MPI: A Message-Passing Interface Standard

Version 3.0

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

March 26, 2011

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

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

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

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

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

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

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

All MPI functions are first specified in the language-independent notation. Immediately below this, language dependent bindings follow:

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

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

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

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

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

arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5.2 Array Arguments

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

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

2.5.3 State

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

2.5.4 Named Constants

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

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

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

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

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

and their C++ counterparts where appropriate.

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

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

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

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file `mpif.h` or the `mpi` module, the document uses <type> to represent a choice variable; with the Fortran `mpi_f08` module, such arguments are declared with the Fortran 2008 syntax TYPE(*), DIMENSION(..); for C and C++, we use void *.

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

2.5.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C, MPI::Aint in C++ and `INTEGER (KIND=MPI_ADDRESS_KIND)` in Fortran. These types must have the same width and encode address values in the same manner such that address values in one language may be passed directly to another language without conversion. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

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

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

Table 2.1: Deprecated constructs

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

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

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

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

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

Array arguments are indexed from one.

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

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

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

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

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

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

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

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

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

3.2.5 Return Status

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

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

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

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

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

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

In C++, the status object is handled through the following methods:
{`int MPI::Status::Get_source() const` *(binding deprecated, see Section 15.2)* }

cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

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

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

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

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

Example 3.1 The `mpi_f08` bindings for MPI_PROBE are:

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

Example 3.2 The C++ bindings for MPI_PROBE are:

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

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(oldtype, newtype)

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

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

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

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

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

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

4.1.11 Use of General Datatypes in Communication

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

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

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

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

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

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

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

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(user_fn, commute, op)

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

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

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

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

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

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

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

The ISO C prototype for the function is the following.

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

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

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

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

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

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

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

The C callback functions are:

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

and

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

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

The Fortran callback functions are:

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

and

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

The C++ callbacks are:

```
{typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
              int comm_keyval, void* extra_state, void* attribute_val_in,
              void* attribute_val_out, bool& flag); (binding deprecated, see
              Section 15.2)}
```

and

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

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 runtime ways to determine which version of the standard is in use in the environment one is using.

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

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

in Fortran,

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

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

```
int MPI_Get_version(int *version, int *subversion)
```

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

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

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

The info argument The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the `mpi_f08` module, MPI::Info in C++ and `INTEGER` in Fortran with the `mpi` module or the include file `mpif.h`. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated `char*` in C, `character*(*)` in Fortran). Routines to create and manipulate the info argument are described in Section 9 on page 349.

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

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 10.3.4 on page 366). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

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

The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array_of_errcodes is filled in with the value MPI_SUCCESS. If only m ($0 \leq m <$ maxprocs) processes are spawned, m of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. In C or in the Fortran `mpi` module or `mpif.h` include file, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. Inthe Fortran `mpi_f08` module or in C++ this constant does not exist, and the array_of_errcodes argument may be omitted from the argument list.

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

MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users. (*End of rationale.*)

11.7.3 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory.

The problem is illustrated by the following code:

Source of Process 1	**Source of Process 2**	**Executed in Process 2**
`bbbb = 777`	`buff = 999`	`reg_A:=999`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
`call MPI_PUT(bbbb`		`stop appl. thread`
`into buff of process 2)`		`buff:=777 in PUT handler`
		`continue appl. thread`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
	`ccc = buff`	`ccc:=reg_A`

In this example, variable `buff` is allocated in the register `reg_A` and therefore `ccc` will have the old value of `buff` and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 16.2.2.

MPI implementations will avoid this problem for standard conforming C programs. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in in modules or `COMMON` blocks, or to variables that were declared `VOLATILE` (but this attribute may inhibit optimization of any code containing the RMA window). Further details and additional solutions are discussed in Section 16.2.2, "A Problem with Register Optimization and Temporary Memory Modifications," on page 549. See also, "Problems Due to Data Copying and Sequence Association," on page 545, for additional Fortran problems.

