

# Implementing the MPI-3.0 Fortran 2008 Binding

Junchao Zhang  
Argonne National Laboratory  
9700 S. Cass Avenue  
Argonne, IL 60439  
jczhang@anl.gov

Bill Long  
Cray Inc.  
380 Jackson Street  
St. Paul, MN 55101  
longb@cray.com

Kenneth Raffenetti  
Argonne National Laboratory  
9700 S. Cass Avenue  
Argonne, IL 60439  
raffenet@anl.gov

Pavan Balaji  
Argonne National Laboratory  
9700 S. Cass Avenue  
Argonne, IL 60439  
balaji@anl.gov

## ABSTRACT

MPI-3.0 defines a new interface for the Fortran 2008 language standard. This is the first Fortran support method for MPI that is consistent with the Fortran standard. This paper introduces our implementation of the Fortran 2008 binding in MPICH. Issues discussed include the binding framework, the implementation of wrapper functions, and the implementation of named constants. Our implementation is neat, efficient, and portable, in the sense that we limit the layers of wrappers, avoid Fortran-specific initialization, avoid unnecessary runtime overhead in wrappers, and rely only on standard Fortran and C.

## Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications—*Languages*

## General Terms

Languages, Standardization

## Keywords

MPI, Fortran, Fortran 2008, Language binding

## 1. INTRODUCTION

A preeminent programming language for high-performance computing (HPC), Fortran has been around for more than 50 years. It has been used to code everything from aircraft design, nuclear reactor modeling, seismic signal processing to weather forecasting. It is especially useful for numerical analysis and technical calculations. The Message Passing Interface (MPI) [1] is the dominant programming model for HPC. MPI provides scalable parallel programming abstractions for computers ranging from a desktop with a few cores to a supercomputer with millions of cores. Fortran and MPI are essential tools for HPC.

In its history, MPI has defined three Fortran bindings. The earliest one is the Fortran 77 (F77) binding, present

in the MPI-1.0 standard. F77 supports only implicit interfaces. An F77 compiler induces the interface of an external subroutine from the actual arguments passed at a call site and generates a sequence of calling code based on that. No compile-time argument number or type checking is performed. Thus, although MPI defines the standard interface of the library, an F77 compiler does not check against it to ensure that programmers supply correct arguments to the library. The lack of argument checking (type safety) in effect makes it easier to implement the MPI F77 binding. Since many MPI implementations are implemented in C and since Fortran passes arguments by address, one can implement the F77 binding by supplying a layer of wrapper functions written in C, which receive addresses of arguments in Fortran and call the backend C code. One issue in the implementation is to ensure that the C wrapper function names match the link names produced by the Fortran compiler. Usually the link name of an external symbol in Fortran is in lower case with one or two trailing underscores. This convention is compiler-dependent, however, and not specified in the Fortran standard.

Although the MPI F77 binding has been successful, the lack of type safety could lead to programming errors that are hard to debug. The Fortran 90 language standard (F90) introduced appealing features such as explicit interfaces, overloading, and modular programming. The MPI Forum defined a new interface in MPI-2.0, taking into account the new features of F90. This version contained all MPI functionality in a Fortran module named `mpi`. Ideally, all MPI routines should be declared in an interface block of the `mpi` module so that type safety can be enforced. But since F90 does not have a generic type like the `void*` in C, there is no standard way to declare the type for choice buffer in MPI. An inelegant workaround might be to use overloading and declare a specific procedure for every possible type and rank combination of actual choice buffers. This is not practical, however, and would lead to an interface explosion problem. An implementation would need to create more than 6 million specific procedures [4]. Even then, user-defined data types are not covered. Another approach is to use compiler-dependent directives such as `!$PRAGMA IGNORE_TKR` to tell the compiler to ignore type checking for choice buffer arguments. Again, it is not Fortran standard conforming, and the external symbol link name problem still exists.

Clearly, the binding problems lie in the poor interoperability of Fortran with C. Effort has been made to fix that. Starting from Fortran 2003, Fortran has provided standard mechanisms to interoperate with C. A recent Fortran techni-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

cal specification (TS 29113 [3]) further improves this feature. With this support, the MPI Forum defined the new Fortran 2008 (F08) binding interface in MPI-3.0, which is the only Fortran support method that is consistent with the Fortran standard (F08 + TS 29113 and later) and thus is highly recommended for all MPI applications.

