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

April 21, 2011

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.

ticket232-D. A new Fortran `mpi_f08` module is introduced to provide extended compile-time argument checking and buffer handling in nonblocking routines. The existing `mpi` module provides compile-time argument checking on the basis of existing MPI-2.2 routine defini- ticket233-E. tions. 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 of the same function used with `USE mpi` or `INCLUDE 'mpif.h'`
- The Fortran version used with `USE mpi_f08`.
- The C++ binding (which is deprecated).

Fortran in this document refers to Fortran 90 and higher; see Section 2.6. 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 ticket231-C.
Fortran with USE mpi_f08, and in C and C++, a different handle type is defined for ticket231-C.
each category of objects. With Fortran USE mpi_f08, the handles are defined as Fortran ticket231-C.
sequenced derived types that consist of only one element INTEGER :: MPI_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and mpif.h. The type names are identical to the names in C, except that they are not case sensitive. For example:

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

In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

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

ticket231-C.

> *Rationale.* Due to the sequence attribute in the definition of handles in the mpi_f08 module, the new Fortran handles are associated with one numerical storage unit; i.e., they have the same C binding as the INTEGER handles of the mpi module. Since the 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. (*End of rationale.*)

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

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

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

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

> *Rationale.* This design hides the internal representation used for MPI data structures, thus allowing similar calls in C, C++, and Fortran. It also avoids conflicts with

the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

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

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

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

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

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE. With the `mpi_f08` module and the C++ bindings, optional arguments through function overloading are used instead of MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, and

ticket244-P.

MPI_UNWEIGHTED. This is done by having two bindings where one has the optional argument and one does not. The constants MPI_ARGV_NULL and MPI_ARGVS_NULL are not substituted by function overloading.

2.5.3 State

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

2.5.4 Named Constants

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

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

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in C/C++ `switch` or Fortran `select/case` statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C/C++ and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE). The handles themselves are constants and can be also used in initialization expressions or assignments.

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

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

and their C++ counterparts where appropriate.

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

```
MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE
MPI_ARGV_NULL
MPI_ARGVS_NULL
MPI_UNWEIGHTED
```

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

2.5.5 Choice

ticket234-F. ticket234-F. ticket234-F.

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file `mpif.h` or the `mpi` module, the document uses <type> to represent a choice variable; with the Fortran `mpi_f08` module, such arguments are declared with the new Fortran syntax TYPE(*), DIMENSION(..); for C and C++, we use void *.

Advice to implementors. The implementor can freely choose how to implement choice arguments in the `mpi` module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the `mpi_f08` module. See details in Section 16.2.1 on page 542. (*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

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

most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran ticket234-F. 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. 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 ticket250-V. (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., ticket239-K. 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 ticket239-K. error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Advice to implementors. With the `mpi_f08` module, an MPI library may implement the ierror argument through function overloading instead of using the `OPTIONAL` attribute. With function overloading, the branch is implemented at link-time; with the the `OPTIONAL` attribute, it is implemented at run-time. (*End of advice to implementors.*)

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

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

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

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

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

ticket-248T.

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

a 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 code of the received message.

ticket243-O. With Fortran `USE mpi_f08`, status is defined as the Fortran sequence 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.

> *Rationale.* The Fortran `TYPE(MPI_Status)` is defined as a sequence derived type to that it can be used at any location where the status integer array representation can be used, e.g., in user defined sequence derived types or 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)* } cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

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

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

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

ticket244-P.

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

ticket244-P.
ticket244-P.
ticket244-P.
ticket244-P.
ticket244-P.

Example 3.1 The `mpi_f08` bindings for MPI_PROBE are:

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

Example 3.2 The C++ bindings for MPI_PROBE are:

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

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

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

ticket-248T.

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

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

Start a nonblocking receive.

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

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

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

> *Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.2-16.2.12, especially in Sections 16.2.4 and 16.2.5 on pages 545-548 about *problems due to data copying and sequence association with subscript triplets* and *vector subscripts*, and in Sections 16.2.8 to 16.2.11 on pages 549 to 558 about *optimization problems*, *code movements and register optimization*, and *temporary* and *permanent data movements*. (*End of advice to users.*)

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

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

4.1.10 Duplicating a Datatype

ticket252-W. MPI_TYPE_DUP(oldtype, newtype)

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

ticket252-W.

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

ticket252-W.

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

ticket252-W. ticket252-W. ticket-248T.

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

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

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 6.7.4 on page 275. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype. ticket252-W. ticket252-W.

4.1.11 Use of General Datatypes in Communication

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

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

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

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

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

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

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

5.9.5 User-Defined Reduction Operations

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.

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

ticket252-W.
ticket252-W.

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

ticket-248T.

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

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);
```

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

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

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

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

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

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

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

6.7.2 Communicators

Functions for caching on communicators are:

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

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

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

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

ticket-248T.

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
             extra_state, ierror)
```

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

{static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function* comm_copy_attr_fn, MPI::Comm::Delete_attr_function* comm_delete_attr_fn, void* extra_state) *(binding deprecated, see Section 15.2)* }

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

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

The C callback functions are:

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

and

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

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

The Fortran callback functions are:

ticket250-V.

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

and

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)*}

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

IN	comm_old	input communicator (handle)
IN	indegree	size of sources and sourceweights arrays (non-negative integer)
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
IN	outdegree	size of destinations and destweights arrays (non-negative integer)
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
IN	info	hints on optimization and interpretation of weights (handle)
IN	reorder	the ranks may be reordered (true) or not (false) (logical)
OUT	comm_dist_graph	communicator with distributed graph topology (handle)

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

```
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
              OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
              COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

ticket-248T.