Example 16.10 `mpi_profile.cc`, to be compiled into `libpmpi.a`.

```
int MPI::Comm::Get_size() const
{
  // Do profiling stuff
  int ret = pmpi_comm.Get_size();
  // More profiling stuff
  return ret;
}
```

(*End of advice to implementors.*)

16.2 Fortran Support

16.2.1 Overview

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

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008, the only new language features used, are of assumed type and assumed rank. They were defined to allow the definition of choice arguments as part of the Fortran language. (*End of rationale.*)

MPI defines three methods of Fortran support:

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

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

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

Advice to users. It is recommended to use one of the MPI modules even if it is not necessary to use it to avoid type mismatch errors on a particular system. Using a module provides several potential advantages over using an include file. (*End of advice to users.*)

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

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

Section 16.2.2 gives an overview on known problems when using Fortran together with MPI. Section 16.2.6 and Section 16.2.7 describe additional functionality that is part of the Fortran support. funcMPI_F_SYNC_REG is needed for one of the methods to prevent register optimization problems. A set of functions provides additional support for Fortran intrinsic numeric types, including parameterized types. The functions are: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. Parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters.

16.2.2 Problems With Fortran Bindings for MPI

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

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

The following MPI features are inconsistent with Fortran 90.

1. An MPI subroutine with a choice argument may be called with different argument types. Using the `mpi_f08` module, this problem is resolved.

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

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

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

5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_UNWEIGHTED, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 for more information.

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

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

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

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

Problems Due to Strong Typing

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

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

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

In practice, it is rare for compilers to do more than issue a warning. Using the `mpi_f08` or `mpi` module, the problem is usually resolved through the standardized assume-type and assume-rank declarations of the dummy arguments, or with non-standard Fortran options preventing type checking for choice arguments.

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

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

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

Problems Due to Data Copying and Sequence Association

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_SUPPORTED:

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

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

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

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

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

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

- If MPI_SUBARRAYS equals MPI_SUBARRAYS_UNSUPPORTED:

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

  In Fortran 90, user data is not necessarily stored contiguously. For example, the array section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, ... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., `B(N)`) or is of assumed size (e.g., `B(*)`). If necessary, they do this by making a copy of the array into contiguous memory. Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.[1]

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

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

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

  Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a 'simple' section such as `A(1:N)` of such an array. (We define 'simple' more fully in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

  Our formal definition of a 'simple' array section is

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

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

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

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

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

The same problem can occur with a scalar argument. Some compilers, even for Fortran 77, make a copy of some scalar dummy arguments within a called procedure. That this can cause a problem is illustrated by the example

```
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end
```

If `a` is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter `a` itself.

Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., `A(1:n:2)`), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.

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

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

Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4 on page 15. In C, these are implemented as constant pointers, usually as `NULL` and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through `parameter` statements) is not possible because an

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

implementation cannot distinguish these values from legal data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). With USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT dummy arguments that allow one of these special constants as input, an INTENT(...) is not specified.

Fortran Derived Types

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

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

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

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

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

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

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

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

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

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

A Problem with Register Optimization and Temporary Memory Modifications

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. This section discusses register optimization pitfalls and problems with temporary memory modifications. These problems are independent of the Fortran support method, i.e., they occur with the `mpi_f08` module, the `mpi` module, and the `mif.h` include file.

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

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

The compiler is allowed to cause two optimization problems

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

The optimization problems can occur not in all usage areas:

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

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

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

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

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

The next paragraphs describe the problems in detail.

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

Example 16.11 shows extreme, but allowed, possibilities.

Example 16.11 Fortran 90 register optimization – extreme.

Source
```
REAL :: buf, b1
call MPI_IRECV(buf,..req)

call MPI_WAIT(req,..)
b1 = buf
```

compiled as
```
REAL :: buf, b1
call MPI_IRECV(buf,..req)
register = buf
call MPI_WAIT(req,..)
b1 := register
```

or compiled as
```
REAL :: buf, b1
call MPI_IRECV(buf,..req)
b1 = buf
call MPI_WAIT(req,..)
```

MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_IRECV and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has returned, and may schedule the load of buf earlier than typed in the source. It has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as in the case on the right.

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

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

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

Example 16.12 Similar example with MPI_ISEND

Source

```
REAL :: buf, copy
buf = val
call MPI_ISEND(buf,..req)
copy = buf

call MPI_WAIT(req,..)
buf = val_overwrite
```

compiled as

```
REAL :: buf, copy
buf = val
call MPI_ISEND(buf,..req)
copy=buf
buf = val_overwrite
call MPI_WAIT(req,..)
```

or compiled as

```
REAL :: buf, copy
buf = val
addr = &buf
copy = val
buf = val_overwrite
send(*addr)
```

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

Example 16.13 Overlapping Communication and Computation

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

The compiler may substitute the nested loops through loop fusion by

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

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

Note, that this problem occurs also

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

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

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

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

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

One-sided communication. Normally users are not afflicted with this. But the user should pay attention to this section if in his/her program a buffer argument to an MPI_SEND, MPI_RECV etc., uses a name which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one mentioned in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 16.14 shows what Fortran compilers are allowed to do.