In this paper, we discuss the F08 binding implementation in MPICH [2]. With a careful design and the support of interoperability of Fortran with C, our implementation is neat, efficient, and portable. The paper is organized as follows. Section 2 gives an overview of the new MPI F08 binding interface. Section 3 gives our implementation, and Section 4 talks about our test experience. Section 5 presents our conclusions and future work.

## 2. MPI F08 BINDING INTERFACE

The MPI F08 binding interface, defined in a module named `mpi_f08`, has several big improvements over its ancestors. The most important one is that choice buffers are now declared as *assumed-type*, *assumed-rank* dummy arguments, that is, of type `type(*)`, `dimension(..)`, which is defined in TS 29113 [3]. By definition, the actual argument for an assumed-type, assumed-rank dummy can be of any type and can be a scalar, an array, or an array section (i.e., subarray). The subarray can even be noncontiguous by using Fortran subscript triplets, for example, `a(2:10:2)`, where `a` is a 1-d array and the subarray contains `a(2)` to `a(10)` with a stride of 2.

The F08 binding further improves type safety in various aspects. MPI handles in the F77/F90 bindings all have type `integer`, making them indistinguishable to compilers. In the F08 binding, handles are defined as Fortran *bind(C)* derived types that consist of only one element, `integer :: MPI_VAL`. The internal handle value is identical to the Fortran integer value used in the F77/F90 bindings. Operators such as `==` and `/=` are overloaded to allow the comparison of these handles. An `MPL_Status` variable in the F77/F90 bindings is an integer array, for example, `integer :: status(MPI_STATUS_SIZE)`. In the F08 binding, it has a Fortran *bind(C)* derived type with three public integer components, `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR`, which are identical to the Fortran integer value used in the F77/F90 bindings. Additionally, the F08 binding defines interfaces for MPI user-defined callback functions. Dummy arguments that are a procedure (similar to function pointers in C) are declared by using Fortran's *procedure* keyword. Listing 1 shows some types defined in the F08 binding.

```

type, bind(C) :: MPI_Comm
  integer :: MPI_VAL
end type MPI_Comm

type, bind(C) :: MPI_Status
  integer :: MPI_SOURCE
  integer :: MPI_TAG
  integer :: MPI_ERROR
  ... ! Implementation-dependant private components
end type MPI_Status

```

Listing 1: Examples of MPI F08 types

The F08 binding also declares the `ierror` argument in all Fortran subroutines except user-defined and predefined callbacks have an *optional* attribute, so that a programmer can omit that argument. Additionally, choice buffers in nonblocking communications now have an *asynchronous* attribute. In Fortran 2003/2008, 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.

## 3. IMPLEMENT THE F08 BINDING IN MPICH

MPICH [2] is a high-performance and widely portable implementation of the MPI standard. MPICH and its derivatives form the most widely used implementations of MPI in the world. The current MPICH F77/F90 bindings are implemented in a set of C wrapper functions. But, as analyzed in Section 1, they have fundamental drawbacks. One drawback is that the F90 binding in MPICH does not support compiler-dependent directives for choice buffers; instead, it falls back on its F77 binding on subroutines with choice buffers. That means it does not perform argument-checking for those subroutines. With the F08 binding interface defined in MPI-3.0, we want to provide better Fortran support in MPICH. In this section we discuss our F08 binding design strategies and various implementation issues in MPICH. Our implementation is independent of MPICH's current F77/F90 bindings and targets a Fortran 2008 + TS 29113-capable compiler, which enables the most important parts of the new binding.