```
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
              outdegree, destinations, destweights, info, reorder,
              comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  indegree, sources(*), outdegree,
    destinations(*)
    INTEGER, INTENT(IN) ::  sourceweights(*) !  optional by overloading
    INTEGER, INTENT(IN) ::  destweights(*) !  optional by overloading
    TYPE(MPI_Info), INTENT(IN) ::  info
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int
              indegree, const int sources[], const int sourceweights[],
              int outdegree, const int destinations[],
              const int destweights[], const MPI::Info& info, bool reorder)
              const(binding deprecated, see Section 15.2) }
```

```
{MPI::Distgraphcomm
              MPI::Intracomm::Dist_graph_create_adjacent(int indegree,
              const int sources[], int outdegree, const int destinations[],
              const MPI::Info& info, bool reorder) const(binding deprecated, see
              Section 15.2) }
```

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about the edges to its neighbors in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication topology is the combination of all edges shown in the sources arrays of all processes in comm_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and that thus do not occur as source or destination rank in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. ticket244-P. In C or in the Fortran mpi module or ticket244-P. mpif.h include file, an application can supply the special value MPI_UNWEIGHTED for the ticket244-P. weight array to indicate that all edges have the same (effectively no) weight. In the Fortran mpi_f08 module or in C++, this constant does not exist and the weight arguments may be ticket244-P. omitted from the argument list. It is erroneous to supply MPI_UNWEIGHTED, or in mpi_f08 or C++ omit the weight arrays, for some but not all processes of comm_old. Note that MPI_UNWEIGHTED is not a special weight value; rather it is a special value for the total array argument. In C, one would expect it to be NULL. In Fortran, MPI_UNWEIGHTED is an object like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

The meaning of the info and reorder arguments is defined in the description of the following routine.

node	**exchange**	**shuffle**	**unshuffle**
	neighbors(1)	neighbors(2)	neighbors(3)
0 (000)	1	0	0
1 (001)	0	2	4
2 (010)	3	4	1
3 (011)	2	6	5
4 (100)	5	1	2
5 (101)	4	3	6
6 (110)	7	5	3
7 (111)	6	7	7

Suppose that the communicator `comm` has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology.

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

IN	comm	communicator with distributed graph topology (handle)
OUT	indegree	number of edges into this process (non-negative integer)
OUT	outdegree	number of edges out of this process (non-negative integer)
OUT	weighted	false if MPI_UNWEIGHTED was supplied or the weights were omitted during creation, true otherwise (logical)

ticket244-P.

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

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
```

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

{void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) *(binding deprecated, see Section 15.2)* }

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied (in C or the `mpi` module) or the argument is omitted (in `mpi_f08` or C++) for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied or the weights were omitted during the construction of the graph then no weight information is returned in that array or those arrays. The only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBOR_COUNT, then only the first part of the full list is returned. Note, that the order of returned edges does need not to be identical to the order that was provided in the creation of comm for the case that MPI_DIST_GRAPH_CREATE_ADJACENT was used.

ticket244-P.

ticket244-P.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

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)
```

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

ticket-248T.

```
MPI_Get_version(version, subversion, ierror)
```

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 16.2.8.

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 16.2.2-16.2.12, especially in Sections 16.2.4 and 16.2.5 on pages 545-548 about *problems due to data copying and sequence* ticket236-H. *association with subscript triplets* and *vector subscripts*, and in Sections 16.2.8 to 16.2.11 on ticket238-J. pages 549 to 558 about *optimization problems*, *code movements and register optimization*, and *temporary* and *permanent data movements.* Sections 16.2.9 to 16.2.9 on pages 553-556 discuss several solutions for the problem in this example.

starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined state for the request. ticket0.

In C, the query function is

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

in Fortran

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

and in C++

{`typedef int MPI::Grequest::Query_function(void* extra_state,`
`MPI::Status& status);` *(binding deprecated, see Section 15.2)*}

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 or has omitted the status argument (with the ticket244-P. mpi_f08 Fortran module or C++), then MPI will pass a valid status object to query_fn, and this status will be ignored upon return of the callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE is called on the request; it may be invoked several times for the same generalized request, e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

and in Fortran

```
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

{`typedef int MPI::Grequest::Free_function(void* extra_state);` *(binding deprecated, see Section 15.2)*}

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

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/ MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid many (though not all) of the problems described in "A Problem with Code Movements and Register Optimization," Section 16.2.9 on page 550. 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 the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END)

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 [35].

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

MPI defines three methods of Fortran support:

1. **INCULDE 'mpif.h'** This method is described in Section 16.2.13. The use of the include file `mpif.h` is strongly discouraged starting with MPI-3.0.

2. **USE mpi** This method is described in Section 16.2.14 and requires compile-time argument checking.

3. **USE mpi_f08** This method is described in Section 16.2.15 and requires compile-time argument checking that also includes unique handle types.

Compliant MPI-3 implementations providing a Fortran interface must provide all three Fortran support methods.

