D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

February 7, 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 17

Language Bindings

17.1 Fortran Support

17.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [2] + TS 29113 [3].

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 + TS 29113, the major new language features used are the `ASYNCHRONOUS` attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 17.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

1. **USE mpi_f08:** This method is described in Section 17.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.

2. **USE mpi:** This method is described in Section 17.1.3 and requires compile-time argument checking. Handles are defined as `INTEGER`. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.

3. **INCLUDE 'mpif.h':** This method is described in Section 17.1.4. The use of the include file `mpif.h` is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

Compliant MPI-3 implementations providing a Fortran interface must provide one or both of the following:

- The `USE mpi_f08` Fortran support method.
- The `USE mpi` and `INCLUDE 'mpif.h'` Fortran support methods.

Section 17.1.6 describes restrictions if the compiler does not support all the needed features.

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

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

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

The `LOGICAL` compile-time constant MPI_SUBARRAYS_SUPPORTED is set to `.TRUE.` if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [3]; otherwise it is set to `.FALSE.`. The `LOGICAL` compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING is set to `.TRUE.` if the `ASYNCHRONOUS` attribute was added to the choice buffer arguments of all nonblocking interfaces **and** the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TS 29113), otherwise it is set to `.FALSE.`. These constants exist for each Fortran support method, but not in the C header file. The values may be different for each Fortran support method. All other constants and the integer values of handles must be the same for each Fortran support method.

Section 17.1.2 through 17.1.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full interface specification together with the specific procedure names and implications for the profiling interface. Section 17.1.6 the implementation of the MPI routines for different versions of the Fortran standard. Section 17.1.7 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 17.1.8 and Section 17.1.9 describe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is needed for one of the methods to prevent register optimization problems. A set of functions provides additional support for Fortran intrinsic numeric types, including parameterized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. In the context of MPI, parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters. Sections 17.1.10 through 17.1.19 give an overview and details on known problems when using Fortran together with MPI; Section 17.1.20 compares the Fortran problems with those in C.

17.1.2 Fortran Support Through the `mpi_f08` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi_f08` that can be used in a Fortran program. Section 17.1.6 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first

of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not `TYPE(*)`, with the following exception:

  Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the `mpi` module, see Section 17.1.3.

  Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from `mpif.h` or the `mpi` module to the `mpi_f08` module. (*End of advice to users.*)
- Define the derived type `MPI_Status`, and define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators `.EQ.` and `.NE.` to allow the comparison of these MPI handles with `.EQ.`, `.NE.`, `==` and `/=`.
- Use the `ASYNCHRONOUS` attribute to protect the buffers of nonblocking operations, and set the `LOGICAL` compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to `.TRUE.` if the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TS 29113). See Section 17.1.6 for older compiler versions.
- Set the `LOGICAL` compile-time constant MPI_SUBARRAYS_SUPPORTED to `.TRUE.` and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., `TYPE(*), DIMENSION(..)` in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

  Rationale. In all blocking routines, i.e., if the choice-buffer is not declared as `ASYNCHRONOUS`, the TS 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)
- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 17.1.6 for details.
- Declare each argument with an `INTENT` of `IN`, `OUT`, or `INOUT` as defined in this standard.

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

Advice to users. If a dummy argument is declared with `INTENT(OUT)`, then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [2], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of `INTENT(OUT)` the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 17.1.3. (*End of advice to users.*)

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

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

The MPI Fortran bindings in the `mpi_f08` module are designed based on the Fortran 2008 standard [2] together with the Technical Specification "TS 29113 Further Interoperability with C" [3] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [3], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the `mpi_f08` module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and

with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

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

17.1.3 Fortran Support Through the mpi Module

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

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section Section 17.1.6 on page 13 for more details.
- Define all MPI handles as type INTEGER.
- Define the derived type MPI_Status and all named handle types that are used in the mpi_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.

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

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces **and** the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

Advice to users. For an MPI implementation that fully supports nonblocking calls with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 17.1.12 for more details. (*End of advice to users.*)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [1], if the compiler supports this TS 29113 language feature. See Section 17.1.6 for further details.

- Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. When MPI_SUBARRAYS_SUPPORTED is defined as .TRUE., non-contiguous sub-arrays can be used as buffers in nonblocking routines.

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 17.1.6 for details.

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

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

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran INTENT. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such as MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran, but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

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

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

17.1.4 Fortran Support Through the mpif.h Include File

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

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

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as INTEGER.
- Be valid and equivalent for both fixed and free source form.

For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted).

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

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

  - Most mpif.h implementations do not include compile-time argument checking.
  - Therefore, many bugs in MPI applications remain undetected at compile-time, such as:
    - Missing ierror as last argument in most Fortran bindings.

  - Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.
  - Incorrect argument positions; e.g., interchanging the count and datatype arguments.
  - Passing incorrect MPI handles; e.g., passing a datatype instead of a communicator.
- The migration from `mpif.h` to the `mpi` module should be relatively straightforward (i.e., substituting `include 'mpif.h'` after an `implicit` statement by `use mpi` before that `implicit` statement) as long as the application syntax is correct.
- Migrating portable and correctly written applications to the `mpi` module is not expected to be difficult. No compile or runtime problems should occur because an `mpif.h` include file was always allowed to provide explicit Fortran interfaces.

(*End of advice to users.*)

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

17.1.5 Interface Specifications, Procedure Names, and the Profiling Interface

The Fortran interface specification of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attributes. The Fortran standard allows a given Fortran interface to be implemented with several methods, e.g., within or outside of a module, with or without `BIND(C)`, or the buffers with or without TS 29113. Such implementation decisions imply different binary interfaces and different specific procedure names. The requirements for several implementation schemes together with the rules for the specific procedure names and its implications for the profiling interface are specified within this section, but not the implementation details.

Rationale. This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:

- Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
- Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from `.FALSE.` to `.TRUE.`.
- The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
- No performance drawbacks.
- Consistency between all three Fortran support methods.
- Consistent with Fortran 2008 + TS 29113.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was

not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (*End of rationale.*)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 17.1 on page 9. Case is not significant in the names.

No.	Specific procedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex **??**, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !$PRAGMA IGNORE_TKR, which provides a call-by-reference argument without type, kind, and dimension checking.
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex **??**, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION(..).
2A	MPI_ISEND	Fortran interface and arguments, as in Annex **??**, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like !$PRAGMA IGNORE_TKR, which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex **??**, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION(..).

Table 17.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

Note that for the deprecated routines in Section 15.1 on page 597, which are reported only in Annex **??**, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi_f08 module.

To set MPI_SUBARRAYS_SUPPORTED to .TRUE. within a Fortran support method, it is required that all non-blocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the mpi_f08 module, and with MPI_XXXX_FTS in the mpi module and the mpif.h include file.

The `mpi` and `mpi_f08` modules and the `mpif.h` include file will each correspond to exactly one implementation scheme from Table 17.1 on page 9. However, the MPI library may contain multiple implementation schemes from Table 17.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

Rationale. After a compiler provides the facilities from TS 29113, i.e., `TYPE(*), DIMENSION(..)`, it is possible to change the bindings within a Fortran support method to support subarrays without recompiling the complete application provided that the previous interfaces with their specific procedure names are still included in the library. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own specific procedure names and all interfaces use the same constants (except the value of MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type definitions. After a compiler also ensures that buffer arguments of nonblocking MPI operations can be protected through the `ASYNCHRONOUS` attribute, and the procedure declarations in the `mpi_f08` and `mpi` module and the `mpif.h` include file declare choice buffers with the `ASYNCHRONOUS` attribute, then the value of MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to `.TRUE.` in the module definition and include file. (*End of rationale.*)

Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is — and what is not — supported. (*End of advice to users.*)

Within the `mpi_f08` and `mpi` modules and `mpif.h`, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_Isend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

A user-written or middleware profiling routine should provide the same specific Fortran procedure names and calling conventions, and therefore can interpose itself as the MPI library routine. The profiling routine can internally call the matching PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should invoke the corresponding PMPI routine using the same Fortran support method as used in the calling application program, because the C, `mpi_f08` and `mpi` callback prototypes are different or the meaning of the choice buffer or attribute_val arguments are different.

Advice to users. Although for each support method and MPI routine (e.g., MPI_ISEND in `mpi_f08`), multiple routines may need to be provided to intercept the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), each profiling routine itself uses only one support method (e.g., `mpi_f08`) and calls the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_Isend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the `mpi_f08` module occupy one Fortran numerical storage unit (same as an `INTEGER` handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as `OPTIONAL`,
- the internal argument passing mechanism for `ASYNCHRONOUS` and non-`ASYNCHRONOUS` arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method `mpif.h`, compile-time argument checking can be also implemented for all routines. For `mpif.h`, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that `mpif.h` must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

```
INTERFACE  PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 70 characters. In principle, continuation lines would be possible in `mpif.h` (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the `mpi_f08` module.

For example, the interface specifications together with the specific procedure names can be implemented with

```
MODULE mpi_f08
  TYPE, BIND(C) :: MPI_Comm
    INTEGER :: MPI_VAL
  END TYPE MPI_Comm
  ...
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
      IMPORT :: MPI_Comm
      TYPE(MPI_Comm),      INTENT(IN)  :: comm
      INTEGER,             INTENT(OUT) :: rank
      INTEGER, OPTIONAL,   INTENT(OUT) :: ierror
    END SUBROUTINE
  END INTERFACE
END MODULE mpi_f08

