D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

February 6, 2015
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices.

2.1 Document Notation

Rationale. Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (*End of advice to implementors.*)

2.2 Naming Conventions

In many cases MPI names for C functions are of the form MPI_Class_action_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules.

1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form MPI_CLASS_ACTION.

2. If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran.

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

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

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

2.3 Procedure Specification

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

- IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

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

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

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

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

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

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

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

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

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

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

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

- The ISO C version of the function.
- The Fortran version used with `USE mpi_f08`.
- The Fortran version of the same function used with `USE mpi` or `INCLUDE 'mpif.h'`.

An exception is Section 14.3 "The MPI Tool Information Interface", which only provides ISO C interfaces. #447

"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 it may return before the associated operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. The word complete is used with respect to operations and any associated requests and/or communications. An *operation completes* when the user is allowed to reuse resources, and any output buffers have been updated. #451

Text block at end of this section was removed

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

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

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

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

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

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

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

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

2.5 Data Types

2.5.1 Opaque Objects

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

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

```
TYPE, BIND(C) :: MPI_Comm
  INTEGER   :: MPI_VAL
END TYPE MPI_Comm
```

Not changed in #388

The 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. (*End of advice to implementors.*)

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle `COMM` can be converted directly into an exactly equivalent `mpi_f08` communicator handle named `comm_f08` by `comm_f08%MPI_VAL=COMM`, and vice versa. The use of the `INTEGER` defined handles and the `BIND(C)` derived type handles is different: Fortran 2003 (and later) define that `BIND(C)` derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these `BIND(C)` derived type handles. Most compilers use one unit for both, the `INTEGER` handles and the handles defined as `BIND(C)` derived types. (*End of rationale.*)

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

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

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

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.

Rationale. This design hides the internal representation used for MPI data structures, thus allowing similar calls in 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 in C 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. In Fortran, the handles are defined such that assignment and comparison are available through the operators of the language or overloaded versions of these operators. (*End of rationale.*) #449

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

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

2.5.2 Array Arguments

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

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, see Chapter 8. #450 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 `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 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 `switch` and Fortran `case/select` statements) are:

MPI_MAX_PROCESSOR_NAME
MPI_MAX_LIBRARY_VERSION_STRING
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_VERSION
MPI_SUBVERSION
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_COUNT_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)

