MPI: A Message-Passing Interface Standard

Version 3.0 ticket0.

(Draft, with MPI 3 Nonblocking Collectives ticket229-A.

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

September 8, 2011

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 ticket0. "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 ticket0. 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 ticket230-B. 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 ticket0. 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.

ticket230-B.

- 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 [45], Express [12], nCUBE's Vertex [41], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [48, 49], 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 [56]. 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 ticket0. 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 ticket0. 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 ticket230-B. nonblocking collective routines.

A new Fortran `mpi_f08` module is introduced to provide extended compile-time argument checking and buffer handling in nonblocking routines. This new Fortran support method provides protection against the optimization problems with with asynchronous acticket232-D. cesses to the buffers of nonblocking calls. The existing `mpi` module is enhanced to provide ticket233-E. basic compile-time argument checking for MPI calls. The use of `mpif.h` is strongly discouraged.

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: ticket230-B.

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

"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6. ticket230-B.

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.

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 BIND(C) 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 mpif.h. The type names are identical to the names in C, except that they are not case sensitive. For example: ticket231-C. ticket231-C. ticket231-C.

```
TYPE, BIND(C) :: MPI_Comm
  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.*)

ticket231-C.

Rationale. Since the Fortran integer values are equivalent, 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. The use of the INTEGER defined handles and the BIND(C) derived type handles is different: Fortran 2003 (and later) define that BIND(C) derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage unit are used for these BIND(C) derived type handles. (*End of rationale.*)

Advice to users. If user wants to substitute mpif.h or the mpi module by the mpi_f08 module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (*End of advice to users.*)

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.

ticket238-J. MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNCHRONOUS_PROTECTS_NONBL (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 ticket234-F. 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 ticket234-F. variable; with the Fortran `mpi_f08` module, such arguments are declared with the Fortran ticket234-F. 2008 + TR 29113 syntax TYPE(*), DIMENSION(..); for C and C++, we use void *.

> *Advice to implementors.* Implementors 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. See details in Section 16.2.1 on page 550. (*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 ticket234-F.
usable in Fortran 77 environments. With the `mpi_f08` module, two new Fortran features, ticket234-F.
assumed type and *assumed rank*, are also required, see Section 2.5.5 on page 16. ticket234-F.

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.

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_[ticket250-V.]FUNCTION
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_[ticket250-V.]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

2.6.2 Fortran Binding Issues

ticket234-F. Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 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.

ticket239-K. ticket250-V. ticket239-K. 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 as a `BIND(C)` derived type with the ticket231-C.
`mpi_f08` module; 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 re- ticket230-B.
spects. These inconsistencies, such as register optimization problems, have implications for ticket230-B.
user codes that are discussed in detail in Section 16.2.16. ticket230-B.

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

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)
```

ticket-248T.

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  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(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, 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)
```

ticket-248T.

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

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

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.

ticket243-O. 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

ticket243-O. code of the received message.

With Fortran `USE mpi_f08`, status is defined as the Fortran `BIND(C)` derived type `TYPE(MPI_Status)` containing three public fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR. `TYPE(MPI_Status)` may contain additional, implementation-specific fields. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code of a received message respectively. Additionally, within both the `mpi` and the `mpi_f08` modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and `TYPE(MPI_Status)` are defined to allow conversion between both status representations. Conversion routines are provided in Section 16.3.5 on page 596.

> *Rationale.* The Fortran `TYPE(MPI_Status)` is defined as a `BIND(C)` derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)

> *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.*)

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

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

```
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: 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_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

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

Start a ready mode nonblocking send.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

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

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

ticket-248T.

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: 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_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

{`MPI::Request MPI::Comm::Irecv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const` *(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 Sections 16.2.10-16.2.20, ticket238-J.
especially in Sections 16.2.12 and 16.2.13 on pages 576-579 about "Problems Due to ticket238-J.
Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", ticket236-H.
and in Sections 16.2.16 to 16.2.19 on pages 581 to 590 about "Optimization Problems", ticket238-J.
"Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

3.7.3 Communication Completion

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

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

We shall use the following terminology: A **null** handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT and MPI_GET_ELEMENTS return count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single

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)
```

ticket-248T.

```
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
              BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  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_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

ticket250-V.

{`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)
```

ticket-248T.

```
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
             BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  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_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

ticket250-V.

{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)
```

ticket-248T.

```
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
             BIND(C)
```

Create (Start Complete)* Free

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

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 576-579 about "Problems Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 581 to 590 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

ticket238-J.
ticket238-J.
ticket236-H.
ticket238-J.

3.10 Send-Receive

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

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

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
```

{`MPI::Aint MPI::Get_address(void* location)` *(binding deprecated, see Section 15.2)* }

This function replaces MPI_ADDRESS, whose use is deprecated. See also Chapter 15. Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions. This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```
   REAL A(100,100)
   INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
   CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, ticket238-J. especially in Sections 16.2.12 and 16.2.13 on pages 576-579 about "Problems Due to ticket238-J. Data Copying and Sequence Association with Subscript Triplets" and "Vector Subscripts", ticket236-H. and in Sections 16.2.16 to 16.2.19 on pages 581 to 590 about "Optimization Problems", ticket238-J. "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	datatype to access (handle)
OUT	num_integers	number of input integers used in the call constructing combiner (non-negative integer)
OUT	num_addresses	number of input addresses used in the call constructing combiner (non-negative integer)
OUT	num_datatypes	number of input datatypes used in the call constructing combiner (non-negative integer)
OUT	combiner	combiner (state)

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)
```

ticket-248T.

```
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
             combiner, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  num_integers, num_addresses, num_datatypes,
    combiner
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

{void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses, int& num_datatypes, int& combiner) const *(binding deprecated, see Section 15.2)* }

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the

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

ticket252-W.

MPI_OP_CREATE(user_fn, commute, op)

IN	[ticket252-W.]user_fn	user defined function (function)
IN	commute	`true` if commutative; `false` otherwise.
OUT	op	operation (handle)

ticket252-W. ticket-248T.

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

MPI_Op_create(user_fn, commute, op, ierror) BIND(C)
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

ticket252-W. ticket252-W.

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

ticket252-W.

{`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.

ticket252-W. 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);
```

ticket230-B. The Fortran declarations of the user-defined function user_fn appear below.

ticket252-W. ticket230-B. ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)
```

```
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), VALUE :: invec, inoutvec
    INTEGER :: len
    TYPE(MPI_Datatype) :: datatype

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

ticket252-W.

ticket252-W.

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. ticket252-W.

> *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

```
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

/* explain to MPI how type Complex is defined
 */
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
/* create the complex-product user-op
 */
MPI_Op_create( myProd, 1, &myOp );

MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);

/* At this point, the answer, which consists of 100 Complexes,
 * resides on process root
 */
```

ticket230-B. **Example 5.21** How to use the mpi_f08 interface of the Fortran MPI_User_function.

```
subroutine my_user_function( invec, inoutvec, len, type )   bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
  type(c_ptr), value :: invec, inoutvec
  integer, intent(in) :: len
  type(MPI_Datatype) :: type
  real, pointer :: invec_r(:), inoutvec_r(:)
  if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
     call c_f_pointer(invec, invec_r, (/ len /) )
     call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
     inoutvec_r = invec_r + inoutvec_r
  end if
end subroutine my_function
```

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

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

ticket250-V.

{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)
```

ticket-248T.

```
MPI_Op_commutative(op, commute, ierror) BIND(C)
    TYPE(MPI_Op), INTENT(IN) :: op
    LOGICAL, INTENT(OUT) :: commute
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

{bool MPI::Op::Is_commutative() const *(binding deprecated, see Section 15.2)* }

5.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

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)
```

ticket-248T.

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

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

ticket250-V.
ticket250-V.

{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)
```

ticket-248T.

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
              extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
    PROCEDURE(MPI_Comm_delete_attr_function) :: 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_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
            EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
{static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function*
            comm_copy_attr_fn,
            MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
            void* extra_state) (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.

With the mpi_f08 module, the Fortran callback functions are: ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Comm) :: oldcomm
      INTEGER :: comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag
```

and ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
  attribute_val, extra_state, ierror) BIND(C)
      TYPE(MPI_Comm) :: comm
      INTEGER :: comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

With the mpi module and mpif.h, the Fortran callback functions are: ticket230-B.

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
```
ticket250-V.

```
             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

ticket250-V.

```
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.*)

ticket230-B.

Advice to implementors. The predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN are defined in the `mpi` module (and `mpif.h`) and the `mpi_f08` module with the same name, but with different interfaces. Each function can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the `mpi` module, i.e., declared as `EXTERNAL`, and the other routine within `mpi_f08` declared with `CONTAINS`. These routines have different link names, which are also different to the link names used for the routines used in C and C++. (*End of advice to implementors.*)

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)
```

ticket-248T.

```
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
              extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) :: 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_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
              EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
{static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function*
              win_copy_attr_fn,
              MPI::Win::Delete_attr_function* win_delete_attr_fn,
              void* extra_state) (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);
```

With the `mpi_f08` module, the Fortran callback functions are: ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Win) ::  oldwin
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

ticket230-B. and

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
  extra_state, ierror) BIND(C)
      TYPE(MPI_Win) ::  win
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

ticket230-B. With the `mpi` module and `mpif.h`, the Fortran callback functions are:

ticket250-V.

```
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

ticket250-V.

```
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_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
             extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: 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_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

{static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
type_delete_attr_fn, void* extra_state) *(binding deprecated, see Section 15.2)* }

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either C, C++, or Fortran. MPI_TYPE_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN from either C, C++, or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
             int type_keyval, void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
             int type_keyval, void *attribute_val, void *extra_state);
```

ticket252-W.

With the mpi_f08 module, the Fortran callback functions are: ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Datatype) :: oldtype
      INTEGER :: type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag
```

and ticket230-B.

```
ABSTRACT INTERFACE
```

ticket-248T.

```
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
  attribute_val, extra_state, ierror) BIND(C)
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

ticket230-B. With the mpi module and mpif.h, the Fortran callback functions are:

ticket250-V.

```
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
```

and

ticket250-V. ticket252-W. ticket252-W.

```
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 C++ callbacks are:

{typedef int
    MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
    int type_keyval, void* extra_state,
    const void* attribute_val_in, void* attribute_val_out,
    bool& flag); *(binding deprecated, see Section 15.2)*}

and

ticket252-W. {typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& datatype,
    int type_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_TYPE_FREE), is erroneous.

MPI_TYPE_FREE_KEYVAL(type_keyval)

INOUT	type_keyval	key value (integer)

```
int MPI_Type_free_keyval(int *type_keyval)
```

ticket-248T.

```
MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
    INTEGER, INTENT(INOUT) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
```

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

MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val) ticket252-W.

INOUT	[ticket252-W.]datatype	datatype to which attribute will be attached (handle)
IN	type_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
              void *attribute_val)
```
ticket252-W.

ticket-248T.

```
MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)
    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_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```
ticket252-W.

ticket252-W.

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

MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag) ticket252-W.

IN	[ticket252-W.]datatype	datatype to which the attribute is attached (handle)
IN	type_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void
              *attribute_val, int *flag)
```
ticket252-W.

ticket-248T.

```
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
              BIND(C)
    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_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```
ticket252-W.

ticket252-W.

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

ticket252-W. MPI_TYPE_DELETE_ATTR(datatype, type_keyval)

INOUT	[ticket252-W.]datatype	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)

ticket252-W.
ticket-248T.

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

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

ticket252-W.
ticket252-W.

```
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, 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 assumes that MPI function results return only error statuses. (*End of advice to users.*)

```
/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
   int ref_count;          /* reference count */
   /* other stuff, whatever else we want */
} gop_stuff_type;
```

- 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) ticket252-W.

INOUT	[ticket252-W.]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)
```
ticket252-W.
ticket-248T.

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

```
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```
ticket252-W.
ticket252-W.

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

MPI_TYPE_GET_NAME (datatype, type_name, resultlen) ticket252-W.

IN	[ticket252-W.]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)
```
ticket252-W.

ticket-248T.

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

ticket252-W.
ticket252-W.