### 3.1 F08 Binding Framework

Like other MPI implementations, MPICH's backend is implemented in C. The process of F08 binding is to write wrapper functions that do necessary argument conversion between Fortran and C and invoke the backend C functions. Wrappers can be implemented in C or Fortran or both. For efficiency, the wrappers need to be thin and light. And when Fortran and C argument types are the same, we want to reduce the conversion overhead to zero. In the MPICH F08 binding, we implemented the wrappers in Fortran whenever possible. The main value is that Fortran intrinsically knows about both Fortran and C types, whereas C knows nothing about Fortran types. Hence, it is much safer and portable to use Fortran to write any code that involves a Fortran type. For most subroutines, one layer of Fortran wrappers is enough. For subroutines with choice buffers, however, we have to write another layer of wrappers in C to decode the C descriptor for the choice buffer before calling the backend MPI C libraries. Thus, we designed an F08 binding framework embodied by the following directory tree:

```

use_mpi_f08/
├── mpi_f08.F90
├── pmpi_f08.F90
├── mpi_f08_types.F90
├── mpi_f08_callbacks.F90
├── mpi_f08_compile_constants.F90
├── mpi_f08_link_constants.F90
├── wrappers_f/{start, send, recv, ...}_f08ts.F90
│   └── profiling/{pstart, psend, precv, ...}_f08ts.F90
├── mpi_c_interface_types.F90
├── mpi_c_interface_nobuf.F90
├── mpi_c_interface_cdesc.F90
├── mpi_c_interface_glue.F90
└── wrappers_c/{send, recv, ...}_cdesc.c

```

On one side, files with name `mpi_f08*.F90` are used to define interfaces required by the F08 binding. Among them, `mpi_f08_types.F90` defines MPI types as well as operators on them (see Listing 1 for examples); `mpi_f08.F90` defines

the `mpi_f08` module with an interface block containing interfaces for all MPI subroutines; and `pmpi_f08.F90` is `mpi_f08.F90`'s PMPI (i.e., profiling) version. Listing 2 shows example interfaces for `MPI_Start` and `MPI_Send`.

```
interface MPI_Start
  subroutine MPI_Start_f08(request, ierror)
    use :: mpi_f08_types, only : MPI_Request
    implicit none
    type(MPI_Request), intent(inout) :: request
    integer, optional, intent(out) :: ierror
  end subroutine MPI_Start_f08
end interface MPI_Start

interface MPI_Send
  subroutine MPI_Send_f08ts(buf, count, datatype, &
    dest, tag, comm, ierror)
    use :: mpi_f08_types, only : MPI_Datatype, MPI_Comm
    implicit none
    type(*), dimension(..), intent(in) :: buf
    integer, intent(in) :: count, dest, tag
    type(MPI_Datatype), intent(in) :: datatype
    type(MPI_Comm), intent(in) :: comm
    integer, optional, intent(out) :: ierror
  end subroutine MPI_Send_f08ts
end interface MPI_Send
```

Listing 2: Examples of MPI F08 routine interfaces

`mpi_f08_callbacks.F90` has an abstract interface block which contains interfaces for user-defined callbacks. It also implements the predefined callbacks (e.g., `MPI_COMM_NULL_COPY_FN`). `mpi_f08_compile_constants.F90` contains all constants that are known at compile time, including error classes, null handles, predefined MPI data types, predefined collective operations, predefined communicators, maximum sizes for strings, etc. `mpi_f08_link_constants.F90` declares the so called *named constants* (see more in Section 3.3). Directory `wrappers_f/` and `profiling/` contain Fortran files with each implementing an MPI subroutine declared in `{mpi, pmpi}_f08.F90`. They are actually wrapper files, in the sense that they wrap around the backend C functions.

To call the backend C functions from Fortran correctly, we need to know their interfaces, too. This information is given in `mpi_c_interface_*.F90`. Specifically, `mpi_c_interface_types.F90` defines types of MPI handles and `MPL_Status` in MPICH C binding. In MPICH C, except that `MPL_File` is a pointer type, all other MPI handles are a C integer. But we treat them all as integer and record their kind values. Listing 3 shows some MPI C data types from Fortran's viewpoint. Note that we access C types from the intrinsic `iso_c_binding` module. For example, `c_int` is the kind value of a C integer.

```
use, intrinsic :: iso_c_binding
integer, parameter :: c_Comm = c_int
integer, parameter :: c_Request = c_int
integer, parameter :: c_File = c_intptr_t

type :: c_Status
  integer(c_int) :: MPI_SOURCE
  integer(c_int) :: MPI_TAG
  integer(c_int) :: MPI_ERROR
  ...
end type c_Status
```