MODULE mpi
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
      INTEGER, INTENT(IN)  :: comm   ! The INTENT may be added although
      INTEGER, INTENT(OUT) :: rank   ! it is not defined in the
      INTEGER, INTENT(OUT) :: ierror ! official routine definition.
    END SUBROUTINE
  END INTERFACE
END MODULE mpi
```

And if interfaces are provided in `mpif.h`, they might look like this (outside of any module and in fixed source format):

```
!2345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
      INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
       SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
        INTEGER, INTENT(IN)  :: comm   ! The argument names may be
        INTEGER, INTENT(OUT) :: rank   ! shortened so that the
        INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
       END SUBROUTINE                  ! maximum of 72 characters.
      END INTERFACE
```

(*End of advice to implementors.*)

Advice to users. The following is an example of how a user-written or middleware profiling routine can be implemented:

```
SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
  USE :: mpi_f08, my_noname => MPI_Isend_f08ts
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER,            INTENT(IN)       :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN)       :: datatype
  TYPE(MPI_Comm),     INTENT(IN)       :: comm
  TYPE(MPI_Request),  INTENT(OUT)      :: request
  INTEGER, OPTIONAL,  INTENT(OUT)      :: ierror
    ! ... some code for the begin of profiling
  call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
    ! ... some code for the end of profiling
END SUBROUTINE MPI_Isend_f08ts
```

Note that this routine is used to intercept the existing specific procedure name MPI_Isend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_Isend. The `USE` of the `mpi_f08` module is needed for definitions of handle types and the interface for PMPI_Isend. However, this module also contains an interface definition for the specific procedure name MPI_Isend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the `USE` here specifically excludes the interface from the module by renaming the unused routine name in the `mpi_f08` module into "`my_noname`" in the scope of this routine. (*End of advice to users.*)

Advice to users. The PMPI interface allows intercepting MPI routines. For example, an additional MPI_ISEND profiling wrapper can be provided that is called by the application and internally calls PMPI_ISEND. There are two typical use cases: a profiling layer that is developed independently from the application and the MPI library, and profiling routines that are part of the application and have access to the application data. With MPI-3.0, new Fortran interfaces and implementation schemes were introduced that have several implications on how Fortran MPI routines are internally implemented and optimized. For profiling layers, these schemes imply that several internal interfaces with different specific procedure names may need to be intercepted, as shown in the example code above. Therefore, for wrapper routines that are part of a Fortran application, it may be more convenient to make the name shift within the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. (*End of advice to users.*)

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 17.1 on page 9 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

17.1.6 MPI for Different Fortran Standard Versions

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

- *For Fortran 77* with some extensions:
  - MPI identifiers may be up to 30 characters (31 with the profiling interface).
  - MPI identifiers may contain underscores after the first character.
  - An MPI subroutine with a choice argument may be called with different argument types.
  - Although not required by the MPI standard, the `INCLUDE` statement should be available for including `mpif.h` into the user application source code.

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

- *For Fortran 90:*
  The major additional features that are needed from Fortran 90 are:

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

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

  - MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
  - S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
  - In this preliminary interface of S1, the following changes are necessary:
    * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !$PRAGMA IGNORE_TKR.
    * The ASYNCHRONOUS attribute is omitted.
    * PROCEDURE(...) callback declarations are substituted by EXTERNAL.
  - The specific procedure names are specified in Section 17.1.5.
  - Due to the rules specified in Section 17.1.5, choice buffer declarations should be implemented only with non-standardized extensions like !$PRAGMA IGNORE_TKR (as long as F2008+TS 29113 is not available).

    In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 17.1.11). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 17.1.15.

    Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the mpi_f08 module if such an extension is not available.
  - The ASYNCHRONOUS attribute can **not** be used in applications to protect buffers in nonblocking MPI calls (S1–S3).
  - The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.

  - In S1 and S2, the definition of the handle types (e.g., `TYPE(MPI_Comm)` and the status type `TYPE(MPI_Status)` must be modified: The `SEQUENCE` attribute must be used instead of `BIND(C)` (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the `mpi_f08` module for Fortran 90 compilers. In this case, the `mpi_f08` handle types and all routines, constants and types related to `TYPE(MPI_Status)` (see Section 17.2.5) are also not available in the `mpi` module and `mpif.h`.

- *For Fortran 95:*
  The quality of the MPI interface and the restrictions are the same as with Fortran 90.

- *For Fortran 2003:*
  The major features that are needed from Fortran 2003 are:

  - Interoperability with C, i.e.,
    - `BIND(C)` derived types.
    - The `ISO_C_BINDING` intrinsic type `C_PTR` and routine `C_F_POINTER`.
  - The ability to define an `ABSTRACT INTERFACE` and to use it for `PROCEDURE` dummy arguments.
  - The ability to overload the operators `.EQ.` and `.NE.` to allow the comparison of derived types (used in MPI-3.0 for MPI handles).
  - The `ASYNCHRONOUS` attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

  With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

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

  MPI-3.0 can be implemented with the following restrictions:

  - MPI_SUBARRAYS_SUPPORTED equals `.FALSE.`.
  - For S1, only a preliminary implementation is possible. The following changes are necessary:
    - `TYPE(*), DIMENSION(..)` is substituted by non-standardized extensions like `!$PRAGMA IGNORE_TKR`.
  - The specific procedure names are specified in Section 17.1.5.
  - With S1, the `ASYNCHRONOUS` is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

- The ASYNCHRONOUS Fortran attribute can be used in applications to *try to* protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.
- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used only for Fortran types that are C compatible.
- The same restriction as for Fortran 90 applies if non-standardized extensions like !$PRAGMA IGNORE_TKR are not available.

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

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

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

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