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

{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)
```

ticket-248T.

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

MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

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

MPI_WIN_GET_NAME (win, win_name, resultlen)

IN	win	window whose name is to be returned (handle)
OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
OUT	resultlen	length of returned name (integer)

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
```

ticket-248T.

```
MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) ::  win
```

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 [ticket240-L.]:: 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)
```

ticket-248T.

```
MPI_Get_version(version, subversion, ierror) BIND(C)
    INTEGER, INTENT(OUT) ::  version, subversion
    INTEGER, OPTIONAL, INTENT(OUT) ::  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.*)

ticket247-S.

Rationale. In the `mpi_f08` interface, the string length is defined because the output of this routine is defined only by the MPI library and therefore this routine must not be called with a shorter string buffer. In other routines with string-output arguments, the `LEN=*` may be specified to indicate that shorter strings are possible if the application already knows about a maximum of characters that where stored by the application. (*End of rationale.*)

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)
```

ticket-248T.

```
MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    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
```

ticket245-Q. ticket245-Q. ticket245-Q. ticket245-Q. ticket245-Q.

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER :: INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
    TYPE(C_PTR) :: BASEPTR !overloaded with following...
    INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR ! ...type
```

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

ticket247-S.

With the Fortran mpi modules, MPI_ALLOC_MEM is an INTERFACE with two routines through function overloading: One routine defines baseptr as an INTEGER(KIND=MPI_ADDRESS_KIND), and the second one as TYPE(C_PTR). The first one is without a linker suffix, the second one has _CPTR as linker suffix, see Section 16.2.5 on page 557.

ticket245-Q.

With Fortran mpif.h or if the compiler does not provide the TYPE(C_PTR) interface, only the INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR is required:

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

The Fortran interfaces with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR in the mpi module and the mpif.h include file are deprecated since MPI-3.0.

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)
```

ticket-248T.

```
MPI_Free_mem(base, ierror) BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER 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 `TYPE(C_PTR)` pointer or 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.*)

ticket245-Q.

> *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.*)

ticket245-Q.

Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with `TYPE(C_PTR)` pointers. We assume 4-byte REALs.

```
USE mpi_f08   !  or  USE mpi      (not guaranteed with INCLUDE 'mpif.h')
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_PTR) :: p
REAL, DIMENSION(:,:), POINTER :: a             ! no memory is allocated
INTEGER, DIMENSION(2) :: shape
INTEGER(KIND=MPI_ADDRESS_KIND) :: size
shape = (/100,100/)
size = 4 * shape(1) * shape(2)                 ! assuming 4 bytes per REAL
CALL MPI_Alloc_mem(size,MPI_INFO_NULL,p,ierr) ! memory is allocated and
```

```
CALL C_F_POINTER(p, a, shape) ! intrinsic     ! now accessible via a(i,j)
...                           ! in ISO_C_BINDING
a(3,5) = 2.71;
...
CALL MPI_Free_mem(a, ierr)                    ! memory is freed
```

Example 8.2 Example of use of MPI_ALLOC_MEM, in Fortran with non-standard *Cray-pointer*. We assume 4-byte REALs, and assume that these pointers are address-sized.

ticket245-Q. ticket245-Q. ticket245-Q.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
[ticket245-Q.]INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
[ticket245-Q.]SIZE = 4*100*100
CALL MPI_ALLOC_MEM([ticket245-Q.]SIZE, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs `-fcray-pointer`.

ticket245-Q. ticket245-Q. ticket245-Q. ticket245-Q.

> *Advice to implementors.* Some compilers map Cray-pointer to address-sized integers, some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointer are available. (*End of advice to implementors.*)

Example 8.3 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.

MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C++, the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

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

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

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

The syntax for these calls is given below.

8.3.1 Error Handlers for Communicators

MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler) ticket252-W.

IN	[ticket252-W.]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)
```
ticket252-W.
ticket-248T.

```
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)
    PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```
ticket252-W.
ticket252-W.

```
{static MPI::Errhandler
              MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_function*
              comm_errhandler_fn) (binding deprecated, see Section 15.2) }
```
ticket252-W.

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.

ticket230-B. ticket-248T. With the Fortran `mpi_f08` module, the user routine comm_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  error_code
```

ticket230-B. ticket230-B. With the Fortran `mpi` module and `mpif.h`, the user routine COMM_ERRHANDLER_FN 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)
```

ticket-248T.

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

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

ticket-248T.

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

```
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, 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) ticket252-W.

IN	[ticket252-W.]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)
```

ticket252-W.
ticket-248T.

```
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)
    PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

ticket252-W.
ticket252-W.

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

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

ticket252-W.

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.

ticket230-B.
ticket-248T.

With the Fortran `mpi_f08` module, the user routine win_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
      TYPE(MPI_Win) ::  win
      INTEGER ::  error_code
```

ticket230-B.
ticket230-B.

With the Fortran `mpi` module and `mpif.h`, the user routine WIN_ERRHANDLER_FN 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)`

ticket-248T.

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

```
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, 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)
```

ticket-248T.

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

```
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, 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) ticket252-W.

IN	[ticket252-W.]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)
```

ticket252-W.
ticket-248T.

```
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)
    PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

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

ticket252-W.
ticket252-W.

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

ticket252-W.

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.

ticket230-B. ticket-248T. With the Fortran mpi_f08 module, the user routine file_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)
      TYPE(MPI_File) ::  file
      INTEGER ::  error_code
```

ticket230-B. ticket230-B. With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN 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)
```

ticket-248T.

```
MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) ::  file
    TYPE(MPI_Errhandler), INTENT(IN) ::  errhandler
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

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

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.

MPI_FILE_GET_ERRHANDLER(file, errhandler)

IN	file	file (handle)
OUT	errhandler	error handler currently associated with file (handle)

```
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
```

ticket-248T.

```
MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)
```

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. ticket231-C. ticket231-C.

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

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 355.

ticket231-C.
ticket231-C.

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 372). 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 Fortran, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. In C++ this constant does not exist, and the array_of_errcodes argument may be omitted from the argument list.

Advice to implementors. MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4 on page 15. (*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 16.2.16.

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 ticket238-J. `COMMON` blocks. To prevent problems with the argument copying and register optimization ticket238-J. done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 576-579 about "Problems Due to Data Copying and ticket236-H. Sequence Association with Subscript Triplets" and "Vector Subscripts", and in Sections 16.2.16 ticket238-J. to 16.2.19 on pages 581 to 590 about "Optimization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". Sections "Solutions" to "VOLATILE" on pages 585-588 discuss several solutions for the problem in this example.

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)
```

ticket-248T.

```
MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
              ierror) BIND(C)
    PROCEDURE(MPI_Grequest_query_function) ::  query_fn
    PROCEDURE(MPI_Grequest_free_function) ::  free_fn
    PROCEDURE(MPI_Grequest_cancel_function) ::  cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
              IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
{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 ticket0. state for the request.

In C, the query function is

```
typedef int MPI_Grequest_query_function(void *extra_state,
            MPI_Status *status);
```

in Fortran with the mpi_f08 module ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
  BIND(C)
      TYPE(MPI_Status) ::  status
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state
      INTEGER ::  ierror
```

in Fortran with the mpi module and mpif.h ticket230-B.

```
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)*}

ticket0.

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). ticket0.

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, 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);
```

in Fortran with the mpi_f08 module ticket230-B.

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
```

```
        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        INTEGER :: ierror
```

ticket230-B. in Fortran with the `mpi` module and `mpif.h`

ticket230-B.

```
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)*}

ticket0. The free_fn function is invoked to clean up user-allocated resources when the generalized request is freed.

ticket0. 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.

ticket0. 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 ticket0. 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);
```

ticket230-B. in Fortran with the `mpi_f08` module

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
  BIND(C)
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
      LOGICAL :: complete
```

```
    INTEGER :: ierror
```

in Fortran with the `mpi` module and `mpif.h` ticket230-B.

```
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)*}

ticket0.

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 ticket0. 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, 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)
```

ticket-248T.

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

```
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

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

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 486).

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 508, 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 112. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 462). 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 — Sections 16.2.10- ticket238-J.
> 16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 576-579 about "Problems ticket238-J.
> Due to Data Copying and Sequence Association with Subscript Triplets" and "Vector ticket236-H.
> Subscripts", and in Sections 16.2.16 to 16.2.19 on pages 581 to 590 about "Opti- ticket238-J.

mization Problems", "Code Movements and Register Optimization", "Temporary Data Movements" and "Permanent Data Movements". (*End of advice to users.*)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS, respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. In C++, the status argument is optional. The status can be passed to MPI_TEST_CANCELLED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

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)
```

ticket-248T.

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
    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) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/ MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 16.2.17 on page 582, but not all of the problems described in Section 16.2.16 on page 581. ticket238-J. ticket238-J. ticket238-J.
- 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

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The size of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 16.2.9, page 570.

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)
```

ticket-248T.

```
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
            dtype_file_extent_fn, extra_state, ierror) BIND(C)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
```

```
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: 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
```

```
{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 514). 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);
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror) BIND(C)
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, 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);
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
  filebuf, position, extra_state, ierror) BIND(C)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE :: userbuf, filebuf
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: count, ierror
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
      INTEGER(KIND=MPI_ADDRESS_KIND) :: 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.*)

Collective file operations are collective over a dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

13.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

> *Advice to users.* In most cases, use of MPI_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

13.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

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 ticket230-B.
MPI_OFFSET_KIND, which is defined in the `mpi_f08` module, the `mpi` module and the `mpif.h` ticket230-B.
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 591).

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

Chapter 14

Profiling Interface

14.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI ticket0. functions *must*

1. provide a mechanism through which all of the MPI defined functions, except those ticket0. allowed as macros (See Section 2.6.5), may be accessed with a name shift. This ticket0. requires, in C and Fortran, an alternate entry point name, with the prefix `PMPI_` for each MPI function. The profiling interface in C++ is described in Section 16.1.10. For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version. ticket247-S.

   For Fortran, the different support methods cause several linker names. Therefore, several profiling routines (with these linker names) are needed for each Fortran MPI routine, as described in Section 16.2.5 on page 557.

2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.

3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by imple- ticket0. menting it only for the lowest level routines.

4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of "wrapper" functions that call the C ticket0. implementation), ensure that these wrapper functions are separable from the rest of the library.

   This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. provide a no-op routine MPI_PCONTROL in the MPI library.

When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be `ar`ed out of the ticket247-S. base library and into the profiling one.

Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support `TYPE(*), DIMENSION(..)` choice buffers) imply different linker names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 16.2.5 on page 557.

14.5 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

ticket0.
- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we ticket206. decided to accept the scheme outlined above.

ticket206. Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^NMPI tool infrastructure [46].

Chapter 15

Deprecated Functions

15.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HVECTOR in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

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

MPI_TYPE_HVECTOR() ticket-248T.

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HINDEXED in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

```
    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) ticket252-W.

IN	[ticket252-W.]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)
```
ticket252-W.

ticket-248T.

```
MPI_ERRHANDLER_CREATE()
```

```
MPI_ERRHANDLER_CREATE(HANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL HANDLER_FN
    INTEGER ERRHANDLER, IERROR
```
ticket252-W.
ticket252-W.

Register the user routine handler_fn for use as an MPI exception handler. Returns in errhandler a handle to the registered exception handler. ticket252-W.

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)
```

ticket-248T.

```
MPI_ERRHANDLER_SET()
```

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

15.3 Deprecated since MPI-3.0

ticket245-Q.

The Fortran interfaces of MPI_ALLOC_MEM with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR in the mpi module and the mpif.h include file are deprecated since MPI-3.0. In the mpi module, the deprecated interface is overloaded with an interface that returns a TYPE(C_PTR) baseptr, see Section 8.2 of page 326.

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 with additional features from Fortran 2003 and Fortran 2008 [34] + TR 29113 [36].

> *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 + TR 29113, the major new language features used are the `ASYNCHRONOUS` attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 16.2.7 on page 563. (*End of rationale.*)

MPI defines three methods of Fortran support:

1. **USE mpi_f08:** This method is described in Section 16.2.2 and requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls.
2. **USE mpi:** This method is described in Section 16.2.3 and requires compile-time argument checking. Handles are defined as `INTEGER`.
3. **INCLUDE 'mpif.h':** This method is described in Section 16.2.4. The use of the include file `mpif.h` is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls.