Listing 3: Examples of MPI C types

Similarly, `mpi_c_interface_{cdesc, nobuf}.F90` define interfaces for MPI C functions with or without choice buffers, respectively. See Listing 4 for examples. A few comments are appropriate here. First, we use `bind(C)` to directly specify the functions' link name, avoiding the underscore name mangling problem. Second, for routines without choice buffers, we want to call their C counterpart from Fortran wrappers

directly, so we declare functions bound to MPI C interfaces (e.g., `PMPI_Start`). We use the `PMPI_` version instead of the `MPI_` version to avoid double profiling in Fortran and C. Third, for routines with choice buffers, we do a second indirection in C and call C wrappers from Fortran. Therefore, we also declare interfaces for these C wrappers (e.g., `MPIR_Send_cdesc`). Fourth, for input arguments, we give them the `value` attribute so that the Fortran compiler will pass them by value instead of address, properly matching with what the C interfaces expect. To do language sensitive work, such as setting language tags or translating a string from Fortran to C, `mpi_c_interface_glue.F90` defines glue functions.

```
function MPIR_Start_c(request) &
  bind(C, name="PMPI_Start") result(ierror)
  use, intrinsic :: iso_c_binding, only : c_int
  use :: mpi_c_interface_types, only : c_Request
  implicit none
  integer(c_Request), intent(inout) :: request
  integer(c_int) :: ierror
end function MPIR_Start_c

function MPIR_Send_cdesc(buf, count, datatype, dest, tag, comm) &
  bind(C, name="MPIR_Send_cdesc") result(ierror)
  use, intrinsic :: iso_c_binding, only : c_int
  use :: mpi_c_interface_types, only : c_Datatype, c_Comm
  implicit none
  type(*), dimension(..), intent(in) :: buf
  integer(c_int), value, intent(in) :: count, dest, tag
  integer(c_Datatype), value, intent(in) :: datatype
  integer(c_Comm), value, intent(in) :: comm
  integer(c_int) :: ierror
end function MPIR_Send_cdesc
```

Listing 4: Examples of MPI C function interfaces

## 3.2 Fortran Wrappers

We now discuss the implementation of the Fortran wrappers. The main task is to convert between arguments specified by the Fortran interfaces and arguments specified by the C interfaces. If their types happen to be the same, there is no need to convert. Otherwise, we need to declare temporaries to accommodate the conversion. Depending on the *intent* value, we may need to convert Fortran to C (for *in* arguments), or convert C to Fortran (for *out* arguments), or do both (for *inout* arguments). When an argument is an array, if a temporary array is needed, we need to allocate it efficiently. We now discuss argument conversion for various Fortran types.

**INTEGER / MPI Handle** Since all MPI handles in Fortran have an integer value, this is the most common case. If the kind value of an integer argument is specified by the MPI Fortran interface, then in the C interface that argument usually has a paired C type of the same width and the same coding manner, which is guaranteed by the MPI standard. Examples include `MPI_Aint` for `integer(kind=MPI_ADDRESS_KIND)`, `MPI_Offset` for `integer(kind=MPI_OFFSET_KIND)`, and `MPI_Count` for `integer(kind=MPI_COUNT_KIND)`. Thus, in `mpi_c_interface_{nobuf, cdesc}.F90`, we declare the C argument with the same type (i.e., `integer(kind=MPI_ADDRESS_KIND)`) as its Fortran partner's and do no conversion.

If an argument's type in Fortran is *integer* or MPI handle and in C is *int*, then in Fortran's view that means we need to convert between `integer` and `integer(c_int)`. Most likely, Fortran's default integer has the same size as C `int`, and we do not need type conversion. But one can pass options like `-i8` to Fortran compilers to change the default. To get around this situation, we test the kind value of Fortran in-

teger against `c_int` in Fortran wrappers as shown in the following example.

```

subroutine MPI_Start_f08(request, ierror)
  use, intrinsic :: iso_c_binding, only : c_int
  use :: mpi_f08, only : MPI_Request
  use :: mpi_c_interface, only : c_Request, MPIR_Start_c

  type(MPI_Request), intent(inout) :: request
  integer, optional, intent(out) :: ierror
  integer(c_Request) :: request_c
  integer(c_int) :: ierror_c

  if (c_int == kind(0)) then
    ierror_c = MPIR_Start_c(request%MPI_VAL)
  else
    request_c = request%MPI_VAL
    ierror_c = MPIR_Start_c(request_c)
    request%MPI_VAL = request_c
  end if
  if (present(ierror)) ierror = ierror_c
end subroutine MPI_Start_f08