Application subroutines and functions may use either one of the modules or the `mpif.h` include file. An implementation may require the use of 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. (*End of advice to users.*)

ticket230-B. ticket230-B. In a single application, it must be possible to link together routines which USE mpi_f08, USE mpi and INCLUDE mpif.h.

ticket234-F. The INTEGER compile-time constant MPI_SUBARRAYS is set to MPI_SUBARRAYS_SUPPORTED if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [35]; otherwise it is set to MPI_SUBARRAYS_NOT_SUPPORTED. This constant exists with each Fortran support method, but not in the C/C++ header files. The value may be different for each Fortran support method. ticket230-B.

Section 16.2.2 to 16.2.11 gives an overview and details on known problems when using Fortran together with MPI; Section 16.2.12 compares the Fortran problems with C. Section 16.2.16 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 16.2.17 and Section 16.2.18 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. ticket238-J. ticket238-J. ticket230-B. ticket238-J. ticket230-B. ticket230-B. ticket230-B. ticket250-V. ticket230-B.

16.2.2 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail. ticket235-G.

The following MPI features are inconsistent with Fortran 90.

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

2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.

3. Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.

4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls.

5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE,

MPI_UNWEIGHTED, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 for more information.

6. The memory allocation routine MPI_ALLOC_MEM can't be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

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

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

MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type MPI_Aint and in Fortran of type `INTEGER`. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking `INTEGER` arguments of `KIND=MPI_ADDRESS_KIND`. A number of new MPI-2 functions also take `INTEGER` arguments of non-default `KIND`. See Section 2.6 on page 17 and Section 4.1.1 on page 87 for more information.

Sections 16.2.3 to 16.2.11 describe several problems between MPI and Fortran and their solutions in detail. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 16.2.16 on page 563.

16.2.3 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90 is technically only allowed if the function is overloaded with a different function for each type. In C, the use of `void*` formal arguments avoids these problems. Similar to C, with Fortran 2008 + TR 29113 (and later) together with the `mpi_f08` module, the problem is avoided by declaring choice arguments with `TYPE(*), DIMENSION(..)`, i.e., as assumed-type and assumed-rank dummy arguments.

Using `INCLUDE mpif.h`, the following code fragment might technically be invalid and may generate a compile-time error.

```
integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

ticket235-G. ticket235-G. 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 non-standard Fortran compiler options preventing type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument. Thus, when using the mpi_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*). ticket235-G. ticket235-G. ticket235-G. ticket235-G. ticket235-G. ticket235-G. ticket235-G.

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

Using INCLUDE 'mpif.h', compiler warnings are not expected unless this include file also uses Fortran explicit interfaces. ticket235-G.

16.2.4 Problems Due to Data Copying and Sequence Association with Subscript Triplets ticket236-H.

ticket230-B. ticket236-H. Arrays with subscript **triplets** describe in 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:

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_SUPPORTED:

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

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

  In this case, the individual elements s(1), s(6), s(11), etc., are sent between the start of MPI_ISEND and the end of MPI_WAIT even though the compiled code must not copy s(1:100:5) to a 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).

  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)` will be filled with the received data when MPI_WAIT returns.

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

> *Rationale.* If MPI_SUBARRAYS equals MPI_SUBARRAYS_SUPPORTED, non-contiguous buffers are handled inside of the MPI library instead of by the 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. If MPI_SUBARRAYS equals MPI_SUBARRAYS_NOT_SUPPORTED, temporary copies made by the compiler will not exist throughout the duration of the entire nonblocking MPI operation, which is too short for implementing the entire MPI operation. (*End of rationale.*)

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_NOT_SUPPORTED:

  Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

  In Fortran 90, user data is not necessarily stored contiguously. For example, the array section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, ... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., `B(N)`) or is of assumed size (e.g., `B(*)`). If necessary, they do this by making a copy of the array into contiguous memory.[1]

ticket236-H.

  Because MPI dummy buffer arguments are assumed-size arrays, this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
    real a(100)
    call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

  Since the first dummy argument to MPI_IRECV is an assumed-size array (`<type> buf(*)`), the array section `a(1:100:2)` is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array `a`. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem

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

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. (We define 'simply contiguous' more fully in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

ticket236-H.
ticket236-H.

Our formal definition of a 'simply contiguous' array section is

ticket236-H.

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. Examples are

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

Because of Fortran's column-major ordering, where the first index varies fastest, a 'simply contiguous' section of a contiguous array will also be contiguous.[2]

ticket236-H.

The same problem can occur with a scalar argument. Some compilers make a copy of some 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.
ticket236-H.

```
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.

[2]To keep the definition of 'simply contiguous' simple, we have chosen to require all but one of the section subscripts to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader that the whole of the dimension is selected. It would have been possible to allow cases where the whole dimension is selected with one or two bounds, but this means for the reader that the array declaration or most recent allocation has to be consulted and for the compiler that a run-time check may be required.

If there is a compiler option that inhibits copying of arguments, in either the calling or called procedure, this should be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, simply contiguous array sections of such arrays, or scalars, and if there is no compiler option to inhibit this, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

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

Arrays with **vector** subscripts describe in Fortran subarrays containing an 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 be used in blocking MPI operations.

16.2.6 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, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C).

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

16.2.7 Fortran Derived Types

MPI supports passing Fortran sequence and BIND(C) derived types to choice dummy arguments, but does not support Fortran derived types that neither have the SEQUENCE nor the BIND(C) attribute.