Compliant MPI-3 implementations providing a Fortran interface must provide all three Fortran support methods. Section 16.2.6 on page 559 describes restrictions if the compiler does not support all the needed features.

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

> *Advice to users.* Users are advised to utilize one of the MPI modules even if `mpif.h` enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the `mpi_f08` module offers the most advantages. (*End of advice to users.*)

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

The `INTEGER` compile-time constant MPI_SUBARRAYS_SUPPORTED is set to `.TRUE.` if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [36]; otherwise it is set to `.FALSE.`. The `INTEGER` compile-time constant MPI_ASYNCHRONOUS_PROTECTS_NONBL is set to `.TRUE.` if the `ASYNCHRONOUS` attribute was added to the choice buffer arguments of all nonblocking interfaces **and** the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113), otherwise it is set to `.FALSE.`. These constants exist with each Fortran support method, but not in the C/C++ header files. The values may be different for each Fortran support method.

Section 16.2.2 through 16.2.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 16.2.5 defines the corresponding full interface specification together with the used linker names and implications for the profiling interface. Section 16.2.6 the implementation of the MPI routines for different versions of the Fortran standard. Section 16.2.7 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 16.2.8 and Section 16.2.9 describe additional functionality that is part of the Fortran support. MPI_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: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. In the context of MPI, parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters. Sections 16.2.10 through 16.2.19 give an overview and details on known problems when using Fortran together with MPI; Section 16.2.20 compares the Fortran problems with those in C.

16.2.2 Fortran Support Through the `mpi_f08` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi_f08` that can be used in a Fortran program. Section 16.2.6 on page 559 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not `TYPE(*)`, with the following exception:
- Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the `mpi` module, see Section 16.2.3 on page 554.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from `mpif.h` or the `mpi` module to the `mpi_f08` module. (*End of advice to users.*)

ticket231-C.

- Define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).

ticket241-M.
ticket238-J.

- Use the `ASYNCHRONOUS` attribute to protect the buffers of nonblocking operations, and set the `INTEGER` compile-time constant MPI_ASYNCHRONOUS_PROTECTS_NONBL to `.TRUE.` if the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113). See Section 16.2.6 on page 559 for older compiler versions.

ticket234-F.

- Set the `INTEGER` compile-time constant MPI_SUBARRAYS_SUPPORTED to `.TRUE.` and declare choice buffers using the Fortran 2008 TR 29113 feature assumed-type and assumed-rank, i.e., `TYPE(*), DIMENSION(..)`, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TR 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 16.2.6 on page 559 for details.

ticket242-N.

- Declare each argument with an `INTENT` of `IN`, `OUT`, or `INOUT` as defined in this standard.

ticket242-N.

Rationale. For these definitions in the `mpi_f08` bindings, in most cases, `INTENT(IN)` is used if the C interface uses call-by-value. For all buffer arguments and for `OUT` and `INOUT` dummy arguments that allow one of the non-ordinary Fortran constants (see MPI_BOTTOM, etc. in Section 2.5.4 on page 15) as input, an `INTENT` is not specified. (*End of rationale.*)

ticket242-N.

Advice to users. If a dummy argument is declared with `INTENT(OUT)`, then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [34], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of `INTENT(OUT)` the copy-in may be suppressed by the optimization and the routine is starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 16.2.3 on page 554. (*End of advice to users.*)

ticket239-K.

- Declare all ierror output arguments as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and predefined callbacks (e.g., MPI_NULL_COPY_FN). ticket250-V. ticket239-K. ticket239-K.

  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. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

ticket230-B.

The MPI Fortran bindings in the `mpi_f08` module are designed based on the Fortran 2008 standard [34] together with the Technical Report (TR 29113) on Further Interoperability with C [36] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TR 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to .[35] page iv, last paragraph, “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

The TR 29113 contains the following language features that are needed for the MPI bindings in the `mpi_f08` module: assumed-type and assumed-rank. 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`, `CHARACTER*5`, `CHARACTER*(*)`, sequence derived types, or `BIND(C)` derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with `mpif.h` without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the `mpi_f08` module).

The `INTERFACE` construct in combination with `BIND(C)` allows the implementation of the Fortran `mpi_f08` interface with a single set of portable wrapper routines written in C, which supports all desired features in the `mpi_f08` interface. TR 29113 also has a provision for `OPTIONAL` arguments in `BIND(C)` interfaces.

A further feature useful for MPI is the extension of the semantics of the `ASYNCHRONOUS` attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TR29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

ticket230-B. ticket230-B. ticket230-B. ticket230-B. ticket230-B.

16.2.3 Fortran Support Through the `mpi` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants

ticket230-B.

- Declare MPI functions that return a value.

- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists.

ticket231-C.

- Define all MPI handles as type `INTEGER`.

ticket243-O. ticket231-C. ticket231-C.

- Define all named handle types and the derived type `MPI_Status` that are used in the `mpi_f08` module.

  Rationale. They are needed only when the application converts old-style `INTEGER` handles into new-style handles with a named type. (*End of rationale.*)

ticket238-J.

- A high quality MPI implementation may enhance the interface by using the `ASYNCHRONOUS` attribute in the same way as in the `mpi_f08` module if it is supported by the underlying compiler.

- Set the `INTEGER` compile-time constant MPI_ASYNCHRONOUS_PROTECTS_NONBL to `.TRUE.` if the `ASYNCHRONOUS` attribute is used in all nonblocking interfaces **and** the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113), otherwise to `.FALSE.`.

ticket247-S.

  Advice to users. For an MPI implementation that fully supports nonblocking calls with the `ASYNCHRONOUS` attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses *contiguous* but not *simply contiguous* `ASYNCHRONOUS` arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying `(1:n)` for a whole dimension instead of using `(:)`. This should be fixed to fulfill the Fortran constaints for `ASYNCHRONOUS` dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the `ASYNCHRONOUS` attribute to protect nonblocking calls. Onother reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copyin/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. (*End of advice to users.*)

ticket232-D.

- A high quality MPI implementation may enhance the interface by using `TYPE(*), DIMENSION(..)` choice buffer dummy arguments instead of using non-standardized extensions such as `!$PRAGMA IGNORE_TKR` or a set of overloaded functions as described by M. Hennecke in [26], if the compiler supports this TR 29113 language feature. See Section 16.2.6 on page 559 for further details.

- Set the `INTEGER` compile-time constant MPI_SUBARRAYS_SUPPORTED to `.TRUE.` if all choice buffer arguments are declared with `TYPE(*), DIMENSION(..)`, otherwise set it to `.FALSE.`. With MPI_SUBARRAYS_SUPPORTED==`.TRUE.`, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TR 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 16.2.6 on page 559 for details.

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 may provide `INTENT` information in these interface blocks. ticket230-B. ticket230-B. ticket230-B. ticket230-B.

Advice to implementors. The appropriate `INTENT` may be different from what is given in the MPI language-neutral bindings. Implementations must choose `INTENT` so that the function adheres to the MPI standard, e.g., by defining the `INTENT` as provided in the `mpi_f08` bindings. (*End of advice to implementors.*) ticket230-B. ticket242-N.

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 was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an `OUT` argument to be `INOUT`. (*End of rationale.*) ticket230-B. ticket230-B. ticket250-V. ticket242-N.

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that *"INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument.* Furthermore, *INTENT(INOUT) is not equivalent to omitting the INTENT attribute*, because *INTENT(INOUT) always requires* that the associated actual argument is *definable*". Applications that include `mpif.h` may not expect that `INTENT(OUT)` is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use `INTENT(OUT)` in the `mpi` module and the `mpif.h` include file, even though `INTENT(OUT)` is specified in an interface description of the `mpi_f08` module. (*End of advice to implementors.*)

ticket230-B. ticket232-D.

16.2.4 Fortran Support Through the `mpif.h` Include File

ticket230-B. ticket230-B.

ticket233-E. The use of the `mpif.h` include file is strongly discouraged and may be deprecated in a future version of MPI.

ticket230-B. ticket230-B. ticket230-B. ticket230-B. An MPI implementation providing a Fortran interface must provide an include file named `mpif.h` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as `INTEGER`.
- 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 for the second Fortran binding (in deprecated routines, the first one may be omitted).

- Set the `INTEGER` compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNCHRONOUS_PROTECTS_NONBL according to the same rules as for the `mpi` module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to `.FALSE.`.

  Advice to users. Instead of using `mpif.h`, the use of the `mpi_f08` or `mpi` module is strongly encouraged for the following reasons:

  - Most `mpif.h` implementations do not include compile-time argument checking.
  - Therefore, too many bugs in MPI applications remain undetected at compile-time, such as:
    - Missing ierror as last argument in most Fortran bindings.
    - Declaration of a status as an `INTEGER` variable instead of an `INTEGER` array with size MPI_STATUS_SIZE.
    - Wrong argument positions; e.g., interchanging the count and datatype arguments.
    - Passing wrong MPI handles; e.g., passing a datatype instead of a communicator.
  - The migration from `mpif.h` to the `mpi` module should be relatively straightforward (i.e., substituting `include 'mpif.h'` after an `implicit` statement by `use mpi` before such `implicit` statement) as long as the application syntax is correct.
  - Migrating portable and correctly written applications to the `mpi` module is not expected to be difficult. No compile or runtime problems should occur because an `mpif.h` include file was always allowed to provide explicit Fortran interfaces.

  (*End of advice to users.*)

  Rationale. With MPI-3.0, the `mpif.h` include file was not deprecated in order to retain strong backward compatibility. Internally, `mpif.h` and the `mpi` module may be implemented so that the same (or similar) library implementation of the MPI routines can be used. (*End of rationale.*)

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 the requirement of usability in free and fixed [ticket230-B.] source form applications be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` may contain only declarations, and [ticket230-B.] because common block declarations can be split among several lines. The argument [ticket230-B.] names may need to be shortened to keep the `SUBROUTINE` statement within the allowed 72-6=66 characters, e.g.,

```
INTERFACE
SUBROUTINE MPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
   ...  !  dummy argument declarations
```

This line has 65 characters and is the longest in MPI-3.0.

TODO: This is only checked for MPI-2.2. We have to check all new MPI-3.0 interfaces that they stay within these 66 characters. Otherwise the routine name should be shortened before the name is standardized.

If `mpif.h` contains also explicit interfaces with `BIND(C,NAME='...')` for providing MPI_SUBARRAYS_SUPPORTED and MPI_ASYNCHRONOUS_PROTECTS_NONBL equals `.TRUE.`, the linker routine name may need to be shortened. For example, MPI_FILE_WRITE_AT_ALL_BEGIN with 6 arguments, may be defined:

```
INTERFACE MPI_FILE_WRITE_AT_ALL_BEGIN
SUBROUTINE MPI_X(a,b,c,d,e,f)BIND(C,NAME='MPI_File_write_at_all_begin_f')
   ...  !  dummy argument declarations
```

This would need a line length of 73 characters, i.e., the C routine name must be shortened by 7 characters to stay within the available 66 characters. **TODO: Do we want to define these shortened routine names for mpif.h; this would help the tools people.** Note that the name `MPI_X` has no meaning for the compilation, and that this problem occurs only with routines with choice buffers implemented with the assumed-type and assumed-rank facility of TR 29113. To support Fortran 77 as well as Fortran 90 and later, it may be necessary to eliminate all comments from [ticket230-B.] `mpif.h`. (*End of advice to implementors.*)

16.2.5 Interface Specifications, Linker Names and the Profiling Interface [ticket247-S.]

[ticket247-S.]

The Fortran interface specifications of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attribbutes. The rules for the linker names and its implications for the profiling interface are specified within this section. The linker name of a Fortran routine is defined as the name that a C routine would have if both routines would have the same name visible for the linker. A typical linker name of the Fortran routine FOOfoo is `foofoo__`. In the case of `BIND(C,NAME='...')`, the linker name is directly defined through the given string.