```

Listing 5: Test Fortran integer and C integer

Note that `kind(0)` returns the default kind value of a Fortran integer, which is known at compile time, so that the `if` branch and the `else` branch are also chosen at the compile time, in other words, the test should incur zero runtime overhead. Additionally, note that we use `%` to access a handle's `MPI_VAL` component and since `ierror` is optional, we use `if (present(ierror))` to test whether it is present.

To simplify the code, one may want to drop the `c_int == kind(0)` test and always do the type conversion as shown in the `else` part. When integer's kind value is indeed `c_int`, an optimizing compiler should easily get rid of the redundant copies through copy prorogation, incurring zero runtime overhead also. We do not choose this approach mainly because of integer array arguments: it is hard for a compiler to find that an array copy is redundant.

Regarding declaration of temporaries for array arguments, we use automatic arrays whenever possible, since they are allocated on stack, which is the most efficient memory allocation method. For many MPI routines with array arguments, either the arguments are already specified as an explicit-shape array, or although the arguments are an assumed-size array, their size is actually given by another argument, such that we can use automatic arrays in both cases. In some MPI routines, however, to get the size of an assumed-size dummy array, we need to call other routines. For example, in `MPI_allgatherv`, the length of two arguments `recvcounts` and `displs` is determined by the group size, so we need to call `MPI_Comm_size` to get that; whereas in `MPI_Neighbor_alltoallv`, the two arguments' length is determined by in-degree of this process, so we need to call `MPIR_Dist_graph_neighbors_count` instead. For this case, we use Fortran *allocatable* arrays for temporary arrays but allocate memory only when `kind(0)` is not equal to `c_int`. Thus, in the common case, they have no overhead.

As mentioned before, `MPI_File` is a pointer in MPICH C. We call `fh_c = MPI_File_f2c(fh%MPI_VAL)` to convert `fh`, a Fortran MPI handle of type `type(MPI_File)`, to `fh_c`, a C file handle of type `integer(c_File)`, and vice versa by `fh%MPI_VAL = MPI_File_c2f(fh_c)`.

Attention must be paid to integer arguments that are used to represent array indices, for example the `index/indices` argument in `MPI_{Wait, Test}_{any, some}`. Since C uses 0-based indices and Fortran uses 1-based indices, we need to

adjust their value accordingly.

**LOGICAL** MPI C uses integer to represent Booleans. For a Fortran dummy argument (say, `x`) of type `logical`, its C partner is `x_c` of type `integer(c_int)`. We convert `x` to `x_c` by `x_c = merge(1, 0, x)`, and vice versa by `x = (x_c /= 0)`. Here, `merge` is a Fortran intrinsic, and `x` and `x_c` can be a scalar or an array.

**CHARACTER** In C, strings are terminated by a NULL character, whereas in Fortran, strings are of fixed length, possibly with trailing blank characters. We need to convert between these two conventions. Generally, one more character is allocated to a C temporary string to accommodate the null character. For example, for an input variable-length string such as `character(len=*) :: string`, its C partner is `character(kind=c_char) :: string_c(len_trim(string)+1)`. We copy `string` to `string_c`, append a `C_NULL_CHAR`, then pass that to the backend C functions. For an output variable-length or fixed-length string such as `character(len=*)::string` or `character(len=MPI_MAX_OBJECT_NAME)::string`, its C partner is `character(kind=c_char)::string_c(len(string)+1)` or `character(kind=c_char)::string_c(MPI_MAX_OBJECT_NAME+1)`. When `string_c` is returned, we copy characters before the NULL characters to `string` and set the trailing characters to blank. Note that the maximal string length constants such as `MPI_MAX_OBJECT_NAME` should be one less in Fortran than their partner in C. Additionally, `MPI_Comm_spawn` has an argument of type `char**`. In C, a `char**` argument is an array of pointers to string. In Fortran, the argument is a 2-d character array. Similar things happen to an argument of type `char***` in `MPI_Comm_spawn_multiple`. Converting such arguments between Fortran and C is even more complicated. We skip the details here.