Example 16.14 Fortran 90 register optimization.

This source ...

```
call MPI_GET_ADDRESS(buf,bufaddr,
                ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,
                bufaddr,
                MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf

call MPI_RECV(MPI_BOTTOM,1,type,...)
val_new = buf
```

can be compiled as:

```
call MPI_GET_ADDRESS(buf,...)

call MPI_TYPE_CREATE_STRUCT(...)


call MPI_TYPE_COMMIT(...)
register = buf
val_old = register
call MPI_RECV(MPI_BOTTOM,...)
val_new = register
```

The compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access of buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

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

Example 16.15 Similar example with MPI_SEND

This source ...

```
! buf contains val_old
buf = val_new

call MPI_SEND(MPI_BOTTOM,1,type,...)
! with buf as a displacement in type
buf = val_overwrite
```

can be compiled as:

```
! buf contains val_old
! dead code:
!   buf=val_new is removed
call MPI_SEND(...)
! i.e. val_old is sent
buf = val_overwrite
```

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

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

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

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

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

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

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

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

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

The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides MPI_F_SYNC_REG for this purpose, see Section 16.2.6 on page 560.

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

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

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

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

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

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

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

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

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

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

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

Note that if the intent is declared in the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, the above call of MPI_RECV might be replaced by

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

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

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

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

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe.

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

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

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

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

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

Advice to implementors. To make `mpif.h` compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended that requirement two be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` contains only declarations, and because common block declarations can be split among several lines. To support Fortran 77 as well as Fortran 90, it may be necessary to eliminate all comments from `mpif.h`. (*End of advice to implementors.*)

16.2.4 Fortran Support Through the `mpi` Module

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

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

An MPI implementation may provide other features in the `mpi` module that enhance the usability of MPI while maintaining adherence to the standard. For example, it mayprovide `INTENT` information in these interface blocks.

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

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

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

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

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

16.2.5 Fortran Support Through the `mpi_f08` Module

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

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

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

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

(*End of advice to implementors.*)

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

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

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

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

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

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

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

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

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

16.2.6 Additional Support for Fortran Register-Memory-Synchronization

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

MPI_F_SYNC_REG(buf)

INOUT	buf	initial address of buffer (choice)

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

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

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

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

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

16.2.7 Additional Support for Fortran Numeric Intrinsic Types

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

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL and

16.3.4 Transfer of Handles

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

The type definition MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran `mpi` module or the `mpif.h` include file, a Fortran handle is a Fortran `INTEGER` value that can be used in the following conversion functions. With the Fortran `mpi_f08` module, a Fortran handle is a derived tpye that contains the Fortran `INTEGER` field MPI_VAL, which contains the the INTEGER value that can be used in the following conversion functions.

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.5 on page 22.

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

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

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

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

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

```
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Request MPI_Request_f2c(MPI_Fint request)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_File MPI_File_f2c(MPI_Fint file)

MPI_Fint MPI_File_c2f(MPI_File file)

MPI_Win MPI_Win_f2c(MPI_Fint win)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Info MPI_Info_f2c(MPI_Fint info)

MPI_Fint MPI_Info_c2f(MPI_Info info)
```

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

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

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

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

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

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

This C routine converts a C c_status into a Fortran `mpi_f08` f08_status.
Conversion between the two Fortran versions of a status can be done with:

MPI_STATUS_F2F08(f_status, f08_status)

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

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

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

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

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

MPI_STATUS_F082F(f08_status, f_status)

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

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

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

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

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

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

16.3.6 MPI Opaque Objects

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

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

Datatypes

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

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see 16.3.9, page 583).

Example 16.19

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

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

Null Handles

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

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

Empty group

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

Topologies

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

Predefined functions

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

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

Deprecated predefined functions

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

File Operation Constants, Part 2

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

F90 Datatype Matching Constants

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

Constants Specifying Empty or Ignored Input

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

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

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

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

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

A.1.3 Prototype Definitions

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

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

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

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

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

A.4 Fortran 2008 Bindings with the mpi_f08 Module

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

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

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

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

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

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

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

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

MPI_Iprobe(source, tag, comm, flag, status, ierror)
    INTEGER, INTENT(IN) ::  source, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    LOGICAL, INTENT(OUT) ::  flag
```