The following rules for linker names apply:

- With the Fortran `mpi_f08` module, if MPI_SUBARRAYS_SUPPORTED equals `.TRUE.`:

  The Fortran binding must use BIND(C) interfaces with an interface name identical to the language independent name, e.g., MPI_SEND. The linker name is a combination of the C name and an `_f08` suffix, e.g., `MPI_Send_f08`. Prototype example:

```
INTERFACE
  SUBROUTINE MPI_Send(...)  BIND(C,NAME='MPI_Send_f08')
```

- With the Fortran `mpi_f08` module, if MPI_SUBARRAYS_SUPPORTED equals `.FALSE.` (i.e., with a preliminary implementation of this module without TR 29113):

  The linker name of each routine is defined through the linker name mapping of the Fortran compiler for the name defined when subarrays are supported. For example, `MPI_Send_f08` may be mapped to `mpi_send_f08__`. Example:

  ```
  INTERFACE MPI_Send
    SUBROUTINE MPI_Send_f08(...)
  ```

- With the Fortran `mpi` module or `mpif.h` include file, if MPI_SUBARRAYS_SUPPORTED equals `.FALSE.`:

  The linker name of each routine is defined through the linker-name mapping of the Fortran compiler. For example, MPI_SEND may be mapped to `mpi_send__`. Example:

  ```
  INTERFACE
    SUBROUTINE MPI_SEND(...)
  ```

- With the Fortran `mpi` module or `mpif.h` include file, if MPI_SUBARRAYS_SUPPORTED equals `.TRUE.`:

  The Fortran binding must use BIND(C) interfaces with an interface name identical to the language independent name, e.g., MPI_SEND. The linker name is a combination of the C name and an `_f` suffix, e.g., `MPI_Send_f`. Prototype example:

  ```
  INTERFACE
    SUBROUTINE MPI_SEND(...)  BIND(C,NAME='MPI_Send_f')
  ```

If the support of subarrays is different for the `mpi` module and the `mpif.h` include file, then both linker-name methods can be used in the same application. If the application also uses the `mpi_f08` module and was compiled with this module partially before and after the subarrays were supported, then all four interfaces are used within the same application.

Rationale. After a compiler provides the facilities from TR29113, i.e., `TYPE(*), DIMENSION(..)`, it is possible to change the bindings within a Fortran support method to support subarrays and without recompiling the complete application. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own linker names and all interfaces use the same constants and type definitions. (*End of rationale.*)

A user-written or middleware profiling routine that is written according to the same binding rules will have the same linker name, and therefore, can interpose itself as the MPI library routine. The profiling routine can internally call the matching `PMPI` routine with any of its existing bindings, except for routines that have callback routine dummy arguments. In this case, the profiling software must use the same Fortran support method as used in the calling application program, because the C, `mpi_f08` and `mpi` callback prototypes are different.

Advice to users. This advice is mainly for tool writers. Even if an MPI library supports subarrays in all three Fortran support methods, a portable profiling layer should also provide the two interfaces for MPI_SUBARRAYS_SUPPORTED==.FALSE. to support older binary user routines that were compiled before TR29113 level support was achieved.

If a user application calls MPI_SEND, then the chosen Fortran support method together with the MPI implement decision about MPI_SUBARRAYS_SUPPORTED imply, to which linker name the compiler will translate this call, i.e., whether the application calls mpi_send__, or MPI_Send_f, or mpi_send_f08__, or MPI_Send_f08. If the profiling layer wants to be independent of the decision of the user program and MPI implementation, then it should provide all four routines. For example:

```
SUBROUTINE MPI_SEND(...)  BIND(C,NAME='MPI_Send_f')
  USE mpi
  CALL PMPI_SEND(...)
END SUBROUTINE
```

The MPI library must provide the PMPI_SEND routine according to the same rules as for providing the MPI_SEND routine. (*End of advice to users.*)

Advice to implementors. If an implementation provides in a first step two sets of routines, one for the mpi module and mpif.h, and the other for the mpi_f08 module, and both sets without TR 29113, i.e., MPI_SUBARRAYS_SUPPORTED equals .FALSE.. If the implementor wants to add a TR 29113 based set of routines, then it is not necessary to add two full sets of routines. For full quality, it is enough to implement in each set only those routines that have a choice buffer argument. (*End of advice to implementors.*)

In the case that a Fortran binding consists of multiple routines through function overloading, the base names of overloaded routines are appended by a suffix notifying the difference in the argument list. For example, MPI_ALLOC_MEM (in the mpi module and mpif.h) has an INTEGER(KIND=...) baseptr argument without a suffix. This routine is overloaded by a routine with TYPE(C_PTR) baseptr and the suffix _CPTR. The implied linker name base is MPI_ALLOC_MEM_CPTR. It is mapped to the linker names MPI_Alloc_mem_cptr_f, and, e.g., mpi_alloc_mem_cptr__. Note that these routines are always called via the interface name MPI_ALLOC_MEM by the application within all Fortran support methods.

16.2.6 MPI for Different Fortran Standard Versions

ticket247-S.
ticket247-S.

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
  - MPI identifiers are limited to thirty or more, not six, significant characters.
  - MPI identifiers may contain underscores after the first character.
  - An MPI subroutine with a choice argument may be called with different argument types.
  - Although not required b the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)

- For Fortran 90:
  The major additional features that are needed from Fortran 90 are:
  - The MODULE and INTERFACE concept.
  - The KIND= and SELECTED_...._KIND concept.
  - Fortran derived TYPEs and the SEQUENCE attribute.
  - The OPTIONAL attribute for dummy arguments.
  - Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.

  With these features, MPI-1.1 - MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with S1 = the mpi_f08 module, S2 = the mpi module, and S3 = the mpif.f include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.

  - MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
  - S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
  - In this preliminary interface of S1, the following changes are necessary:
    - The routines are not BIND(C).
    - TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !$PRAGMA IGNORE_TKR.
    - The ASYNCHRONOUS attribute is omitted.
    - PROCEDURE(...) callback declarations are substituted by EXTERNAL.
  - The linker names are specified in Section 16.2.5 on page 557.
  - Due to the rules specified in Section 16.2.5 on page 557, choice buffer declarations should be implemented only with non-standardized extensions like !$PRAGMA IGNORE_TKR (as long as F2008+TR29113 is not available).

    In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 16.2.11 on page 575). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 16.2.15 on page 580.

    Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the mpi_f08 module if such an extension is not available.

  - The ASYNCHRONOUS attribute can **not** be used in applications to protect buffers in nonblocking MPI calls (S1-S3).
  - The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.
  - In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types ralated to TYPE(MPI_Status) (see Section 16.3.5 on page 596) are also not available in the mpi module and mpif.h.

- For Fortran 95:
  The quality of the MPI interface and the restrictions are the same as with Fortran 90.
- For Fortran 2003:
  The major features that are needed from Fortran 2003 are:
  - Interoperability with C, i.e.,
    * BIND(C, NAME='...') interfaces.
    * BIND(C) derived types.
    * The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.
  - The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
  - The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TR 29113.

  With these features (but still without the features of TR29113), MPI-1.1 - MPI-2.2 can be implemented without restrictions, but with one enhancement:

  - The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void * argument.

  MPI-3.0 can be implemented with the following restrictions:

  - MPI_SUBARRAYS_SUPPORTED equals .FALSE..
  - For S1, only a preliminary implementation is possible. The following changes are necessary:
    * The routines are not BIND(C).
    * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !$PRAGMA IGNORE_TKR.

- The linker names are specified in Section 16.2.5 on page 557.
- With S1, the ASYNCHRONOUS is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.
- The ASYNCHRONOUS Fortran attribute can be used in applications to **try to** protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.
- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines can be used only for Fortran types that are C compatible.
- The same restriction as for Fortran 90 applies if non-standardized extensions like !$PRAGMA IGNORE_TKR are not available.

- For Fortran 2008 + TR 29113 and later and
For Fortran 2003 + TR 29113:
The major feature that are needed from TR29113 are:
  - TYPE(*), DIMENSION(..) is available.
  - The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI communication.
  - OPTIONAL dummy arguments are allowed in combination with BIND(C) interfaces.
  - CHARACTER(LEN=*) dummy arguments are allowed in combination with BIND(C) interfaces.
  - The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.

  Using these features, MPI-3.0 can be implemented without any restrictions.
  - With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE.. The ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines can be used for any Fortran type.
  - With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.
  - If non-standardized extensions like !$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION(..).

ticket234-F.

Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice argument may be implemented with an explicit interface using compiler directives, for example:

```
INTERFACE
  SUBROUTINE MPI_...(buf, ...)
    !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
    !$PRAGMA IGNORE_TKR buf
```

```
      !DIR$ IGNORE_TKR buf
      !IBM* IGNORE_TKR buf
      REAL, DIMENSION(*) :: buf
      ... ! declarations of the other arguments
    END SUBROUTINE
  END INTERFACE
```

(*End of advice to implementors.*)

16.2.7 Requirements on Fortran Compilers ticket230-B.

ticket230-B.

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

> *Advice to users.* Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., `mpif90`, `mpicc`) that set these options automatically. (*End of advice to users.*)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 16.2.11 through 16.2.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`) are: ticket234-F.

- The language features assumed-type and assumed-rank from Fortran 2008 TR 29113 [36] are available. This is required only for `mpi_f08`. As long as this requirement is not supported by the compiler, it is valid to build a preliminary MPI-3.0 (and not later) library that implements the `mpi_f08` module with MPI_SUBARRAYS_SUPPORTED set to `.FALSE.`. ticket230-B.
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the `ASYNCHRONOUS` attribute. See Section 16.2.12 on page 576 for more details. ticket237-I.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== `.FALSE.`, they are passed with call by reference, and passed by descriptor in the case of `.TRUE.`. ticket232-D.

ticket234-F.

- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., `CHARACTER(LEN=*)` strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods. ticket246-Q.
- The array dummy argument of the `ISO_C_BINDING` intrinsic module procedure `C_F_POINTER` is not restricted to Fortran types for which a corresponding type in C exists. ticket238-J.

- The Fortran compiler shall not provide `TYPE(*)` unless the `ASYNCHRONOUS` attribute protects MPI communication as described in TR 29113. Specifically, the TR 29113 must be implemented as a whole.

ticket238-J.

The following rules are required at least as long as the compiler does not provide the extension of the `ASYNCHRONOUS` attribute as part of TR 29113 and there is still one Fortran support method with MPI_ASYNCHRONOUS_PROTECTS_NONBL==`.FALSE.`. It is helpful when these rules are observed, especially for backward compatibility of existing applications ticket238-J. that use the `mpi` module or the `mpif.h` include file. The rules are as follows:

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with `BIND(C)` Fortran interfaces (e.g., MPI_F_SYNC_REG on page 586 and Section 16.2.8 on page 565, and DD on page 588) solve the problems ticket238-J. described in Section 16.2.17 on page 582.

- The problems with temporary data movement (described in detail in Section 16.2.18 on page 589) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective IO) and the ticket238-J. computation when overlapping communication and computation.

- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 16.2.19 on page 590) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in calling MPI operations.

ticket230-B.

All of these rules are valid independently of whether the MPI routine interfaces in the `mpi_f08` and `mpi` modules are internally defined with an `INTERFACE` or `CONTAINS` construct, and with or without `BIND(C)`, and also when `mpif.h` uses explicit interfaces.

> *Advice to implementors.* Some of these rules are already part of the Fortran 2003 standard if the MPI interfaces are defined without `BIND(C)`. Additional compiler support may be necessary if `BIND(C)` is used. Some of these additional requirements are defined in the Fortran 2008 TR 29113 [36]. Some of these requirements for MPI-3.0 are beyond the scope of TR 29113. (*End of advice to implementors.*)

Further requirements apply when the MPI library internally uses `BIND(C)` routine interfaces ticket230-B. (i.e, for a full implementation of `mpi_f08`):

- Non-buffer arguments are `INTEGER`, `INTEGER(KIND=...)`, `CHARACTER(LEN=*)`, ticket231-C. `LOGICAL`, and `BIND(C)` derived types, (handles and status in `mpi_f08`) variables and ticket230-B. arrays; function results are `DOUBLE PRECISION`. All these types must be valid as dummy arguments in the `BIND(C)` MPI routine interfaces. When compiling an MPI application, the compiler should not issue warnings inidicating that these types may not be interoperable with an existing type in C. Some of these types are already valid in `BIND(C)` interfaces since Fortran 2003, some may be valid based on TR 29113 (e.g., ticket239-K. `CHARACTER*(*)`).