**PROCEDURE** User-defined callback function argument are declared with the *procedure* keyword in Fortran. Suppose we have an input argument `user_fn`. We use `c_funloc(user_fn)` to get a C function pointer of type `type(c_funptr)` and pass that to the backend C interface by value. In a multi-language environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPICH attaches a language tag or sets a language proxy to callback functions to make sure that such an invocation will use the calling convention of the language the function is bound to. In Fortran wrappers where a callback function is passed in, we set the tag/proxy so that the right calling sequence is generated when the callback function is invoked.

**MPL\_Status** In MPICH Fortran and C interfaces, `MPL_Status` is a derived data type containing only *integer* or *int* components. We represent them by `type(MPI_Status)` and `type(c_Status)` (see Listing 1, 3) in Fortran wrappers, respectively, and overload the assignment operator (`=`) to convert between them. If `kind(0) == c_int`, they are basically the same type. But unfortunately, we cannot mimic the coding style in List 5 and write code like the following since the compiler complains that `MPIR_Recv_cdesc` expects an argument of `type(c_Status)` but not `type(MPI_Status)`, even though they are in effect the same when `kind(0) == c_int`.

```

subroutine MPI_Recv_f08ts(buf, count, datatype, &
  source, tag, comm, status, ierror)
  use :: mpi_f08, only : MPI_Status
  use :: mpi_c_interface, only : c_Status

```

```

...
type(c_Status), target :: status_c
integer(c_int) :: ierror_c

if (c_int == kind(0)) then
  ierror_c = MPIR_Recv_cdesc(buf,...,status) ! Error
else
  ierror_c = MPIR_Recv_cdesc(buf,...,status_c)
  status = status_c
end if
...
end subroutine MPI_Recv_f08ts

```

Listing 6: Pass MPI\_Status by object

Hence, we must always choose the `else` part for MPI\_Status arguments. This is unpleasant since it is hard for a compiler to optimize the copy away, leading to unnecessary runtime overhead especially for routines with an array of MPI\_Status argument (e.g., `MPI_Testall/Waitall`). Note that the code above is simplified since we have not yet handled the special status value `MPI_STATUS_IGNORE`. Handling that by the current “pass by object” approach would further complicate the code. Therefore, we choose another approach, detailed next, when we consider named constants.

### 3.3 Named Constants

MPI sometimes assigns a special meaning to a special value of a basic type argument. The value, with a name, does not change between MPI initialization and MPI completion and thus is called a *named constant*. Named constants are just special values of their types, but not special types. When they are used as an actual argument, argument checking still applies. In Fortran, the value of a few named constants could not be set at compile time since using special values for them through *parameter* statements is not possible because an implementation cannot distinguish these values from valid data. Thus, we call them link-time named constants.

Typically (as in MPICH F77/F90 bindings), these constants are implemented as predefined static variables in common blocks. Relying on the fact that the target compiler passes data by address, these variables are passed to a special C routine during Fortran MPLInit, and their addresses are recorded so that later in C wrappers we can check an argument’s address against the record to know whether a special value is passed in. However, this approach implies C wrappers are always needed for routines where named constants can be passed in. Also, when a C application calls a Fortran library with MPI calls, we need to add a test in the MPI calls to see whether the addresses of these named constants are already recorded. If they are not, we need to do some initialization. Since in the F08 binding we want to reduce the layers of wrappers and avoid the extra test due to Fortran-specific initialization, we do not follow this old approach.

One might want to bind named constants in Fortran with those in C. Doing so, however, is impossible. One reason is that named constants are a value in Fortran and an address in C. For example, `MPI_STATUS_IGNORE` in Fortran is of type `type(MPI_Status)`, whereas in C it is of type `MPI_Status*`. Another reason is that the C header file `mpi.h` in MPICH defines most named constants to some bad addresses. For example, `MPI_STATUS_IGNORE` is defined as `#define MPI_STATUS_IGNORE (MPI_Status *)1`. To maintain application binary interface (ABI) compatibility with previous MPICH releases, we do not want to change the bad values to some

good ones. But we can still take advantage of `bind(C)` to simplify the design. We now discuss our implementation strategies for link-time named constants.