  (*End of advice to implementors.*)

17.1.7 Requirements on Fortran Compilers

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

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

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 17.1.11 through 17.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [3] are available. This is required only for `mpi_f08`. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the `mpi_f08` module with MPI_SUBARRAYS_SUPPORTED set to `.FALSE.`.
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the `ASYNCHRONOUS` attribute. See Section 17.1.12 for more details.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== `.FALSE.`, they are passed with call by reference, and passed by descriptor in the case of `.TRUE.`.
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., `CHARACTER(LEN=*)` strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the `ISO_C_BINDING` intrinsic module procedure `C_F_POINTER` is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide `TYPE(*)` unless the `ASYNCHRONOUS` attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the `ASYNCHRONOUS` attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==`.FALSE.`. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the `mpi` module or the `mpif.h` include file. The rules are as follows:

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with `BIND(C)` Fortran interfaces (e.g., MPI_F_SYNC_REG on page 40 and Section 17.1.8, and DD on page 41) solve the problems described in Section 17.1.17.

- The problems with temporary data movement (described in detail in Section 17.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.

- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 17.1.19) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the `mpi_f08` and `mpi` modules and independently of whether `mpif.h` uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [3], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (*End of advice to implementors.*)

17.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 17.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf)

INOUT	buf	initial address of buffer (choice)

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

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

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

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

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is

explicitly allowed to implement this routine within the `mpi_f08` module according to the definition for the `mpi` module or `mpif.h` to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (*End of advice to implementors.*)

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

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

17.1.9 Additional Support for Fortran Numeric Intrinsic Types

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

Fortran (starting with Fortran 90) provides so-called `KIND`-parameterized types. These types are declared using an intrinsic type (one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, and `CHARACTER`) with an optional integer `KIND` parameter that selects from among one or more variants. The specific meaning of different `KIND` values themselves are implementation dependent and not specified by the language. Fortran provides the `KIND` selection functions `selected_real_kind` for `REAL` and `COMPLEX` types, and `selected_int_kind` for `INTEGER` types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare `KIND`-parameterized `REAL`, `COMPLEX`, and `INTEGER` variables in Fortran. This scheme is backward compatible with Fortran 77. `REAL` and `INTEGER` Fortran variables have a default `KIND` if none is specified. Fortran `DOUBLE PRECISION` variables are of intrinsic type `REAL` with a non-default `KIND`. The following two declarations are equivalent:

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

MPI provides two orthogonal methods for handling communication buffers of numeric intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default `KIND` or using `KIND` parameters obtained with the `selected_int_kind` or `selected_real_kind` functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method (see "Support for size-specific MPI Datatypes" on page 23) gives the user complete control over communication by exposing machine representations.

Parameterized Datatypes with Specified Precision and Exponent Range

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

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

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

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

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

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

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

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

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

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. In communication, an MPI datatype A returned by MPI_TYPE_CREATE_F90_REAL matches a datatype B if and only if B was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for p and r or B is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 23.

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

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

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

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

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

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

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 23.

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

MPI_TYPE_CREATE_F90_INTEGER(r, newtype)

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

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

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

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

```
INTEGER R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a `INTEGER` variable of `KIND selected_int_kind(r)`. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 23.

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

Example:

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

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

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

1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_XXX routines.

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

(*End of advice to users.*)

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.6.2) or user-defined (Section 13.6.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.6.2.

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

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

For MPI_TYPE_CREATE_F90_REAL:

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

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.

For MPI_TYPE_CREATE_F90_INTEGER:

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

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

Support for Size-specific MPI Datatypes

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

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler,

MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and **n** is the length in bytes of the machine representation. This datatype locally matches all variables of type (**typeclass**, **n**) in Fortran. The list of names for such types includes:

```
MPI_REAL4
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16
```

One datatype is required for each representation supported by the Fortran compiler.