- `OPTIONAL` dummy arguments are also valid within `BIND(C)` interfaces. This requirement is fulfilled if TR 29113 is fully supported by the compiler.

ticket238-J.
ticket238-J.

16.2.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 16.2.17 on page 582, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf)

INOUT	buf	initial address of buffer (choice)

ticket-248T.

```
MPI_F_sync_reg(buf) BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
```

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

This routine is a no-operation. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

> *Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

> *Advice to implementors.* This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the `mpi_f08` module according to the definition for the `mpi` module or `mpif.h` to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (*End of advice to implementors.*)

> *Rationale.* This routine is not defined with `TYPE(*), DIMENSION(*)`, i.e., assumed size instead of assumed rank, because this would restrict the usability to simply contiguous arrays and would require overloading with another interface for scalar arguments. (*End of rationale.*)

> *Advice to users.* If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not to be called if MPI_ASYNCHRONOUS_PROTECTS_NONBL is `.TRUE.` and the application fully uses the facilities of `ASYNCHRONOUS` arrays. (*End of advice to users.*)

16.2.9 Additional Support for Fortran Numeric Intrinsic Types

ticket230-B.

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 CHARACTER) with an optional integer KIND parameter that selects from among one or more variants. The specific meaning of different KIND values themselves are implementation dependent and not specified by the language. Fortran provides the KIND selection functions selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX and INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:

```
double precision x
real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods to communicate using numeric intrinsic types. ticket230-B. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion ticket230-B. in heterogeneous environments. The second method (see Support for size-specific MPI Datatypes on page 570) gives the user complete control over communication by exposing machine representations.

Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types — MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND parameter, where p is decimal digits of precision and r is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is defined for each value of (p, r) supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index (p, r) not supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX datatypes. For integers, there is a similar implicit array related to selected_int_kind and indexed by the requested number of digits r. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but a new set.

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

Advice to users. `selected_real_kind()` maps a large number of `(p,r)` pairs to a much smaller number of `KIND` parameters supported by the compiler. `KIND` parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and `KIND` parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same `(p,r)` value (`REAL` and `COMPLEX`) or `r` value (`INTEGER`). Thus MPI has many more datatypes than there are fundamental language types. (*End of advice to users.*)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	p	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

ticket-248T.

```
MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)
    INTEGER, INTENT(IN) :: p, r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

{`static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r)` *(binding deprecated, see Section 15.2)* }

This function returns a predefined MPI datatype that matches a `REAL` variable of `KIND` `selected_real_kind(p, r)`. In the model described above it returns a handle for the element `D(p, r)`. Either `p` or `r` may be omitted from calls to `selected_real_kind(p, r)` (but not both). Analogously, either `p` or `r` may be set to MPI_UNDEFINED. In communication, an MPI datatype `A` returned by MPI_TYPE_CREATE_F90_REAL matches a datatype `B` if and only if `B` was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for `p` and `r` or `B` is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 570.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	p	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
```

ticket-248T.

```
MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
    INTEGER, INTENT(IN) :: p, r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

{static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r) *(binding deprecated, see Section 15.2)* }

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 570.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYPE_CREATE_F90_INTEGER(r, newtype)

IN	r	decimal exponent range, i.e., number of decimal digits (integer)
OUT	newtype	the requested MPI datatype (handle)

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
```

ticket-248T.

```
MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
    INTEGER, INTENT(IN) :: r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR
```

{static MPI::Datatype MPI::Datatype::Create_f90_integer(int r) *(binding deprecated, see Section 15.2)* }

This function returns a predefined MPI datatype that matches a INTEGER variable of KIND selected_int_kind(r). Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.

Restrictions on using the returned datatype with the "external32" data representation are given on page 570.

It is erroneous to supply a value for r that is not supported by the compiler.

Example:

```
integer       longtype, quadtype
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
...

call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x,  10, quadtype, ...)
```

Advice to users. The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction operations. There are two situations in which they behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_xxxx routines. ticket250-V.

If a variable was declared specifying a non-default KIND value that was not obtained with `selected_real_kind()` or `selected_int_kind()`, the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(*End of advice to users.*)

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_xxxx with the same combination of (xxxx,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_xxxx and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx,p,r). (*End of advice to implementors.*) ticket250-V.

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2 on page 498) or user-defined (Section 13.5.3 on page 499) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2 on page 498.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double" and "Double Extended" formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.

For MPI_TYPE_CREATE_F90_REAL:

```
if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
else if (p > 15) or (r >  307) then  external32_size = 16
else if (p >  6) or (r >   37) then  external32_size =  8
else                                 external32_size =  4
```

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.
For MPI_TYPE_CREATE_F90_INTEGER:

```
if      (r > 38) then  external32 representation is undefined
else if (r > 18) then  external32_size =  16
else if (r >  9) then  external32_size =  8
else if (r >  4) then  external32_size =  4
else if (r >  2) then  external32_size =  2
else                   external32_size =  1
```

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran and of the form MPI::<TYPE>n in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and **n** is the length in bytes of the machine representation. This datatype locally matches all variables of type (**typeclass**, **n**). The list of names for such types includes:

```
MPI_REAL4
```

```
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16
```

One datatype is required for each representation supported by the compiler. To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations `REAL*n`, `INTEGER*n`, always create a variable whose representation is of size **n**. These datatypes may also be used for variables declared with `KIND=INT8/16/32/64` or `KIND=REAL32/64/128`, which are defined in the `ISO_FORTRAN_ENV` intrinsic module. Note that the MPI datatypes and the `REAL*n`, `INTEGER*n` declarations count bytes whereas the Fortran `KIND` values count bits. All these datatypes are predefined. ticket250-V.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

MPI_SIZEOF(x, size)

IN	x	a Fortran variable of numeric intrinsic type (choice)
OUT	size	size of machine representation of that type (integer)

ticket-248T.

```
MPI_Sizeof(x, size, ierror) BIND(C)
    TYPE(*), DIMENSION(..)  ::  x
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR
```

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

> *Advice to users.* This function is similar to the C and C++ *sizeof* operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (*End of advice to users.*)

> *Rationale.* This function is not available in other languages because it would not be useful. (*End of rationale.*)

MPI_TYPE_MATCH_SIZE(typeclass, size, datatype) ticket252-W.

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	[ticket252-W.]datatype	datatype with correct type, size (handle)

ticket252-W.
```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
```

ticket-248T.
```
MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)
    INTEGER, INTENT(IN) ::  typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

ticket252-W. ticket252-W.
```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, 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.*)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype MPI_<TYPE>n can be received with this same datatype on another process. Most modern computers use 2's complement for integers and IEEE format for floating point. Thus, communication using these size-specific datatypes will not entail loss of precision or truncation errors.

Advice to users. Care is required when communicating in a heterogeneous environment. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
else if (myrank .eq. 1) then
    call MPI_RECV(x, xtype, 100, 0, ...)
endif
```

This may not work in a heterogeneous environment if the value of size is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second is to use `selected_real_kind` or `selected_int_kind` and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., `selected_real_kind(12)` on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communication. This may require explicit conversion to a variable of size that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)

if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',                &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,     &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
```

```
      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
      call MPI_FILE_CLOSE(fh, ierror)
   endif

   call MPI_BARRIER(MPI_COMM_WORLD, ierror)

   if (myrank .eq. 1) then
      call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,  &
                         MPI_INFO_NULL, fh, ierror)
      call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                             MPI_INFO_NULL, ierror)
      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
      call MPI_FILE_CLOSE(fh, ierror)
   endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

16.2.10 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. ticket230-B. While these may 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. ticket230-B. With Fortran 2008 and the new semantics defined in TR 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. ticket235-G. The rest of this section describes the potential problems in detail.

ticket230-B. The following MPI features are inconsistent with Fortran 90 and Fortran 77.

1. An MPI subroutine with a choice argument may be called with different argument types. ticket235-G. When using the `mpi_f08` module together with a compiler that supports Fortran 2008 + TR 29113, this problem is resolved.

2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. ticket234-F. This is only solved for choice buffers through the use of `DIMENSION(..)`.

3. ticket238-J. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. ticket238-J. This problem is solved with the use of the `ASYNCHRONOUS` attribute.

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. ticket238-J. This problem is resolved by relying on the extended semantics of the `ASYNCHRONOUS` attribute as specified in TR 29113.

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 77/90/95 without a language extension (for example, Cray pointers) that allows ticket245-Q. the allocated memory to be associated with a Fortran variable. Therefore, address ticket245-Q. sized integers were used in MPI-2.0 - MPI-2.2. In Fortran 2003, `TYPE(C_PTR)` enti- ticket245-Q. ties were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The `mpi_f08` module only supports `TYPE(C_PTR)` pointers.

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. ticket230-B.

Sections 16.2.11 through 16.2.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 16.2.7 on page 563. ticket230-B.

16.2.11 Problems Due to Strong Typing ticket230-B.

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, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 16.2.6 on page 559). In C, the use of `void*` for- ticket247-S. mal arguments avoids these problems. Similar to C, with Fortran 2008 + TR 29113 (and ticket235-G. 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. ticket235-G.

Using INCLUDE mpif.h, the following code fragment is technically 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 assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer 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     ! or  USE mpi
INTEGER size
CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

16.2.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send( a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:

- If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:

  Choice buffer arguments are declared as TYPE(*), DIMENSION(..). For example, consider 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), and s(11) are sent between the start of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to MPI_ISEND that allows MPI to operate directly on `s(1)`, `s(6)`, `s(11)`, ..., `s(96)`. The called MPI_ISEND routine will take only the first three of these elements due to the type signature "`3, MPI_REAL`".

All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT, MPI_FILE_WRITE_ALL_BEGIN) 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)` are guaranteed to be defined with the received data when MPI_WAIT returns.

> *Advice to implementors.* The Fortran descriptor for `TYPE(*), DIMENSION(..)` arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and released this buffer not before the nonblocking commincation has completed (e.g., in an MPI wait routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

> *Rationale.* If MPI_SUBARRAYS_SUPPORTED equals `.TRUE.`, non-contiguous buffers are handled inside of the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library 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. (*End of rationale.*)

- If MPI_SUBARRAYS_SUPPORTED equals `.FALSE.`:

  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, array data is not necessarily stored contiguously. For example, the array ticket236-H. section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, ... The same is ticket236-H. 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.[1] ticket236-H.

  Because MPI dummy buffer arguments are assumed-size arrays if ticket236-H. MPI_SUBARRAYS_SUPPORTED equals `.FALSE.`, this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI

[1] Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the `CONTIGUOUS` attribute.

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 'simply contiguous' section such as `A(1:N)` of such an array. ('Simply contiguous' is defined 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.

According to the Fortran 2008 Standard, Section 6.5.4, a 'simply contiguous' 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. The compiler can detect from analyzing the source code that the array is contiguous. Examples are

```
A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
```

Because of Fortran's column-major ordering, where the first index varies fastest, a 'simply contiguous' section of a contiguous array will also be contiguous.