`MPI_STATUS_IGNORE` is used by programmers to indicate they want to ignore the return MPI\_Status. In the `mpi_f08` module, we declare it as an MPI\_Status object with a *target* attribute, along with a `bind(C)` C pointer.

```

type(MPI_Status), target :: MPI_STATUS_IGNORE
type(c_ptr), bind(C, name="MPIR_C_MPI_STATUS_IGNORE") &
  :: MPIR_C_MPI_STATUS_IGNORE

```

As we can see, the pointer is bound to a C global variable, which is defined in C as follows.

```
MPI_Status *MPIR_C_MPI_STATUS_IGNORE;
```

During C MPLInit, `MPIR_C_MPI_STATUS_IGNORE` is initialized to have the same value as `MPI_STATUS_IGNORE` in `mpi.h`. Then we declare the `status` argument as `"type(c_ptr), value"` instead of `type(c_Status)` in C functions declared in `mpi_c_interface_{cdesc, nobuf}.F90`. For example, the `status` argument in `MPIR_Recv_cdesc` is declared as `"type(c_ptr), value, intent(in)"`. Then the Fortran wrapper for `MPI_Recv` is coded as follows.

```

subroutine MPI_Recv_f08ts(buf, count, datatype, source, &
  tag, comm, status, ierror)
  use, intrinsic :: iso_c_binding, only : c_loc, c_associated
  use :: mpi_f08, only : MPI_Status, assignment(=) &
    MPI_STATUS_IGNORE, MPIR_C_MPI_STATUS_IGNORE
  use :: mpi_c_interface, only : c_Status, MPIR_Recv_cdesc

  type(MPI_Status), target :: status
  type(c_Status), target :: status_c
  ...
  if (c_int == kind(0)) then
    if (c_associated(c_loc(status), c_loc(MPI_STATUS_IGNORE))) then
      ierror_c = MPIR_Recv_cdesc(buf, ..., MPIR_C_MPI_STATUS_IGNORE)
    else
      ierror_c = MPIR_Recv_cdesc(buf, ..., c_loc(status))
    end if
  else
    if (c_associated(c_loc(status), c_loc(MPI_STATUS_IGNORE))) then
      ierror_c = MPIR_Recv_cdesc(buf, ..., MPIR_C_MPI_STATUS_IGNORE)
    else
      ierror_c = MPIR_Recv_cdesc(buf, ..., c_loc(status_c))
      status = status_c
    end if
  end if
end subroutine MPI_Recv_f08ts

```

Listing 7: Pass MPI\_Status by pointer

In this code, `c_loc` returns the C address of an object, and `c_associated` tests whether two C addresses are the same. When we detect `MPI_STATUS_IGNORE` is passed in by Fortran, we forward the bad address to C. Otherwise, we just forward the address of `status`. All these are transparent to the back-end C. Note that when `c_int == kind(0)`, we do not convert between `type(MPI_Status)` and `type(c_Status)`, thus incurring no overhead. We also apply the “pass `bind(C)` pointer” trick to the following named constants, except `MPI_IN_PLACE` and `MPI_BOTTOM`.

`MPI_STATUSES_IGNORE` is used to ignore an array of MPI\_Statuses. In `mpi_f08` it is `"type(MPI_Status), dimension(1), target :: MPI_STATUSES_IGNORE"`.

`MPI_ERRCODES_IGNORE` is used to ignore the input error codes argument. In `mpi_f08` it is `"integer, dimension(1), target :: MPI_ERRCODES_IGNORE"`.

`MPI_ARGV(S)_NULL` are used to indicate the `argv` argument in `MPI_Comm_spawn` or the `argvs` argument in `MPI_`

`Comm_spawn_multiple` are empty. They are defined as `"character(len=1), dimension(1), target :: MPI_ARGV_NULL"` and `"character(len=1), dimension(1,1), target::MPI_ARGVS_NULL"`, respectively.

`MPLUNWEIGHTED`, `MPLWEIGHTS_EMPTY` are used for the weight array arguments in distributed graph creating routines to indicate either that all edges have the same weight or that the process has no in/out edges. In `mpi_f08`, they have the same type: `integer, dimension(1), target`.