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

```
type mytype
   [ticket237-I.]SEQUENCE
```

```
       integer i
       real x
       double precision d
    end type mytype

    type(mytype) foo
    integer blocklen(3), type(3)
    integer(MPI_ADDRESS_KIND) disp(3), base

    call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
    call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
    call MPI_GET_ADDRESS(foo%d, disp(3), ierr)

    base = disp(1)
    disp(1) = disp(1) - base
    disp(2) = disp(2) - base
    disp(3) = disp(3) - base

    blocklen(1) = 1
    blocklen(2) = 1
    blocklen(3) = 1

    type(1) = MPI_INTEGER
    type(2) = MPI_REAL
    type(3) = MPI_DOUBLE_PRECISION

    call MPI_TYPE_CREATE_STRUCT(3, blocklen, disp, type, newtype, ierr)
    call MPI_TYPE_COMMIT(newtype, ierr)

[ticket237-I.]
[ticket237-I.]
    call MPI_SEND(foo%i, 1, newtype, ...)
```

ticket230-B.
ticket238-J.

16.2.8 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):

- Usage of nonblocking routines *(Nonbl.)*.

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

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

- Code movements and register optimization problems; see Section 16.2.9 on page 550.
- Temporary data movements and temporary memory modifications. see Section 16.2.10 on page 556.
- Permanent data movements (e.g., through garbage collection). see Section 16.2.11 on page 558.

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 movements and register optimization	yes	yes	no	yes
Temporary data movements	yes	yes	yes	no
Permanent data movements	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 16.2.9 to 16.2.9 on pages 552-556,
- to minimize the additional needs in the Fortran standard, e.g., in the Fortran 2008 TR 29113 [35],
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 16.2.16 on page 563.

ticket238-J.
ticket238-J.

16.2.9 Problems with Code Movements and Register Optimization

Nonblocking operations

If a variable is local to a Fortran subroutine (i.e., not in a module or `COMMON block`), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

Example 16.11 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. It has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as in the case on the right. ticket238-J.

Example 16.12 Similar example with MPI_ISEND

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

Due to valid compiler code movement optimizations in Example 16.12, the content of buf may already be overwritten 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 425. 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 MPI_SEND, MPI_RECV etc., which hides the actual variables involved. MPI_BOTTOM with ticket238-J.

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.

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

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

In Example 16.14, several successive assigments to the same variable buf can be combined in the way, 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

The following sections show in detail how the problems with code movements and register optimizations 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 all operation if MPI_BOTTOM is used or datatype handles that combine several variables:

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

Each of these methods can solve the problems of code movement and register optimization, however, may involve different degree of performance impact, and may not be usable in every application context. The `VOLATILE` attribute may have the most negative impact on performance. There is one attribute that cannot be used for this purpose:

- The Fortran `ASYNCHRONOUS` attribute may not solve code movement problems in MPI applictions.

Table 16.5 shows the usability of each method. Each problem category is descibed in detail below.

	Nonbl.	1-sided	Bottom	Overhead may be
Examples	16.11, 16.12	Section 11.7.3	16.13, 16.14	
`TARGET`	solved	solved	solved	low - medium
MPI_F_SYNC_REG	solved	solved	solved	low
Module Data	solved	solved	solved	low - medium
`VOLATILE`	solved	solved	solved	high - huge
`ASYNCHRONOUS`	NOT solved	NOT solved	NOT solved	medium - high

Table 16.5: Usability of methods to prevent Fortran optimization problems

The Fortran TARGET attribute

ticket238-J.
ticket238-J.

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

- The `TARGET` attribute solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles. Declaring `REAL, TARGET :: buf` solves the register optimization problem in Examples 16.11, 16.12, 16.13, and 16.14

- Unfortunately, the TARGET attribute 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. Example 16.15 on page 556 can **not** be solved with the `TARGET` attribute.

ticket238-J.
ticket238-J.
ticket238-J.

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. ticket238-J. The MPI library provides the MPI_F_SYNC_REG routine for this purpose; see Section 16.2.17 on page 565. ticket238-J. ticket238-J.

- 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                        Example 16.12
  can be solved with                   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                        Example 16.14
  can be solved with                   can be solved with
  call MPI_F_SYNC_REG(buf)             call MPI_F_SYNC_REG(buf)
  call MPI_RECV(MPI_BOTTOM,...)        call MPI_SEND(MPI_BOTTOM,...)
  call MPI_F_SYNC_REG(buf)             call MPI_F_SYNC_REG(buf)
  ```

  The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND; 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 425, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 16.11, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer, i.e., before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

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

call MPI_WIN_FENCE              call MPI_WIN_FENCE
call MPI_F_SYNC_REG(bbbb)       call MPI_F_SYNC_REG(buff)
                                ccc = buff
```

- The temporary memory modification problem, i.e., Example 16.15 on page 556, 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: ticket238-J.

```
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, the a call to MPI_RECV with MPI_BOTTOM as buffer might be replaced by ticket238-J. ticket238-J. ticket238-J. ticket238-J.

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

ticket238-J.
ticket238-J.
ticket238-J.

Module variables and COMMON blocks

An alternative is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure (MPI_RECV in the above example) may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block. ticket238-J. ticket238-J.

- 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 (evil) 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.

The Fortran ASYNCHRONOUS attribute

Declaring a buffer with the `ASYNCHRONOUS` Fortran attribute in a scoping unit (or `BLOCK`) tells the compiler that any statement of the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation. Because a Fortran compiler may implement asynchronous Fortran input/output operations with blocking I/O, the `ASYNCHRONOUS` attribute may be totally ignored by the compiler. Therefore, the `ASYNCHRONOUS` attribute can **not** be used to solve the code movement and register optimization problem in a portable MPI program.

16.2.10 Temporary Data Movements and Temporary Memory Modifications

The compiler is allowed to modify temporarily data in memory. Normally, this problem may occur only when overlapping communication and computation. Example 16.15 shows a possibility that could be problematic.

[ticket238-J.]

Example 16.15 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,...)
```

In the compiler-generated, possible optimization in Example 16.16, buf(100,100) from Example 16.15 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

[ticket238-J.]