> *Rationale.* Particularly for the longer floating-point types, C and Fortran may use different representations. For example, a Fortran compiler may define a 16-byte REAL type with 33 decimal digits of precision while a C compiler may define a 16-byte long double type that implements an 80-bit (10 byte) extended precision floating point value. Both of these types are 16 bytes long, but they are not interoperable. Thus, these types are defined by Fortran, even though C may define types of the same length. (*End of rationale.*)

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

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

MPI_SIZEOF(x, size)

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

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

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

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

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

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

MPI_TYPE_MATCH_SIZE(typeclass, size, datatype)

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	datatype	datatype with correct type, size (handle)

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
```

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

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

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C, one can use the C function sizeof(), instead of MPI_SIZEOF. In addition, for variables of default kind the variable's size can be computed by a call to MPI_TYPE_GET_EXTENT, if the `typeclass` is known. It is erroneous to specify a size not supported by the compiler.

Rationale. This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{
  switch(typeclass) {
      case MPI_TYPECLASS_REAL: switch(size) {
        case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
```

```
            case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
            default: error(...);
          }
          case MPI_TYPECLASS_INTEGER: switch(size) {
             case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
             case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
             default: error(...);
          }
        ... etc. ...
      }

      return MPI_SUCCESS;
    }
```

(*End of advice to implementors.*)

Communication With Size-specific Types

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

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

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

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

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

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

if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',                &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,     &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif

call MPI_BARRIER(MPI_COMM_WORLD, ierror)

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

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

17.1.10 Problems With Fortran Bindings for MPI

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

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

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

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

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

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

4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the `ASYNCHRONOUS` attribute as specified in TS 29113.

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

6. The memory allocation routine MPI_ALLOC_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, `TYPE(C_PTR)` entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The `mpi_f08` module only supports `TYPE(C_PTR)` pointers.

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

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

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

Sections 17.1.11 through 17.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions

require special capabilities from the compilers. Major requirements are summarized in Section 17.1.7.

17.1.11 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, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 17.1.6). In C, the use of `void*` formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the `mpi_f08` module, the problem is avoided by declaring choice arguments with `TYPE(*), DIMENSION(..)`, i.e., as assumed-type and assumed-rank dummy arguments.

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

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

In practice, it is rare for compilers to do more than issue a warning. When using either the `mpi_f08` or `mpi` module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the `mpi_f08` or `mpi` module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays `INTEGER :: DIMS(*)` and `LOGICAL :: PERIODS(*)`.

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

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

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

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

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

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

- If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:

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

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

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

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

  Note that the above definition does not supercede restrictions about buffers used with non-blocking operations (e.g., those specified in Section 3.7.2).

  > *Advice to implementors.* The Fortran descriptor for TYPE(*), DIMENSION(..) arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the MPI_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)
  >
  > *Rationale.* If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*End of rationale.*)

- If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:

In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

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

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

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

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

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

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

According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array section is

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

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

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are

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

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

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

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

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

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

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

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

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

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

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

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

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

17.1.14 Special Constants

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

In Fortran, using special values for the constants (e.g., by defining them through `parameter` statements) is not possible because an implementation cannot distinguish these values from 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, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

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

17.1.15 Fortran Derived Types

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

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

```
type, :: mytype
   integer :: i
   real :: x
   double precision :: d
   logical :: l
end type mytype

type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent

call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%l, disp(4), ierr)

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

```
    disp(3) = disp(3) - base
    disp(4) = disp(4) - base

    blocklen(1) = 1
    blocklen(2) = 1
    blocklen(3) = 1
    blocklen(4) = 1

    type(1) = MPI_INTEGER
    type(2) = MPI_REAL
    type(3) = MPI_DOUBLE_PRECISION
    type(4) = MPI_LOGICAL

    call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
    call MPI_TYPE_COMMIT(newtype, ierr)

    call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
    ! or
    call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
    ! expects that base == address(foo%i) == address(foo)

    call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
    call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
    extent = disp(2) - disp(1)
    lb = 0
    call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
    call MPI_TYPE_COMMIT(newarrtype, ierr)