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
MPI_WEIGHTS_EMPTY
```

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 valid 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 + TR 29113 syntax `TYPE(*), DIMENSION(..)`; for C, we use `void *`.

Advice to implementors. Implementors can freely choose how to implement choice arguments in the `mpi` module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the `mpi_f08` module. See details in Section 17.1.1. (*End of advice to implementors.*)

2.5.6 Absolute Addresses and Relative Address Displacements

#431

Some MPI procedures use *address* arguments that represent an absolute address in the calling program, or relative displacement arguments that represent differences of two absolute addresses. The datatype of such arguments is 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. For retrieving absolute addresses or any calculation with absolute addresses, one should use the routines and functions provided in Section 4.1.5. Section 4.1.12 provides additional rules for the correct use of absolute addresses. For expressions with relative displacements or other usage without absolute addresses, intrinsic operators (e.g., `+`, `-`, `*`) can be used.

#421

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

2.5.8 Counts

As described above, MPI defines types (e.g., MPI_Aint) to address locations within memory and other types (e.g., MPI_Offset) to address locations within files. In addition, some MPI procedures use *count* arguments that represent a number of MPI datatypes on which to operate. At times, one needs a single type that can be used to address locations within either memory or files as well as express *count* values, and that type is MPI_Count in C and `INTEGER (KIND=MPI_COUNT_KIND)` in Fortran. These types must have the same width and encode values in the same manner such that count values in one language may be passed directly to another language without conversion. The size of the MPI_Count type is determined by the MPI implementation with the restriction that it must be minimally capable of encoding any value that may be stored in a variable of type `int`, MPI_Aint, or MPI_Offset in C and of type `INTEGER`, `INTEGER (KIND=MPI_ADDRESS_KIND)`, or `INTEGER (KIND=MPI_OFFSET_KIND)` in Fortran.

> *Rationale.* Count values logically need to be large enough to encode any value used for expressing element counts, type maps in memory, type maps in file views, etc. For backward compatibility reasons, many MPI routines still use `int` in C and `INTEGER` in Fortran as the type of count arguments. (*End of rationale.*)

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, and ISO C, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the `mpi_f08` module, two new Fortran features, *assumed type* and *assumed rank*, are also required, see Section 2.5.5.

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

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid any prefix of the form "`MPI_`" and "`PMPI_`", where any of the letters are either upper or lower case. #453

2.6.1 Deprecated and Removed Names and Functions

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with newer

versions of MPI. For example, the Fortran binding for MPI-1 functions that have address arguments uses `INTEGER`. This is not consistent with the C binding, and causes problems on machines with 32 bit `INTEGER`s and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions was declared as deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated. #452

Some of the deprecated constructs are now removed, as documented in Chapter 16. They may still be provided by an implementation for backwards compatibility, but are not required.

Table 2.1 shows a list of all of the deprecated and removed constructs. Note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

Deprecated or removed construct	deprecated since	removed since	Replacement
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_LB[1]	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_UB[1]	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function[2]	MPI-2.0	MPI-3.0	MPI_Comm_errhandler_function[2]
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
MPI_DUP_FN[3]	MPI-2.0		MPI_COMM_DUP_FN[3]
MPI_NULL_COPY_FN[3]	MPI-2.0		MPI_COMM_NULL_COPY_FN[3]
MPI_NULL_DELETE_FN[3]	MPI-2.0		MPI_COMM_NULL_DELETE_FN[3]
MPI_Copy_function[2]	MPI-2.0		MPI_Comm_copy_attr_function[2]
COPY_FUNCTION[3]	MPI-2.0		COMM_COPY_ATTR_FUNCTION[3]
MPI_Delete_function[2]	MPI-2.0		MPI_Comm_delete_attr_function[2]
DELETE_FUNCTION[3]	MPI-2.0		COMM_DELETE_ATTR_FUNCTION[3]
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
MPI_COMBINER_HVECTOR_INTEGER[4]	-	MPI-3.0	MPI_COMBINER_HVECTOR[4]
MPI_COMBINER_HINDEXED_INTEGER[4]	-	MPI-3.0	MPI_COMBINER_HINDEXED[4]
MPI_COMBINER_STRUCT_INTEGER[4]	-	MPI-3.0	MPI_COMBINER_STRUCT[4]
MPI::...	MPI-2.2	MPI-3.0	C language binding

[1] Predefined datatype.
[2] Callback prototype definition.
[3] Predefined callback routine.
[4] Constant.
Other entries are regular MPI routines.

Table 2.1: Deprecated and Removed constructs

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the `mpi_f08` module is used.

All MPI names have an `MPI_` prefix, and all characters are capitals. Programs must not declare names, e.g., for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix `MPI_`, with the exception of `MPI_` routines written by the user to make use of the profiling interface. To avoid conflicting with the profiling interface, programs must also avoid subroutines and functions with the prefix `PMPI_`. This is mandated to avoid possible name collisions. #453

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 19.

Constants representing the maximum length of a string are one smaller in Fortran than in C as discussed in Section 17.2.9.

Handles are represented in Fortran as `INTEGER`s, or as a `BIND(C)` derived type with the `mpi_f08` module; see Section 2.5.1. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

The older MPI Fortran bindings (`mpif.h` and `use mpi`) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 17.1.16.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare names (identifiers), e.g., for variables, functions, constants, types, or macros, beginning with any prefix of the form `MPI_`, where any of the letters are either upper or lower case. An exception are `MPI_` routines written by the user to make use of the profiling interface. To support the profiling interface, programs must not declare functions with names beginning with any prefix of the form `PMPI_`, where any of the letters are either upper or lower case. #453

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

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

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

Array arguments are indexed from zero.

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

Choice arguments are pointers of type `void *`.

Text block at end of this section was removed #455

sequence changed:
always MPI_..., PMPI_...

2.6.4 Functions and Macros

An implementation is allowed to implement MPI_WTIME, PMPI_WTIME, MPI_WTICK, PMPI_WTICK, MPI_AINT_ADD, PMPI_AINT_ADD, MPI_AINT_DIFF, PMPI_AINT_DIFF, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 17.2.4, and no others, as macros in C. #349 + #404 + #421

Advice to implementors. Implementors should document which routines are implemented as macros. (*End of advice to implementors.*)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (*End of advice to users.*)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Of course, MPI programs may still be erroneous. A *program error* can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in

any implementation. In addition, a *resource error* may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller in a consistent state.

Another subtle issue arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode). Such an error must be treated as fatal, since information cannot be returned for the user to recover from it.

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services

are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as `write` in Fortran and `printf` and `malloc` in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that `printf` is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

In addition, calls that fail because of resource exhaustion or other error are not considered a violation of the requirements here (however, they are required to complete, just not to complete successfully).

2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended that it not use `SIGALRM`, `SIGFPE`, or `SIGIO`. Implementations may also prohibit the use of MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

Index