The same problem can occur with a scalar argument. A compiler may make a copy of scalar dummy arguments within a called procedure when passed as an actual argument to a choice buffer routine. That this can cause a problem is illustrated by the example

```
[ticket236-H.]real :: a
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 a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.
>
> If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, simply contiguous array sections of such arrays, or scalars, and if no compiler option exists to inhibit such copying, 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.

16.2.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts

Fortran arrays with **vector** subscripts describe subarrays containing a possibly irregular set of elements

```
REAL a(100)
CALL MPI_Send( A((/7,9,23,81,82/)), 5, MPI_REAL, ...)
```

Arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

16.2.14 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, 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, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran `INTENT`: 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 dummy arguments that may be modified and allow one of these special constants as input, an `INTENT` is not specified.

16.2.15 Fortran Derived Types

ticket230-B.

ticket237-I. ticket237-I. MPI supports passing Fortran entities of `BIND(C)` and `SEQUENCE` derived types to choice dummy arguments, provided no type component has the `ALLOCATABLE` or `POINTER` attribute.

ticket237-I. ticket237-I. ticket237-I. ticket237-I. ticket237-I. The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
    type[ticket237-I.], BIND(C) :: mytype
       integer i
       real x
       double precision d
    end type mytype

    type(mytype) [ticket250-V.]:: foo[ticket237-I.], fooarr(5)
    integer [ticket250-V.]:: blocklen(3), type(3)
    integer(MPI_ADDRESS_KIND) [ticket250-V.]:: disp(3), base[ticket237-I.], lb, extent

    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)

[ticket237-I.]
[ticket237-I.]
    call MPI_SEND(foo%i, 1, newtype, ...)
[ticket237-I.]! or
[ticket237-I.]    call MPI_SEND(foo, 1, newtype, ...)
[ticket237-I.]    ! expects that base == address(foo%i) == address(foo)

[ticket237-I.]    call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
[ticket237-I.]    call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
[ticket237-I.]    extent = disp(2) - disp(1)
```

```
[ticket237-I.]    lb = 0
[ticket237-I.]    call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
[ticket237-I.]    call MPI_TYPE_COMMIT(newarrtype, ierr)
[ticket237-I.]
[ticket237-I.]    call MPI_SEND(fooarr, 5, newarrtype, ...)
```

ticket247-S.

Using the derived type variable `foo` instead of its first basic type element `foo%i` may be impossible if the MPI library implements choice buffer arguments through overloading instead of using `TYPE(*), DIMENSION(..)`, or through a non-standardized extensions such as `!$PRAGMA IGNORE_TKR`; see Section 16.2.6 on page 559.

TODO: The following text about the extent of derived types should be strongly checked by specialist on derived types!!! The correct variant should be chosen.

To use a derived type in an array requires a correct extent of the datatype handle to take care of the alignment rules applied by the compiler. These alignment rules may imply that there are gaps between the elements of a derived type, and also between the array elements. The alignment rules in Section 4.1 on page 85 and Section 4.1.6 on page 106 apply only to

VARIANT 1: `SEQUENCE`

VARIANT 2: `BIND(C)`

derived types. The extent of an iteroperable derived type (i.e., defined with `BIND(C)`) and a `SEQUENCE` derived type with the same content may be different because C and Fortran may apply different alignment rules.

VARIANT 1:

Using the `SEQUENCE` attribute instead of `BIND(C)` in the declararion on `mytype`, one can directly use `newtype` to send the `fooarr` array.

VARIANT 2:

In the example, one can directly use `newtype` to send the `fooarr` array. The resized `newarrtype` datatype is only needed, if `mytype` is a `SEQUENCE` derived type.

Using the extended semantics defined in TR 29113, it is also possible to use entities or derived types without either the `BIND(C)` or the `SEQUENCE` attribute as choice buffer arguments; some additional constraints must be observed e.g., no `ALLOCATABLE` or `POINTER` type components may exist. In this case, the `base` address in the example must be changed to become the address of `foo` instead of `foo%i`, because the Fortran compiler may rearrange type components or add padding as it may fit for such types. Sending the structure `foo` should then also be performed by providing it (and not `foo%i`) as actual argument for `MPI_Send`. ticket230-B.

ticket238-J.

16.2.16 Optimization Problems, an Overview

ticket230-B.

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. These problems are independent of the Fortran support method; i.e., they occur with the `mpi_f08` module, the `mpi` module, and the `mpif.h` include file. ticket238-J. ticket238-J. ticket238-J.

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

- Use of nonblocking routines or persistent requests *(Nonbl.)*.
- Use of one-sided routines *(1-sided)*.
- Use 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 two variables in such datatypes *(Bottom)*.

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 16.2.17 on page 582.
- Temporary data movement and temporary memory modifications; see Section 16.2.18 on page 589.
- Permanent data movement (e.g., through garbage collection); see Section 16.2.19 on page 590.

Table 16.4 shows in which usage areas the optimization problems may only occur.

Optimization ...	... may cause a problem in following usage areas			
	Nonbl.	1-sided	Split	Bottom
Code movement and register optimization	yes	yes	no	yes
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

Table 16.4: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" to "VOLATILE" on pages 585-585,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 16.2.7 on page 563.

ticket238-J.
16.2.17 Problems with Code Movement and Register Optimization

ticket238-J.
Nonblocking operations

ticket238-J. If a variable is local to a Fortran subroutine (i.e., not in a module or a `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 Fortran 90 register optimization – extreme.

```
Source                      compiled as                 or compiled as
[ticket238-J.]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
```

Example 16.11 shows extreme, but allowed, possibilities. 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. The compiler has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column. ticket238-J.

[ticket238-J.]

Example 16.12 Similar example with MPI_ISEND

```
Source                      compiled as                 with a possible MPI-internal
                                                        execution sequence
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= buf
                            buf = val_overwrite         buf = val_overwrite
call MPI_WAIT(req,..)       call MPI_WAIT(req,..)       send(*addr) ! within MPI_WAIT
buf = val_overwrite
```

ticket238-J.

Due to valid compiler code movement optimizations in Example 16.12, the content of buf may already be overwritten by the compiler when the content of buf is sent. 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).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization / code movement problem for nonblocking operations 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 an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication or parallel file I/O operation. ticket238-J.

One-sided communication

An example with instruction reordering due to register optimization can be found in Section 11.7.3 on page 431. ticket238-J.

MPI_BOTTOM and combining independent variables in datatypes

ticket238-J. This section is only relevant if the MPI program uses a buffer argument to an ticket238-J. MPI_SEND, MPI_RECV etc., 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 ticket238-J. one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 16.13 shows what Fortran compilers are allowed to do.

Example 16.13 Fortran 90 register optimization.

```
This source ...                          can be compiled as:

call MPI_GET_ADDRESS(buf,bufaddr,        call MPI_GET_ADDRESS(buf,...)
               ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,         call MPI_TYPE_CREATE_STRUCT(...)
               bufaddr,
               MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)        call MPI_TYPE_COMMIT(...)
val_old = buf                            register = buf
                                         val_old = register
call MPI_RECV(MPI_BOTTOM,1,type,...)     call MPI_RECV(MPI_BOTTOM,...)
val_new = buf                            val_new = register
```

ticket238-J.

ticket238-J. In Example 16.13, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

[ticket238-J.]

Example 16.14 Similar example with MPI_SEND

```
This source ...                          can be compiled as:

! buf contains val_old                   ! buf contains val_old
buf = val_new
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_new  is moved to here
                                         ! and detected as dead code
                                         ! and therefore removed
                                         !
buf = val_overwrite                      buf = val_overwrite
```

ticket238-J.

In Example 16.14, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.

Solutions

ticket238-J. ticket238-J.

The following sections show in detail how the problems with code movement and register optimization can be solved in a portable way. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations with the send and receive buffers used in nonblocking operations, or in operations in which MPI_BOTTOM is used, or datatype handles that combine several variables are used:

- Use of the Fortran `ASYNCHRONOUS` attribute.
- Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy routine.
- Declare the buffer as a Fortran module variable or within a Fortran common block.
- Use of the Fortran `VOLATILE` attribute.

Each of these methods solves the problems of code movement and register optimization, but may involve different degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 compliant (and later) MPI library and their compiler according to the requirements listed in Section 16.2.7 on page 563. The methods may have different impact on performance. MPI_F_SYNC_REG may have low impact, module data and the `ASYNCHRONOUS` attribute low through medium, and the `VOLATILE` attribute may have the most negative impact on performance. Note that there is one attribute that cannot be used for this purpose: the Fortran `TARGET` attribute does not solve code movement problems in MPI applications.

The Fortran ASYNCHRONOUS attribute

ticket238-J. ticket238-J.

Declaring an actual buffer argument with the `ASYNCHRONOUS` Fortran attribute in a scoping unit (or `BLOCK`) tells the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TR 29113 extension). Without the extensions specified in TR 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The `ASYNCHRONOUS` attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with `ASYNCHRONOUS` (which is mandatory for the `mpi_f08` module, with allowable exceptions listed in Section 16.2.6 on page 559), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNCHRONOUS_PROTECTS_NONBL is set to `.TRUE.` if both the protection of the actual buffer argument through `ASYNCHRONOUS` according to the TR 29113 extension and the declaration of the dummy argument with `ASYNCHRONOUS` in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to `.FALSE.`.

The `ASYNCHRONOUS` attribute has some restrictions. The TR 29113 defines (in the PDTR N1869):

> "Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent. Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 16.15 Case (a) on page 608, the read accesses to `b` within `function(b(i-1), b(i), b(i+1))` cannot be moved by compiler optimizations to before the wait call because `b` was declared as `ASYNCHRONOUS`. Note that only the elements 0, 1, 100, and 101 of `b` are involved in asynchronous communication but by definition, the total variable `b` is the pending communication affector and is usable for input and output asynchronous communication between the `MPI_I...` routines and `MPI_Waitall`. Case (a) works fine because the read accesses to `b` occur after the communication completed.

In Case (b), the read accesses to `b(1:100)` in the loop `i=2,99` are read accesses to a pending communication affector while input communication (i.e., the two `MPI_Irecv` calls) is pending. This is a contradiction to the rule that *for input communication, a pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjunct subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 16.19 on page 610.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the `ASYNCHRONOUS` attribute.

The problems with MPI_BOTTOM, as shown in Example 16.13 and Example 16.14, can also be solved by declaring the buffer `buf` with the `ASYNCHRONOUS` attribute.

Calling MPI_F_SYNC_REG

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 the MPI_F_SYNC_REG routine for this purpose; see Section 16.2.8 on page 565.

- The problems illustrated by the Examples 16.11 and 16.12 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.

Example 16.11 can be solved with	Example 16.12 can be solved with

```
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 to 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 below MPI_WAIT and before `buf=val_overwrite`.

- The problems illustrated by the Examples 16.13 and 16.14 can be solved with two additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/ MPI_SEND, and one directly after this communication operation.

  Example 16.13 can be solved with

  ```
  call MPI_F_SYNC_REG(buf)
  call MPI_RECV(MPI_BOTTOM,...)
  call MPI_F_SYNC_REG(buf)
  ```

  Example 16.14 can be solved with

  ```
  call MPI_F_SYNC_REG(buf)
  call MPI_SEND(MPI_BOTTOM,...)
  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; 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 11.7.3 on page 431, 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. That is, 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 16.16 on page 609, can **not** be solved with this method.

ticket238-J.
ticket238-J.

A user defined routine instead of MPI_F_SYNC_REG

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

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

Note that if the intent is declared in an explicit interface for 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, a call to MPI_RECV with MPI_BOTTOM as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other posibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented according to the rules specified in Section 16.2.7 on page 563.

Module variables and COMMON blocks

An alternative to the already mentioned methods 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 may alter the buffer or variable, provided that the compiler cannot infer 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 does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.

The (poorly performing) Fortran VOLATILE attribute

The VOLATILE attribute gives the buffer or variable the properties needed, but it may inhibit optimization of any code containing references or definitions of the buffer or variable.

ticket238-J.
ticket238-J.

The Fortran TARGET attribute

The `TARGET` attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the `TARGET` attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

> *Rationale.* The Fortran standardization body decided to extend the `ASYNCHRONOUS` attribute within the TR 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the `TARGET` attribute. (*End of rationale.*)

16.2.18 Temporary Data Movement and Temporary Memory Modification

ticket238-J.
ticket238-J.

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 16.15, Case (b) on page 608. Example 16.16 on page 609 shows a possibility that could be problematic.

ticket238-J.

In the compiler-generated, possible optimization in Example 16.17, buf(100,100) from Example 16.16 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 16.18 shows a second possible optimization. The whole array is temporarily moved to local_buf. When storing local_buf back to the original location buf, then this includes also an overwriting of the receive buffer part buf(1,1:100), i.e., this storing back may overwrite the asynchronously received data.