    call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
```

Using the derived type variable `foo` instead of its first basic type element `foo%i` may be impossible if the MPI library implements choice buffer arguments through overloading instead of using `TYPE(*), DIMENSION(..)`, or through a non-standardized extension such as `!$PRAGMA IGNORE_TKR`; see Section 17.1.6.

To use a derived type in an array requires a correct extent of the datatype handle to take care of the alignment rules applied by the compiler. These alignment rules may imply that there are gaps between the components of a derived type, and also between the subsuquent elements of an array of a derived type. The extent of an interoperable derived type (i.e., defined with `BIND(C)`) and a `SEQUENCE` derived type with the same content may be different because C and Fortran may apply different alignment rules. As recommended in the advice to users in Section 4.1.6, one should add an additional fifth structure element with one numerical storage unit at the end of this structure to force in most cases that the array of structures is contiguous. Even with such an additional element, one should keep this resizing due to the special alignment rules that can be used by the compiler for structures, as also mentioned in this advice.

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the `BIND(C)` or the `SEQUENCE` attribute as choice buffer arguments; some additional constraints must be observed, e.g., no `ALLOCATABLE` or `POINTER`

type components may exist. In this case, the `base` address in the example must be changed to become the address of `foo` instead of `foo%i`, because the Fortran compiler may rearrange type components or add padding. Sending the structure `foo` should then also be performed by providing it (and not `foo%i`) as actual argument for `MPI_Send`.

17.1.16 Optimization Problems, an Overview

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

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

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

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

- Code movement and register optimization problems; see Section 17.1.17.
- Temporary data movement and temporary memory modifications; see Section 17.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 17.1.19.

Table 17.2 shows the only usage areas where these optimization problems may occur.

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

Table 17.2: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 38–42,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 17.1.7.

17.1.17 Problems with Code Movement and Register Optimization

Nonblocking Operations

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

Example 17.1 Fortran 90 register optimization — extreme.

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

Example 17.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_IRECV and the completion of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has returned, and may schedule the load of buf earlier than typed in the source. The compiler has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column.

Example 17.2 Similar example with MPI_ISEND

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

Due to valid compiler code movement optimizations in Example 17.2, the content of buf may already have been overwritten by the compiler when the content of buf is sent.

The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

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

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI_START and MPI_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI_START or MPI_STARTALL call, similar to the MPI_WAIT call as described in the Nonblocking Operations subsection in Section 17.1.17.

One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4.

MPI_BOTTOM and Combining Independent Variables in Datatypes

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

Example 17.3 shows what Fortran compilers are allowed to do.

In Example 17.3, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

In Example 17.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.

Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations

Example 17.3 Fortran 90 register optimization.

This source ... can be compiled as:

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

Example 17.4 Similar example with MPI_SEND

This source ... can be compiled as:

```
! buf contains val_old                    ! buf contains val_old
buf = val_new
call MPI_SEND(MPI_BOTTOM,1,type,...)       call MPI_SEND(...)
! with buf as a displacement in type      ! i.e. val_old is sent
                                           !
                                           ! buf=val_new  is moved to here
                                           ! and detected as dead code
                                           ! and therefore removed
                                           !
buf = val_overwrite                        buf = val_overwrite
```

with the send and receive buffers used in nonblocking operations, or in operations in which MPI_BOTTOM is used, or if datatype handles that combine several variables are used:

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

Each of these methods solves the problems of code movement and register optimization, but may incur various degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated compiler suite according to the requirements listed in Section 17.1.7. The performance impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables or the `ASYNCHRONOUS` attribute is expected to be low to medium, and that of using the

VOLATILE attribute is expected to be high or very high. Note that there is one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not solve code movement problems in MPI applications.

The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable exceptions listed in Section 17.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to .FALSE..

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

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

In Example 17.5 Case (a) on page 45, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to b occur after the communication has completed.

In Case (b), the read accesses to `b(1:100)` in the loop `i=2,99` are read accesses to a pending communication affector while input communication (i.e., the two `MPI_Irecv` calls) is pending. This is a contradiction to the rule that *for input communication, a pending communication affector shall not be referenced.* The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 17.9.

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

The problems with MPI_BOTTOM, as shown in Example 17.3 and Example 17.4, can also be solved by declaring the buffer `buf` with the `ASYNCHRONOUS` attribute.

In some MPI routines, a buffer dummy argument is defined as `ASYNCHRONOUS` to guarantee passing by reference, provided that the actual argument is also defined as `ASYNCHRONOUS`.

Calling MPI_F_SYNC_REG

The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides the MPI_F_SYNC_REG routine for this purpose; see Section 17.1.8.

- The problems illustrated by the Examples 17.1 and 17.2 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.

  Example 17.1
  can be solved with

```
call MPI_IRECV(buf,..req)


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

  Example 17.2
  can be solved with

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

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

- The problems illustrated by the Examples 17.3 and 17.4 can be solved with two additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/ MPI_SEND, and one directly after this communication operation.

  Example 17.3
  can be solved with

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

  Example 17.4
  can be solved with

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

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

- In the example in Section 11.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 17.1, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

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

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

- The temporary memory modification problem, i.e., Example 17.6, can **not** be solved with this method.

A User Defined Routine Instead of MPI_F_SYNC_REG

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

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

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

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

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to

such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 17.1.7.