Example 16.16 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,...)
DO h=1,100
  tmp(h)=buf(1,h)
END DO
DO j=1,10000
  buf_1dim(h)=...
END DO
DO h=1,100
  buf(1,h)=tmp(h)
END DO
CALL MPI_Wait(req,...)
```

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 16.17 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 occurs:

- With the local buffer at the origin process, between an RMA call and the ensuing synchronization call.
- 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.

Note that such compiler optimization with temporary data movements can **not** be prevented when buf is declared with the ASYNCHRONOUS Fortran attribute.

Rationale. Using the ASYNCHRONOUS attribute for a buffer while asynchronous operations are pending, the access is restricted as if the pending operation involves the whole buffer (see the *pending I/O storage sequence affector*, Section 5.3.4 and Section 9.6.4.1 paragraphs 5 and 6 of the Fortran 2008 standard [34]). Said this, it is invalid to use parts of an array in pending nonblocking operations and other parts in numerical computation if the array is declared as ASYNCHRONOUS. Due to this reason and also due to the reason in Section *The Fortran ASYNCHRONOUS attribute* on page 556, the ASYNCHRONOUS attribute can **not** be used to solve temporary data movement problems even if the meaning of ASYNCHRONOUS would be extended within the Fortran standard from Fortran input/output to general nonblocking operation. (*End of rationale.*)

[ticket238-J.]

Example 16.17 Another optimization is based on the usage of a local memory, 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,...)
```

Note also that compiler optimization with temporary data movements 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 attribute VOLATILE prevents every register or cache optimization. Therefore, VOLATILE may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem, i.e., 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.

ticket238-J.
16.2.11 Permanent Data Movements

ticket238-J.
A Fortran compiler may implement permanent data movements during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An autamatic garbage collection is a use case. Such permanent data movements are 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 movements are executed in parallel to the MPI operation.

This MPI standard requires that the problems with permanent data movements are solved by the MPI library together with the used compiler; see Section 16.2.16 on page 563.
ticket238-J.

16.2.12 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 ticket238-J.
occur in C. Also here, the best advice is to avoid the problem specifically, to use different variables for buffers in nonblocking MPI operations and computation that is executed while the nonblocking operations are pending. ticket238-J.

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

ticket230-B. ticket230-B.

The use of the `mpif.h` include file is strongly discouraged. ticket233-E.

Because Fortran 90 is (for all practical purposes) a superset of Fortran 77, Fortran 90 (and future) programs can use the original Fortran interface. The Fortran bindings are ticket230-B.
compatible with Fortran 77 implicit-style interfaces in most cases. The include file `mpif.h` ticket230-B.
must: ticket230-B.

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

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

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

- Most `mpif.h` include files do not implement compile-time argument checking.
- Therefore, too many bugs in MPI applications are still undetected:
  - Missing ierror as last additional 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.
- The migration from `mpif.h` to the `mpi` module should be without problems (i.e., only substituting `include 'mpif.h'` below an `implicit none` statement by `use mpi` before such `implicit` statement) as long as the application syntax is correct.
- Migrating portable applications to the `mpi` module, it is not expected to experience any compile or runtime problems 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 because of backward compatiblity reasons, and internally `mpif.h` and the `mpi` module can be implemented so that the same 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 requirement two be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` contains only declarations, and because common block declarations can be split among several lines. To support Fortran 77 as well as Fortran 90, it may be necessary to eliminate all comments from `mpif.h`. (*End of advice to implementors.*)

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

16.2.14 Fortran Support Through the `mpi` Module

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

- Define all named MPI constants

ticket230-B.

- Declare MPI functions that return a value.
- Provide explicit interfaces for all MPI routines, i.e., this module guarantees compile-time argument checking, and allows positional and keyword-based argument lists.
- Define all handles as `INTEGER`. This is reflected in the first of the two Fortran interfaces in each MPI function definition throughout this document. ticket231-C.
- ticket243-O. ticket231-C. Define all named handle types and MPI_Status that are used in the `mpi_f08` module. They are needed only when the application converts old-style `INTEGER` handles into a new-style handles with a named type.

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

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.

Advice to implementors. The appropriate `INTENT` may be different from what is given in the MPI generic interface. Implementations must choose `INTENT` so that the function adheres to the MPI standard. (*End of advice to implementors.*)

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran `INTENT`. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy `OUT` argument. Moreover, "constants" such as MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran, but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent is changed in several places by MPI-2. For instance, MPI_IN_PLACE changes the intent of an `OUT` argument to be `INOUT`. (*End of rationale.*) ticket250-V.

ticket230-B. ticket232-D. ticket232-D.

Advice to implementors. Some compilers allow to implement a choice buffer argument in the `mpi` module with the following explicit interface:

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

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

ticket234-F.

16.2.15 Fortran Support Through the `mpi_f08` Module

ticket230-B. ticket230-B.

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

ticket247-S. ticket241-M. ticket230-B. ticket230-B.