Note, that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11 on page 393.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations with between the ..._BEGIN and ..._END calls; see Section 13.4.5 on page 486.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 585 in Section 16.2.17 on page 582, the `ASYNCHRONOUS` attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the numerical read accesses are separated into different variables, as shown in Example 16.19 on page 610 and in Example 16.20 on page 611.

Note also that the methods

- calling MPI_F_SYNC_REG (or such a user-defined routine),
- using module variables and `COMMON` blocks, and

- the `TARGET` attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the numerical code shown in Example 16.16 and 16.17.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring `buf` as `VOLATILE` because the `VOLATILE` implies that all accesses to any storage unit (word) of `buf` must be directly done in the main memory exactly in the sequence defined by the application program. The `VOLATILE` attribute prevents all register and cache optimizations. Therefore, `VOLATILE` may cause a huge performance degradation.

Instead of solving the problem, it is needed to **prevent** the problem. When overlapping communication and computation, the nonblocking communication (or nonblocking or split collective IO) and the computation should be executed **on different sets of variables**. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

> *Rationale.* This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature is available in a later version of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 16.19 on page 610 (which is a solution for the problem shown in Example 16.15 on page 608) and in Example 16.20 on page 611 (which is a solution for the problem shown in Example 16.18 on page 609), the array is split into inner and halo part and both disjunct parts are passed to a subroutine `separated_sections`. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TR 29113 based MPI library.

ticket238-J.

16.2.19 Permanent Data Movement

ticket238-J.

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. Automatic garbage collection implementation is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.
- Nonblocking MPI operations (communication, one-sided, I/O) if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the `ASYNCHRONOUS` attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 16.2.7 on page 563.

ticket238-J.

16.2.20 Comparison with C

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. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while the nonblocking operations are pending. ticket238-J. ticket238-J.

16.3 Language Interoperability

16.3.1 Introduction

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

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

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

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

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

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

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

16.3.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C/C++ character strings may not be compatible with Fortran `CHARACTER` variables. However, we assume that a Fortran `INTEGER`, as well as a (sequence associated) Fortran array of `INTEGER`s, can be passed to a C or C++ program. We also assume that Fortran, C, and

C++ have address-sized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that `INTEGER(KIND=MPI_OFFSET_KIND)` can be passed from Fortran to C as MPI_Offset.

16.3.3 Initialization

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

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

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

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

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

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

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

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

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.

ticket231-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 `BIND(C)` derived type that contains an `INTEGER` field named MPI_VAL. This `INTEGER` value 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_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
```

Example 16.21 The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
INTEGER [ticket250-V.]:: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END
```

16.3.5 Status

ticket243-O. 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 ticket243-O. f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in ticket243-O. the `mpi` module or `mpif.h`. 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.

ticket250-V. ticket243-O. *Advice to users.* There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Fig. 16.1 on page 597. (*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.*)

ticket243-O. Using the `mpi_f08` Fortran module, a status is declared as `TYPE(MPI_Status)`. The C type `MPI_F08_status` can be used to pass a Fortran `TYPE(MPI_Status)` argument into a C routine. Figure 16.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

```
int MPI_Status_f082c(MPI_F08_status *f08_status, MPI_Status *c_status)
```

This C routine converts a Fortran `mpi_f08` `TYPE(MPI_Status)` into a C `MPI_Status`.

```
int MPI_Status_c2f08(MPI_Status *c_status, MPI_F08_status *f08_status)
```

```
! FORTRAN CODE
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: 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)

/* C code */

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

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

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

   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   /* the message sent contains an int count of 5, followed  */
   /* by the 5 REAL entries of the Fortran array R.          */
}
```

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

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

Callback Functions

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

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

ticket230-B.

Advice to users. Callbacks themselves, including the predefined Fortran functions (e.g., MPI_COMM_NULL_COPY_FN) should not be passed from one application routine written in one language or Fortran support method to another application routine written in another language or Fortran support method, which passes this callback routine to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice to users on page 274. (*End of advice to users.*)

Error Handlers

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

Reduce Operations

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

Addresses

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

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

```
char name [MPI_MAX_OBJECT_NAME];
```

(*End of advice to users.*)

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

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

16.3.10 Interlanguage Communication

ticket250-V. The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 16.28 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
USE mpi_f08
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: IERR, MYRANK, AOBLEN(1), AOTYPE(1)
[ticket250-V.]TYPE(MPI_Type) :: TYPE
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: 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 MPI_TYPE_COMMIT(TYPE, IERR)

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

```
ELSE
   CALL C_ROUTINE(TYPE[ticket250-V.]%MPI_VAL)
END IF


/* C code */

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

   type = MPI_Type_f2c(*fhandle);

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

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

[ticket238-J.]

Example 16.15 Protecting nonblocking communication with the `ASYNCHRONOUS` attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)            ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b(  0), ..., left,  ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b(  1), ..., left,  ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)

#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
  CALL MPI_Waitall(4,req,...)
  DO i=1,100 ! compute all new local data
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
#endif

#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
  DO i=2,99  ! compute  only elements for which halo data is not needed
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
  CALL MPI_Waitall(4,req,...)
  i=1 ! compute leftmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
  i=100 ! compute rightmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
```

[ticket238-J.]

Example 16.16 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,...)
```

[ticket238-J.]

Example 16.17 The compiler may substitute the nested loops through loop fusion.

```
REAL :: buf(100,100),  buf_1dim(10000)
EQUIVALENCE (buf(1,1), buf_1dim(1))
CALL MPI_Irecv(buf(1,1:100),...req,...)
tmp(1:100) = buf(1,1:100)
DO j=1,10000
  buf_1dim(h)=...
END DO
buf(1,1:100) = tmp(1:100)
CALL MPI_Wait(req,...)
```

[ticket238-J.]

Example 16.18 Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

```
REAL :: buf(100,100), local_buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
local_buf = buf
DO j=1,100
  DO i=2,100
    local_buf(i,j)=....
  END DO
END DO
buf = local_buf ! may overwrite asynchronously received
                ! data in buf(1,1:100)
CALL MPI_Wait(req,...)
```

[ticket238-J.]

Example 16.19 Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: b(0:101)     ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)  ! elements 1 and 100 are newly computed
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
END

SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
USE mpi_f08
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
REAL :: bnew(0:101)  ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b_lefthalo (  0), ..., left,  ..., req(1), ...)
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
! b_lefthalo and b_righthalo is written asynchronously.
! There is no other concurrent access to b_lefthalo and b_righthalo.
CALL MPI_Isend(b_inner(  1),     ..., left,  ..., req(3), ...)
CALL MPI_Isend(b_inner(100),     ..., right, ..., req(4), ...)

DO i=2,99  ! compute  only elements for which halo data is not needed
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
  ! b_inner is read and send at the same time.
  ! This is allowed based on the rules for ASYNCHRONOUS.
END DO
CALL MPI_Waitall(4,req,...)
END SUBROUTINE
```

[ticket238-J.]

Example 16.20 Protecting GPU optimizations with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: buf(100,100)
CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
END

SUBROUTINE separated_sections(buf_halo, buf_inner)
REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
REAL :: buf_inner(2:100,1:100)
REAL :: local_buf(2:100,100)

CALL MPI_Irecv(buf_halo(1,1:100),...req,...)
local_buf = buf_inner
DO j=1,100
  DO i=2,100
    local_buf(i,j)=....
  END DO
END DO
buf_inner = local_buf ! buf_halo is not touched!!!

CALL MPI_Wait(req,...)
```

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

[ticket247-S.]Fortran Support Method Specific Constants

Fortran type: `LOGICAL`
[ticket234-F.]MPI_SUBARRAYS_SUPPORTED (Fortran only)
[ticket238-J.]MPI_ASYNCHRONOUS_PROTECTS_NONBL (Fortran only)

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` [ticket231-C.]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

Named Predefined Datatypes		C/C++ types
C type: MPI_Datatype Fortran type: INTEGER [ticket231-C.]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 [ticket231-C.]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 [ticket231-C.]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*[ticket231-C.]2
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 [ticket231-C.]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 [ticket231-C.]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 [ticket231-C.]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 [ticket231-C.]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` [ticket231-C.]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 [ticket231-C.] or TYPE(MPI_Group)	MPI::GROUP_NULL const MPI::Group
MPI_COMM_NULL MPI_Comm / INTEGER [ticket231-C.] or TYPE(MPI_Comm)	MPI::COMM_NULL [1])
MPI_DATATYPE_NULL MPI_Datatype / INTEGER [ticket231-C.] or TYPE(MPI_Datatype)	MPI::DATATYPE_NULL const MPI::Datatype
MPI_REQUEST_NULL MPI_Request / INTEGER [ticket231-C.] or TYPE(MPI_Request)	MPI::REQUEST_NULL const MPI::Request
MPI_OP_NULL MPI_Op / INTEGER [ticket231-C.] or TYPE(MPI_Op)	MPI::OP_NULL const MPI::Op
MPI_ERRHANDLER_NULL MPI_Errhandler / INTEGER [ticket231-C.] or TYPE(MPI_Errhandler)	MPI::ERRHANDLER_NULL const MPI::Errhandler
MPI_FILE_NULL MPI_File / INTEGER [ticket231-C.] or TYPE(MPI_File)	MPI::FILE_NULL
MPI_INFO_NULL MPI_Info / INTEGER [ticket231-C.] or TYPE(MPI_Info)	MPI::INFO_NULL const MPI::Info
MPI_WIN_NULL MPI_Win / INTEGER [ticket231-C.] or TYPE(MPI_Win)	MPI::WIN_NULL

[1]) C++ type: See Section 16.1.7 on page 544 regarding class hierarchy and the specific type of MPI::COMM_NULL

Empty group

C type: MPI_Group Fortran type: INTEGER [ticket231-C.]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 [ticket230-B.]with mpi module [ticket230-B.]/ Fortran type with mpi_f08 module	C++ name C++ type
MPI_COMM_NULL_COPY_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Comm_copy_attr_function) [2])	MPI_COMM_NULL_COPY_FN same as in C [1])
MPI_COMM_DUP_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Comm_copy_attr_function) [2])	MPI_COMM_DUP_FN same as in C [1])
MPI_COMM_NULL_DELETE_FN MPI_Comm_delete_attr_function / COMM_DELETE_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Comm_delete_attr_function) [2])	MPI_COMM_NULL_DELETE_FN same as in C [1])
MPI_WIN_NULL_COPY_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Win_copy_attr_function) [2])	MPI_WIN_NULL_COPY_FN same as in C [1])
MPI_WIN_DUP_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Win_copy_attr_function) [2])	MPI_WIN_DUP_FN same as in C [1])
MPI_WIN_NULL_DELETE_FN MPI_Win_delete_attr_function / WIN_DELETE_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Win_delete_attr_function) [2])	MPI_WIN_NULL_DELETE_FN same as in C [1])
MPI_TYPE_NULL_COPY_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Type_copy_attr_function) [2])	MPI_TYPE_NULL_COPY_FN same as in C [1])
MPI_TYPE_DUP_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Type_copy_attr_function) [2])	MPI_TYPE_DUP_FN same as in C [1])
MPI_TYPE_NULL_DELETE_FN MPI_Type_delete_attr_function / TYPE_DELETE_ATTR_[ticket250-V.]FUNCTION / [ticket230-B.]PROCEDURE(MPI_Type_delete_attr_function) [2])	MPI_TYPE_NULL_DELETE_FN same as in C [1])

[1] See the advice to implementors [ticket230-B.](on page 273) and advice to users (on page 273) on [ticket230-B.]the predefined C functions MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2 on page 270

[ticket230-B.][2] See the advice to implementors (on page 273) and advice to users (on page 274)
[ticket230-B.] on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, ... in
[ticket230-B.] Section 6.7.2 on page 270

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	C++ name C++ type
MPI_ARGVS_NULL char*** / 2-dim. array of CHARACTER*(*)	MPI::ARGVS_NULL const char ***
MPI_ARGV_NULL char** / array of CHARACTER*(*)	MPI::ARGV_NULL const char **
MPI_ERRCODES_IGNORE int* / INTEGER array	Not defined for C++
MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*) [ticket231-C.] or TYPE(MPI_Status), DIMENSION(*)	Not defined for C++
MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) [ticket231-C.] or TYPE(MPI_Status)	Not defined for C++
MPI_UNWEIGHTED int* / INTEGER array	Not defined for C++

C Constants Specifying Ignored Input (no C++ or Fortran)

C type: MPI_Fint*	[ticket243-O.]equivalent to Fortran
MPI_F_STATUSES_IGNORE	[ticket243-O.]MPI_STATUSES_IGNORE in mpi / mpif.h
MPI_F_STATUS_IGNORE	[ticket243-O.]MPI_STATUS_IGNORE in mpi / mpif.h
[ticket243-O.]C type: MPI_F08_status*	[ticket243-O.]equivalent to Fortran
[ticket243-O.]MPI_F08_STATUSES_IGNORE	[ticket243-O.]MPI_STATUSES_IGNORE in mpi_f08
[ticket243-O.]MPI_F08_STATUS_IGNORE	[ticket243-O.]MPI_STATUS_IGNORE in mpi_f08

C and C++ preprocessor Constants and Fortran Parameters
C/C++ type: `const int` (or unnamed `enum`)
Fortran type: `INTEGER`
MPI_SUBVERSION
MPI_VERSION

A.1.2 Types

The following are defined C type definitions, included in the file `mpi.h`.