Module Variables and COMMON Blocks

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

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

The (Poorly Performing) Fortran VOLATILE Attribute

The `VOLATILE` attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the `VOLATILE` attribute to enforce correct execution of MPI programs is discouraged.

The Fortran TARGET Attribute

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

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

17.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 17.5, Case (b) on page 45. Example 17.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 17.7, buf(100,100) from Example 17.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused

loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 17.8 shows a second possible optimization. The whole array is temporarily moved to local_buf.

When storing local_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of local_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

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

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 38 of Section 17.1.17, the `ASYNCHRONOUS` attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 17.9 and in Example 17.10.

Note also that the methods

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

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 17.6 and 17.7.

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

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the `ASYNCHRONOUS` attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 17.9 (which is a solution for the problem shown in Example 17.5 and in Example 17.10 (which is a solution for the problem shown in Example 17.8), the array is split into inner and halo part and both disjoint parts are passed to a subroutine `separated_sections`. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TS 29113 based MPI library.

17.1.19 Permanent Data Movement

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

- MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the `ASYNCHRONOUS` attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 17.1.7.

17.1.20 Comparison with C

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the `&` operator and later referencing the objects by indirection on the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while a nonblocking operation is pending.

Example 17.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.

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

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

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

Example 17.6 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,...)
```

Example 17.7 The compiler may substitute the nested loops through loop fusion.

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

Example 17.8 Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

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

Example 17.9 Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

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

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

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

Example 17.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.

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

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

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

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

17.2 Language Interoperability

17.2.1 Introduction

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

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

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

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

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

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

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

17.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran `CHARACTER` variables. However, we assume that a Fortran `INTEGER`, as well as a (sequence associated) Fortran array of `INTEGER`s, can be passed to a C program. We also assume that Fortran and C have address-sized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that `INTEGER(KIND=MPI_OFFSET_KIND)` can be passed from Fortran to C as MPI_Offset.

17.2.3 Initialization

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

Advice to users. Certain implementations use the (inout) `argc`, `argv` arguments of the C version of MPI_INIT in order to propagate values for `argc` and `argv` to all

executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

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

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

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

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

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

17.2.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

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

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.4.

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

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

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

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

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

```
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
```

```
MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Request MPI_Request_f2c(MPI_Fint request)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_File MPI_File_f2c(MPI_Fint file)

MPI_Fint MPI_File_c2f(MPI_File file)

MPI_Win MPI_Win_f2c(MPI_Fint win)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Info MPI_Info_f2c(MPI_Fint info)

MPI_Fint MPI_Info_c2f(MPI_Info info)

MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)

MPI_Message MPI_Message_f2c(MPI_Fint message)

MPI_Fint MPI_Message_c2f(MPI_Message message)
```

Example 17.11 The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
INTEGER :: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END

/* C wrapper */

void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)
{
   MPI_Datatype datatype;

   datatype = MPI_Type_f2c( *f_handle);
   *ierr = (MPI_Fint)MPI_Type_commit( &datatype);
   *f_handle = MPI_Type_c2f(datatype);
   return;
```

```
}
```

The same approach can be used for all other MPI functions. The call to MPI_XXX_f2c (resp. MPI_XXX_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

17.2.5 Status

The following two procedures are provided in C to convert from a Fortran (with the `mpi` module or `mpif.h`) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

```
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
```

If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type `MPI_Fint*`, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the `mpi` module or `mpif.h`. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

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

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

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

Advice to users. There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Figure 17.1. (*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 type `MPI_F08_status` can be used to pass a Fortran `TYPE(MPI_Status)` argument into a C routine. Figure 17.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

