONAL :: ierror

MPI_Type_create_f90_complex(p, r, newtype, ierror)
    INTEGER, INTENT(IN) :: p, r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_f90_integer(r, newtype, ierror)
    INTEGER, INTENT(IN) :: r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_f90_real(p, r, newtype, ierror)
    INTEGER, INTENT(IN) :: p, r
```

```
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) :: typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.4.13 Profiling Interface Fortran 2008 Bindings

```
MPI_Pcontrol(level)
    INTEGER, INTENT(IN) :: level
```

A.4.14 Deprecated Fortran 2008 Bindings

{void Op::Init(User_function* user_fn, bool commute) *(binding deprecated, see Section 15.2)* }

{Request Comm::Ireduce(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op, int root) const = 0 *(binding deprecated, see Section 15.2)* }

{Request Comm::Ireduce_scatter(const void* sendbuf, void* recvbuf, int recvcounts[], const Datatype& datatype, const Op& op) const = 0 *(binding deprecated, see Section 15.2)* }

{Request Comm::Ireduce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount, const Datatype& datatype, const Op& op) const = 0 *(binding deprecated, see Section 15.2)* }

{bool Op::Is_commutative() const *(binding deprecated, see Section 15.2)* }

{Request Intracomm::Iscan(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op) const *(binding deprecated, see Section 15.2)* }

{Request Comm::Iscatter(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 *(binding deprecated, see Section 15.2)* }

{Request Comm::Iscatterv(const void* sendbuf, const int sendcounts[], const int displs[], const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 *(binding deprecated, see Section 15.2)* }

{void Comm::Reduce(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op, int root) const = 0 *(binding deprecated, see Section 15.2)* }

{void Op::Reduce_local(const void* inbuf, void* inoutbuf, int count, const Datatype& datatype) const *(binding deprecated, see Section 15.2)* }

{void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf, int recvcounts[], const Datatype& datatype, const Op& op) const = 0 *(binding deprecated, see Section 15.2)* }

{void Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount, const Datatype& datatype, const Op& op) const = 0 *(binding deprecated, see Section 15.2)* }

{void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op) const *(binding deprecated, see Section 15.2)* }

{void Comm::Scatter(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 *(binding deprecated,* {static Errhandler Comm::Create_errhandler(Comm::Errhandler_function* comm_errhandler_fn) *(binding deprecated, see Section 15.2)* }

{static Errhandler File::Create_errhandler(File::Errhandler_function* file_errhandler_fn) *(binding deprecated, see Section 15.2)* }

{static Errhandler Win::Create_errhandler(Win::Errhandler_function* win_errhandler_fn) *(binding deprecated, see Section 15.2)* }

{void Finalize() *(binding deprecated, see Section 15.2)* }

{void Errhandler::Free() *(binding deprecated, see Section 15.2)* }

{void Free_mem(void *base) *(binding deprecated, see Section 15.2)* }

{Errhandler Comm::Get_errhandler() const *(binding deprecated, see Section 15.2)* }

{Errhandler File::Get_errhandler() const *(binding deprecated, see Section 15.2)* }

{Errhandler Win::Get_errhandler() const *(binding deprecated, see Section 15.2)* }

{int Get_error_class(int errorcode) *(binding deprecated, see Section 15.2)* }

{void Get_error_string(int errorcode, char* name, int& resultlen) *(binding deprecated, see Section 15.2)* }

{void Get_processor_name(char* name, int& resultlen) *(binding deprecated, see Section 15.2)* }

{void Get_version(int& version, int& subversion) *(binding deprecated, see Section 15.2)* }

{void Init() *(binding deprecated, see Section 15.2)* }

{void Init(int& argc, char**& argv) *(binding deprecated, see Section 15.2)* }

{bool Is_finalized() *(binding deprecated, see Section 15.2)* }

{bool Is_initialized() *(binding deprecated, see Section 15.2)* }

{void Comm::Set_errhandler(const Errhandler& errhandler) *(binding deprecated, see Section 15.2)* }

{void File::Set_errhandler(const Errhandler& errhandler) *(binding deprecated, see Section 15.2)* }

{void Win::Set_errhandler(const Errhandler& errhandler) *(binding deprecated, see Section 15.2)* }

{double Wtick() *(binding deprecated, see Section 15.2)* }

{double Wtime() *(binding deprecated, see Section 15.2)* }

};

A.5.7 The Info Object C++ Bindings

namespace MPI {