```
/* C opaque types */
MPI_Aint
MPI_Fint
MPI_Offset
MPI_Status
MPI_F08_status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Op
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)
MPI::Aint
MPI::Offset
MPI::Status

// C++ handles to assorted structures (classes,
// all within the MPI namespace)
MPI::Comm
MPI::Intracomm
MPI::Graphcomm
MPI::Distgraphcomm
MPI::Cartcomm
MPI::Intercomm
MPI::Datatype
MPI::Errhandler
MPI::Exception
MPI::File
MPI::Group
MPI::Info
MPI::Op
MPI::Request
```

ticket243-O.

MPI::Prequest
MPI::Grequest
MPI::Win

ticket243-O.

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)
```

ticket231-C.

A.1.3 Prototype Definitions

ticket0.
ticket230-B.

C Bindings

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 [ticket252-W.]datatype,
              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);
```

ticket230-B.

Fortran 2008 Bindings with the mpi_f08 Module

ticket230-B.

With the Fortran `mpi_f08` module, the callback prototypes are:

ticket-248T.

The user-function argument to MPI_Op_create should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)
      USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE ::  invec, inoutvec
      INTEGER ::  len
      TYPE(MPI_Datatype) ::  datatype
```

ticket230-B.

The copy and delete function arguments to MPI_Comm_create_keyval should be declared according to:

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Comm) ::  oldcomm
      INTEGER ::  comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
  attribute_val, extra_state, ierror) BIND(C)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

ticket230-B.

The copy and delete function arguments to MPI_Win_create_keyval should be declared according to:

ticket-248T.

```
ABSTRACT INTERFACE
```

```
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Win) ::  oldwin
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
  extra_state, ierror) BIND(C)
      TYPE(MPI_Win) ::  win
      INTEGER ::  win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

ticket230-B.

The copy and delete function arguments to MPI_Type_create_keyval should be declared according to: ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
      TYPE(MPI_Datatype) ::  oldtype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
  attribute_val, extra_state, ierror) BIND(C)
      TYPE(MPI_Datatype) ::  datatype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

ticket230-B.

The handler-function argument to MPI_Comm_create_errhandler should be declared like this: ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  error_code
```

ticket230-B.

The handler-function argument to MPI_Win_create_errhandler should be declared like this: ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
      TYPE(MPI_Win) ::  win
      INTEGER ::  error_code
```

ticket230-B.

The handler-function argument to MPI_File_create_errhandler should be declared like this: ticket-248T.

```
ABSTRACT INTERFACE
```

```
  SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)
      TYPE(MPI_File) :: file
      INTEGER :: error_code
```

ticket230-B.

The query, free, and cancel function arguments to MPI_Grequest_start should be declared according to:

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
  BIND(C)
      TYPE(MPI_Status) :: status
      INTEGER :: ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
      INTEGER :: ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
  BIND(C)
      INTEGER :: ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
      LOGICAL :: complete
```

ticket230-B.

The extend and conversion function arguments to MPI_Register_datarep should be declared according to:

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror) BIND(C)
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
```

ticket-248T.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
  filebuf, position, extra_state, ierror) BIND(C)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE :: userbuf, filebuf
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: count, ierror
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

ticket230-B.

Fortran Bindings with mpif.h or the mpi Module

ticket230-B.

With the Fortran `mpi` module or `mpif.h`, 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, [ticket252-W.]DATATYPE)
   <type> INVEC(LEN), INOUTVEC(LEN)
```

```
   INTEGER LEN, [ticket252-W.]DATATYPE
```

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

```
SUBROUTINE COMM_COPY_ATTR_[ticket250-V.]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_[ticket250-V.]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_[ticket250-V.]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_[ticket250-V.]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_[ticket250-V.]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_[ticket250-V.]FUNCTION([ticket252-W.]DATATYPE, TYPE_KEYVAL, ATTI
              EXTRA_STATE, IERROR)
   INTEGER [ticket252-W.]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
```

ticket230-B.

Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. ticket0.

B.1 Changes from Version 2.2 to Version 3.0

ticket230-B. ticket247-S. ticket248-T. ticket231-C.

1. Section 2.3 on page 10, and Sections 16.2.1, 16.2.2, 16.2.7 on pages 550, 551, and 563.
   The new `mpi_08` Fortran module is introduced.

2. Section 2.5.1 on page 12, Section 16.2.3 on page 554, Section 16.2.2 on page 551, and Section 16.2.7 on page 563.
   Handles to opaque objects are defined as named types within the `mpi_08` Fortran module. The handle types are also available through the `mpi` Fortran module. ticket234-F. ticket235-G. ticket236-H.

3. Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 16.2.1, 16.2.10, 16.2.11, 16.2.12, 16.2.13 on pages 550, 574, 575, 576, 579, and Sections 16.2.3, 16.2.2, 16.2.7 on pages 554, 551, 563.
   Within the `mpi_08` Fortran module, choice buffers are defined as assumed-type and assumed-rank according to Fortran 2008 TR 29113 [36], and the compile-time constant MPI_SUBARRAYS_SUPPORTED is set to `.TRUE.`. With this, Fortran subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TR 29113 feature, the constant is set to `.FALSE.`. ticket239-K.

4. Section 2.6.2 on page 18, Section 16.2.2 on page 551, and Section 16.2.7 on page 563.
   The ierror dummy arguments are `OPTIONAL` within the `mpi_08` Fortran module. ticket243-O.

5. Section 3.2.5 on page 34, Section 16.2.3 on page 554, Section 16.2.2 on page 551, Section 16.2.7 on page 563, and Section 16.3.5 on page 596.
   Within the `mpi_08` Fortran module, the status is defined as `TYPE(MPI_Status)`. New conversion routines are added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, ticket252-W.

6. Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 15.1, 16.2.9 on pages 111, 186, 192, 280, 286, 331, 333, 335, 527, and 565. In some routines, the dummy argument names were changed, because they were identical to the Fortran keywords `TYPE` and `FUNCTION`. The new dummy argument names must be used because the `mpi` and `mpi_08` modules guarantee keyword-based actual argument lists. The argument name type was changed into oldtype in MPI_TYPE_DUP, and into datatype in the Fortran `USER_FUNCTION` of MPI_OP_CREATE, and in MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, in the callback prototype definition MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed into user_fn in MPI_OP_CREATE, into comm_errhandler_fn in MPI_COMM_CREATE_ERRHANDLER, into win_errhandler_fn in MPI_WIN_CREATE_ERRHANDLER, into file_errhandler_fn in MPI_FILE_CREATE_ERRHANDLER, into handler_fn in MPI_ERRHANDLER_CREATE.
ticket251-V. For consistency reasons, INOUBUF was changed into INOUTBUF in MPI_REDUCE_LOCAL, and intracomm into newintracomm in MPI_INTERCOMM_MERGE.

ticket245-Q.

7. Section 8.2 on page 326.
In Fortran with the `mpi` and `mpi_f08` modules, MPI_ALLOC_MEM now also supports `TYPE(C_PTR)` C-pointer instead of only returning an address-sized integer that may be usable together a with non-standard Cray-pointer. The Fortran interfaces with `INTEGER(KIND=MPI_ADDRESS_KIND)` BASEPTR in the `mpi` module and the `mpif.h` include file are deprecated since MPI-3.0.

ticket237-I.

8. Section 16.2.15 on page 580, and Section 16.2.7 on page 563.
Fortran `SEQUENCE` and `BIND(C)` derived application types can be used as buffers in MPI operations.

ticket238-J.

9. Section 16.2.16 on page 581 to Section 16.2.19 on page 590, Section 16.2.7 on page 563, and Section 16.2.8 on page 565.
The sections about Fortran optimization problems and their solution is partially rewritten and new methods are added, e.g., the use of the `ASYNCHRONOUS` attribute. The constant MPI_ASYNCHRONOUS_PROTECTS_NONBL tells whether the meaning of the `ASYNCHRONOUS` attribute is extended to protect nonblocking operations. The Fortran routine MPI_F_SYNC_REG is added. To achieve a secure and portable programming interfaces, in Section 16.2.7, several requirements are defined for the combination of an MPI library and a Fortran compiler to be MPI-3.0 compliant.

ticket233-E.

10. Section 16.2.4 on page 556.
The use of the `mpif.h` Fortran include file is strongly discouraged.

ticket232-D.

11. Section 16.2.3 on page 554, and Section 16.2.7 on page 563.
The existing `mpi` Fortran module must implement compile-time argument checking.

ticket242-N.

12. Section 16.2.2 on page 551.
Within the `mpi_08` Fortran module, dummy arguments are declared with `INTENT=IN`, `OUT`, or `INOUT` as defined in the `mpi_08` interfaces.

ticket230-B.
ticket231-C.
ticket232-D.
ticket234-F.
ticket237-I.
ticket238-J.
ticket239-K.
ticket233-O.

13. Section 16.2.7 on page 563.
    This new section summarizes requirements that an MPI library together with a Fortran compiler is compliant to the MPI standard.

ticket230-B.

14. Section A.1.1, Table "*Predefined functions*" on page 622, Section A.1.3 on page 627, and Section A.3.4 on page 669.
    Within the new `mpi_f08` module, all callback prototype definitions are defined with explicit interfaces `PROCEDURE(MPI_...)` with `BIND(C)` attribute.

ticket250-V.

15. Section A.1.3 on page 627.
    In some routines, the Fortran callback prototype names were changed from ..._FN to ..._FUNCTION to be consistent with the other language bindings.

B.2 Changes from Version 2.1 to Version 2.2

1. Section 2.5.4 on page 15.
   It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.

2. Section 2.6 on page 17, Section 2.6.4 on page 19, and Section 16.1 on page 537.
   The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.

3. Section 3.2.2 on page 29.
   MPI_CHAR for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.

4. Section 3.2.2 on page 29.
   MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.

5. Section 3.4 on page 41, Section 3.7.2 on page 53, Section 3.9 on page 76, and Section 5.1 on page 143.
   The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.

6. Section 3.7 on page 51.
   The Advice to users for IBSEND and IRSEND was slightly changed.

7. Section 3.7.3 on page 57.
   The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.

8. Section 3.7.6 on page 70.
   MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.

[26] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from `http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90`. 16.2.3

[27] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12

[28] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. 5.12

[29] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007. 5.12

[30] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication Optimization for Medical Image Reconstruction Algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12

[31] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2

[32] International Organization for Standardization, Geneva, ISO 8859-1:1987. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987. 13.5.2

[33] International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. 12.4, 13.2.1

[34] International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. *Information technology – Programming languages – Fortran – Part 1: Base language*, November 2010. 16.2.1, 16.2.2

[35] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113, Draft N1869. *TR on further interoperability with C*, July, 18 2011. http://www.nag.co.uk/sc22wg5/ and ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1869.pdf. 16.2.2

[36] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113. *TR on further interoperability with C*, 2012. http://www.nag.co.uk/sc22wg5/. 16.2.1, 16.2.1, 16.2.2, 16.2.7, 3

[37] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. 4.1.4