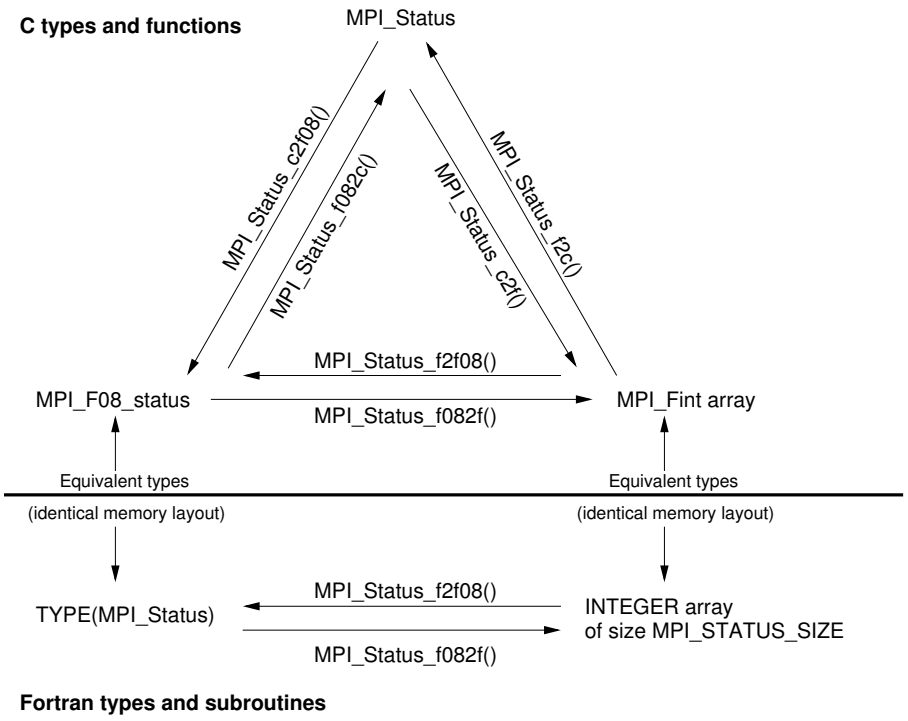

Figure 17.1: Status conversion routines

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

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

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

This C routine converts a C `MPI_Status` into a Fortran `mpi_f08` `TYPE(MPI_Status)`. Two global variables of type `MPI_F08_status*`, MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the `mpi_f08` module. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

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_F08_status *f08_status)

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

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

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

MPI_STATUS_F082F(f08_status, f_status)

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

```
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)

MPI_Status_f082f(f08_status, f_status, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  f08_status
    INTEGER, INTENT(OUT) ::  f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

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

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

17.2.6 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 Section 17.2.9).

Example 17.12

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

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

/* C code */

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

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

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

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

```
}
```

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

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

Callback Functions

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

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

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI_COMM_NULL_COPY_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice to users on page **??**. (*End of advice to users.*)

Error Handlers

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

Reduce Operations

All predefined named and unnamed datatypes as listed in Section 5.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

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

17.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

> *Advice to implementors.* This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 6.7 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 6.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer-valued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if

deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr.

Example 17.13

A. Setting an attribute value in C

```
int set_val = 3;
struct foo set_struct;

/* Set a value that is a pointer to an int */

MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
/* Set a value that is a pointer to a struct */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
/* Set an integer value */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
```

B. Reading the attribute value in C

```
int flag, *get_val;
struct foo *get_struct;

/* Upon successful return, get_val == &set_val
   (and therefore *get_val == 3) */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
/* Upon successful return, get_struct == &set_struct */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
/* Upon successful return, get_val == (void*) 17 */
/*        i.e., (MPI_Aint) get_val == 17 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

Example 17.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call

```
INTEGER IERR, VAL
VAL = 7
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
```

B. Reading the attribute value in C

```
int flag;
int *value;

/* Upon successful return, value points to internal MPI storage and
   *value == (int) 7 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, VALUE

! Upon successful return, VALUE == 7
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE

! Upon successful return, VALUE == 7 (sign extended)
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

Example 17.15 A. Setting an attribute value via a Fortran MPI-2 call

```
INTEGER IERR
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
VALUE1 = 42
VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40

CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
```

B. Reading the attribute value in C

```
int flag;
MPI_Aint *value1, *value2;

/* Upon successful return, value1 points to internal MPI storage and
   *value1 == 42 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
/* Upon successful return, value2 points to internal MPI storage and
   *value2 == 2^40 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40, or 0 if truncation
! needed (i.e., the least significant part of the attribute word)
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

The predefined MPI attributes can be integer valued or address-valued. Predefined integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran, MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,

MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to `(void *)`.

Rationale. The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as `INTEGER(KIND=MPI_ADDRESS_KIND)` attributes or (3) as `INTEGER` attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_XXX_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (*End of advice to implementors.*)

17.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a `COMMON` array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

17.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 17.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

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

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

(*End of advice to users.*)

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

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the *Datatypes* subsection in Section 17.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

17.2.10 Interlanguage Communication

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

Example 17.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
SUBROUTINE MYEXAMPLE()
USE mpi_f08
REAL :: R(5)
INTEGER :: IERR, MYRANK, AOBLEN(1)
TYPE(MPI_Datatype) :: TYPE, 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 MPI_TYPE_COMMIT(TYPE, IERR)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
   CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
   CALL C_ROUTINE(TYPE%MPI_VAL)
END IF
END SUBROUTINE
```

```
/* C code */

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

   type = MPI_Type_f2c(*fhandle);

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

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

Bibliography

[1] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from `http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90`. 17.1.3

[2] International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. *Information technology – Programming languages – Fortran – Part 1: Base language*, November 2010. 17.1.1, 17.1.2

[3] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. *TS on further interoperability with C*, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 17.1.1, 17.1.1, 17.1.2, 17.1.7

Index