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces for all MPI routines, i.e., this module guarantees compile-time argument checking for all arguments which are not `TYPE(*)`. ticket231-C.
- Define all handles with uniquely named handle types (instead of INTEGER handles in the `mpi` module). This is reflected in the second of the two Fortran interfaces in each MPI function definition throughout this document. ticket234-F.
- Set the `INTEGER` compile-time constant MPI_SUBARRAYS to MPI_SUBARRAYS_SUPPORTED and declare choice buffers with the Fortran 2008 TR 29113 feature assumed-type and assumed-rank `TYPE(*), DIMENSION(..)` if the underlying Fortran compiler supports it. With this, the use of non-contiguous sub-arrays is valid also in nonblocking routines.
- Set the MPI_SUBARRAYS compile-time constant to MPI_SUBARRAYS_NOT_SUPPORTED 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 in nonblocking calls may be restricted as with the `mpi` module.

ticket234-F.

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

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

(*End of advice to implementors.*)

ticket242-N.

- Declare each argument with an `INTENT=IN`, `OUT`, or `INOUT` as appropriate.

ticket242-N.

Rationale. 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.*)

ticket244-P.

- Declare all status and array_of_statuses output arguments as optional through function overloading, instead of using MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE. For this, two specific bindings are provided in a gerneric interface where one has the optional argument and one does not.

ticket244-P.

- Declare all array_of_errcodes output arguments as optional through function overloading, instead of using MPI_ERRCODES_IGNORE.

ticket244-P.

- Declare all sourceweights, destweights, and weights arguments as optional through function overloading, instead of using MPI_UNWEIGHTED.

ticket244-P.

Rationale. To assure that the correspondence of the optional ierror argument can always be established by position, it is necessary that other arguments are not optional through using the `OPTIONAL` attribute. Therefore, function overloading is used for such other arguments. (*End of rationale.*)

ticket239-K.

- Declare all ierror output arguments as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their 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. These user-defined functions need not to check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

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 [35] 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 provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. "It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations."[3]

This TR 29113 contains language features that are needed for the MPI bindings in the `mpi_f08` module: assumed-type and assumed-rank. It is important that any

[3][36] page iv, paragraph 7.

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, 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 can be used in an implementation with assumed-type and assumed-rank in an explicit interface.

Furthermore, the implementors of the MPI Fortran bindings can freely choose whether the interfaces are declared within a Fortran `INTERFACE` or `CONTAINS` construct, and whether all bindings are defined as native Fortran or a `BIND(C)` interface. The `INTERFACE` construct in combination with `BIND(C)` allows to implement the Fortran `mpi_f08` interface with a single set of portable wrapper routines written in C; to support all desired features in the `mpi_f08` interface, TR 29113 also has a provision for `OPTIONAL` arguments in `BIND(C)` interfaces."

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

16.2.16 Requirements on Fortran Compilers

ticket230-B.
ticket230-B.

The compliance to MPI-3.0 (and later) Fortran bindings is not only a property of the MPI library itself, but is always a property of an MPI library together with the Fortran compiler it is compiled for.

Advice to users. Many MPI libraries are shipped together with special compilation scripts (e.g., `mpif90`, `mpicc`). These scripts start the compiler probably together with special options to guarantee this compliance. (*End of advice to users.*)

An MPI library is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 16.2.3 to 16.2.11 work correctly. Based on this rule, major requirements are for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`): ticket234-F.

- The language features assumed-type and assumed-rank from Fortran 2008 TR 29113 [35] are available. 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, which implements the `mpi_f08` module with MPI_SUBARRAYS set to MPI_SUBARRAYS_NOT_SUPPORTED. This is required only for `mpi_f08`. ticket230-B.
- Simply contiguous arrays and scalars must be passed to choice buffer dummy arguments with call by reference. ticket237-I.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments and they are passed with call by reference. ticket238-J.
- The `TARGET` attribute (as described in Section *The Fortran TARGET attribute* on page 553) solves the problems described in Section 16.2.9 on page 550 independent of whether the MPI library internally uses Fortran or C pointers to memorize the location of a buffer between start and completion of a nonblocking operation, and to handle absolute addresses in MPI derived datatype handles. ticket238-J.

- A separately compiled empty routine (as MPI_F_SYNC_REG on page 554 and Section 16.2.17 on page 565, and DD on page 555) solves the problems described in Section 16.2.9 on page 550. ticket238-J.

- The problems with temporary data movements (as described in Section 16.2.10 on page 556) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective IO) and the computation when overlapping communication and computation. ticket238-J.

- Problems caused by automatic and permanent data movements (e.g., within a garbage collection, see Section 16.2.11 on page 558) 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. 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*(*)` strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods. ticket231-C.

ticket243-O.

- The handle and status types in `mpi_f08` (i.e., sequence derived types with `INTEGER` elements) are (handle) or can be (status) identical to one numerical storage unit or a sequence of those. These types must be valid at every location where an `INTEGER` and a fixed-size array of `INTEGER`s (i.e., handle and status in the `mpi` module and `mpif.h`) is valid, especially also within `BIND(C)` derived types defined by the application.

  Rationale. This is not yet part of the draft N1845 of TR 29113 [36], but may be part of the final version of this TR 29113 [35]. It is already implemented in some of the available Fortran compilers (e.g., ifort and pgi).
  A contrary definition of handles and statuses would have been `BIND(C)` derived types. With such definition, it would have been necessary that such `BIND(C)` derived types could be part of application-defined sequence derived types which is contratictory to the current rules about `BIND(C)` and Fortran storage units. (*End of rationale.*)

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 interfces 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 [35]. Some of these requirements for MPI-3.0 are beyond of TR 29113. (*End of advice to implementors.*)

ticket230-B.

Further requirements apply if the MPI library internally uses `BIND(C)` routine interfaces:

- Non-buffer arguments are `INTEGER`, `INTEGER(KIND=...)`, `LOGICAL`, `CHARACTER*(*)`, and sequence derived types (handles and status in `mpi_f08`) variables and arrays, and `EXTERNAL` routines; 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 because these types may

ticket231-C. ticket230-B.

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 through TR 29113 (e.g., `CHARACTER*(*)` and sequence derived types (not yet part of draft N1845)).

ticket231-C.
ticket230-B.
ticket239-K.

- `OPTIONAL` dummy arguments are also valid within `BIND(C)` interfaces. This requirements is part of the TR 29113.

ticket238-J.
ticket238-J.

16.2.17 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 16.2.9 on page 550, a dummy call is needed to tell the compiler that registers are to be flushed for a given buffer. It has only a Fortran binding.

MPI_F_SYNC_REG(buf)

INOUT	buf	initial address of buffer (choice)

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

ticket-248T.

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

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

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

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

16.2.18 Additional Support for Fortran Numeric Intrinsic Types

The routines in this section are part of Extended Fortran Support described in Section 16.2.14.

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`

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.

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 sequence derived type that contains an `INTEGER` field named MPI_VAL. This `INTEGER` value can be used in the following conversion functions. ticket231-C.

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.

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

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

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

Example 16.20 A C library routine is called from a C++ program. The C library routine is prototyped to take an `MPI_Comm` as an argument.

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

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

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

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

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

16.3.5 Status

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 f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, respectively. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

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

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

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

Advice to users. There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status. (*End of advice to users.*) ticket250-V.

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 datatype `MPI_F_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_F_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_F_status *f08_status)
```

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

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

MPI_STATUS_F2F08(f_status, f08_status)

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

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

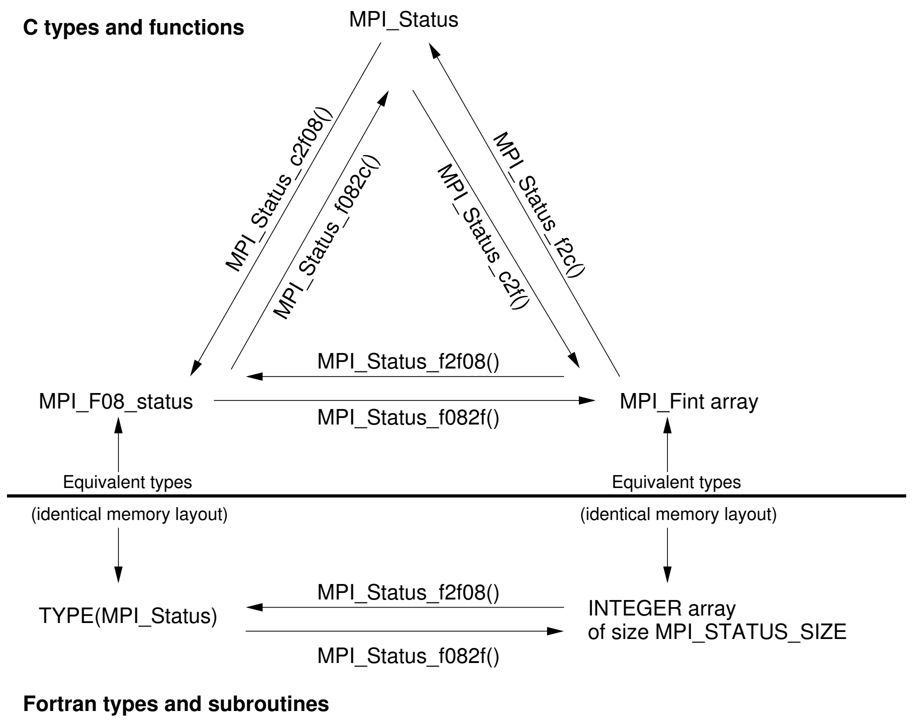

Figure 16.1: Status conversion routines

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

ticket-248T.

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

This routine converts a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array into a Fortran mpi_f08 TYPE(MPI_Status).

MPI_STATUS_F082F(f08_status, f_status)

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

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

```
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
    TYPE(MPI_Status) :: F08_STATUS
    INTEGER :: F_STATUS(MPI_STATUS_SIZE)
    INTEGER IERROR
```

ticket-248T.

```
MPI_Status_f082f(f08_status, f_status, ierror)
```

```
    TYPE(MPI_Status), INTENT(IN) ::  f08_status
    INTEGER, INTENT(OUT) ::  f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array.

16.3.6 MPI Opaque Objects

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

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

Datatypes

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

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see 16.3.9, page 588).

Example 16.21

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

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

/* C code */

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

Null Handles

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

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

Empty group

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

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

Deprecated predefined functions

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

File Operation Constants, Part 2

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_DISTRIBUTE_BLOCK	MPI::DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG
MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE
MPI_ORDER_C	MPI::ORDER_C
MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN
MPI_SEEK_CUR	MPI::SEEK_CUR
MPI_SEEK_END	MPI::SEEK_END
MPI_SEEK_SET	MPI::SEEK_SET

F90 Datatype Matching Constants

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

Constants Specifying Empty or Ignored Input

C/Fortran name C type / Fortran type [ticket244-P.]with `mpi` module [ticket244-P.]/ Fortran type with `mpi_f08` module	C++ name C++ type
MPI_ARGVS_NULL `char***` / 2-dim. array of `CHARACTER*(*)` [ticket244-P.]/ 2-dim. array of `CHARACTER*(*)`	MPI::ARGVS_NULL `const char ***`
MPI_ARGV_NULL `char**` / array of `CHARACTER*(*)` [ticket244-P.]/ array of `CHARACTER*(*)`	MPI::ARGV_NULL `const char **`
MPI_ERRCODES_IGNORE `int*` / `INTEGER` array [ticket244-P.]/ not defined	Not defined for C++
MPI_STATUSES_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE,*)` [ticket244-P.]/ not defined	Not defined for C++
MPI_STATUS_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE)` [ticket244-P.]/ not defined	Not defined for C++
MPI_UNWEIGHTED `int*` / `INTEGER` array [ticket244-P.]/ not defined	Not defined for C++

MPI::File
MPI::Group
MPI::Info
MPI::Op
MPI::Request
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)
```

ticket231-C.

```
!  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)
```

ticket0.

A.1.3 Prototype Definitions

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

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

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

typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
              void *attribute_val, void *extra_state);

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
              int type_keyval, void *extra_state,
```

ticket247-S.

A.4 Fortran 2008 Bindings with the mpi_f08 Module

A.4.1 Point-to-Point Communication Fortran 2008 Bindings

ticket-248T.

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

ticket-248T.

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

ticket-248T.

```
MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer
    INTEGER, INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

ticket-248T.

```
MPI_Buffer_detach(buffer_addr, size, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer_addr
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

ticket-248T.

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

ticket-248T.

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

ticket-248T.

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

ticket-248T.

```
MPI_Iprobe(source, tag, comm, flag, status, ierror)
    INTEGER, INTENT(IN) ::  source, tag
```

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

ticket231-C.

1. Section 2.5.1 on page 12, Section 16.2.14 on page 560, Section 16.2.15 on page 561, and Section 16.2.16 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.

ticket244-P.

2. Section 2.5.2 on page 14, Section 3.2.6 on page 36, Section 7.5.4 on page 294, Section 7.5.5 on page 301, Section 10.3 on page 358, Section 10.3.3 on page 363, Section 12.2 on page 427, and Section 16.2.15 on page 561.
   With the mpi_f08 module, optional arguments through function overloading are used instead of MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, and MPI_UNWEIGHTED.

ticket234-F.

3. Section 2.5.5 on page 16, Section 16.2.15 on page 561, and Section 16.2.16 on page 563.
   Choice buffers are defined as assumed-type and assumed-rank according to Fortran 2008, TR 29113 [35] within the mpi_08 Fortran module.

ticket239-K.

4. Section 2.6.2 on page 17, Section 16.2.15 on page 561, and Section 16.2.16 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 16.2.14 on page 560, Section 16.2.15 on page 561, Section 16.2.16 on page 563, and Section 16.3.5 on page 578.
   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,

ticket245-Q.

6. Section 8.2 on page 320.
   TODO: MPI_ALLOC_MEM and C pointer (C_PTR) instead of only with Cray pointer.

ticket236-H.

7. Section 16.2.4 on page 545, Section 16.2.5 on page 548, Section 16.2.14 on page 560, and Section 16.2.15 on page 561.
   A new method is available to use subscript triplets in nonblocking Fortran operations; vector subscripts are not supported in nonblocking operations.

ticket237-I.

8. Section 16.2.7 on page 548, and Section 16.2.16 on page 563.
   Fortran `SEQUENCE` and `BIND(C)` derived application types can be used as buffers in MPI operations.

ticket238-J.

9. Section 16.2.8 on page 549 to Section 16.2.11 on page 558, Section 16.2.16 on page 563, and Section 16.2.17 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 `TARGET` attribute. The Fortran routine MPI_F_SYNC_REG is added. To achieve a secure and portable programming interfaces, in Section 16.2.16, 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 16.2.13 on page 559.
    The use of the `mpif.h` Fortran include file is strongly discouraged.

ticket232-D.

11. Section 16.2.14 on page 560, and Section 16.2.16 on page 563.
    The existing `mpi` Fortran module must implement compile-time argument checking.

ticket230-B.
ticket247-S.
ticket248-T.
ticket252-W.

12. Section 16.2.15 on page 561, and Section 16.2.16 on page 563.
    The new `mpi_08` Fortran module is introduced. In some routines, the dummy argument names were changed. The new dummy argument names must be used because the `mpi_08` module guarantees keyword-based actual argument lists.

ticket242-N.

13. Section 16.2.15 on page 561.
    Within the `mpi_08` Fortran module, dummy arguments are declared with `INTENT=IN`, `OUT`, or `INOUT` as defined in the `mpi_08` interfaces.

ticket250-V.

14. Section A.1.3 on page 604.
    In some routines, the Fortran callback prototype names were changed.

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 529.
   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

[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`. 16.2.14

[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. *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. *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. *Information technology – Programming languages – Fortran – Part 1: Base language*, November 2010. 16.2.1, 16.2.10, 16.2.15

[35] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva. *TR on further interoperability with C*, MONTH NOT YET DEFINED 2011. 16.2.1, 16.2.1, 16.2.5, 16.2.15, 16.2.16, 3

[36] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva. *TR on further interoperability with C*, March, 3 2011. 3, 16.2.16

[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

[38] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1