`MPLIN_PLACE` is used in collectives to indicate that the output buffer is identical to the input buffer. Since it is used as an actual argument for an assumed-type, assumed-rank dummy argument, it can be of any type. Note that in C wrappers, we get a C descriptor (`cdesc`) instead of a pointer to the buffer (See details in Section 3.4). We decode the `cdesc` to get the base address of the buffer. Thus we need to know the Fortran `MPI_IN_PLACE`'s address on the C side. In `mpi_f08`, we declare `MPI_IN_PLACE` as a `bind(C)` variable with a C name `MPIR_F08_MPI_IN_PLACE`.

```
integer(c_int), bind(C, name="MPIR_F08_MPI_IN_PLACE") &  
:: MPI_IN_PLACE
```

In C, we declare `MPIR_F08_MPI_IN_PLACE` as a global integer variable. In C wrappers, we test the base address of a choice buffer against `&MPIR_F08_MPI_IN_PLACE` to know whether the Fortran `MPI_IN_PLACE` is passed in. If it is, we forward C `MPI_IN_PLACE` to the backend.

`MPLBOTTOM` indicates the start of the address range. In MPI calls, it can be used as a choice buffer argument together with an absolute MPI data type. MPICH C implements it as a NULL pointer, indicating that C address starts at 0. Since Fortran forbids passing a disassociated (e.g., NULL) pointer to a nonpointer dummy argument (e.g., an assumed-type, assumed-rank argument), we cannot use the same `MPI_BOTTOM` value in C from Fortran. Thus we followed the style of `MPLIN_PLACE`. In `mpi_f08`, `MPI_BOTTOM` is a `bind(C)` variable. In C wrappers, we test against its address. Note that MPI uses `MPI_Get_address` to get the address of a location in memory and that MPI requires `MPI_Get_address` to return the same value in all languages. Thus, for `MPI_Get_address`, we need just to bind it directly to C, ignoring the value of `MPI_BOTTOM` in Fortran.

### 3.4 C Wrappers

As mentioned, we need C wrappers for subroutines with assumed-type, assumed-rank dummy arguments. Such an argument is translated by the compiler into a C descriptor argument of type `CFI_cdesc_t*` in the corresponding `bind(C)` functions. `CFI_cdesc_t` is defined in `ISO_Fortran_binding.h`, which also provides interfaces to query information such as base address, type, rank, dimension, stride, and contiguousness of the actual choice buffer argument in Fortran. If the buffer is contiguous, then we can directly call the backend C function. Otherwise, it means the buffer is a strided subarray, which is currently not supported in our implementation (see Section 5)

## 4. TESTING

To test the correctness of our implementation, we ported a set of programs from the MPICH test suite that originally used the F90 bindings. Converting the tests required

changing the declaration and accesses of MPI objects such as communicators, statuses, datatypes, and RMA windows. Thanks to the stronger type safety in the F08 binding, a bug was discovered in the test suite that had gone undetected for years. A function call, `MPI_Abort(1, MPI_COMM_WORLD, ierror)`, incorrectly reordered the first `comm` and the second `errcode` argument in `MPI_Abort`. Previously, the compiler could not detect this kind of error, since both arguments are of type integer. With `comm` having a unique type `type(MPI_Comm)`, this error is revealed at compile time. Finding this error was exemplary of the enhanced usability of the F08 binding.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed the design and implementation of the MPI-3.0 Fortran 2008 binding in MPICH. Our design targets a Fortran 2008 + TS 29113-capable compiler. It is neat, efficient, and portable since we limit the layers of wrappers, avoid Fortran-specific initialization, avoid unnecessary runtime overhead in wrappers, and rely only on standard Fortran and C.

Being able to pass noncontiguous subarrays is a nice feature of the new MPI F08 binding interface. A potential use case is in stencil computing, where one need to exchange noncontiguous halos (e.g., border of a submatrix). To get performance, programmers usually need to create MPI derived data types in advance to describe such halos. With the new feature, such data types can be hidden in C wrappers in bindings and thus is convenient for programmers. But since these MPI calls are usually embedded in loops, creating and freeing MPI data types in every loop iteration will incur a very high overhead compared with that of a manually optimized code. An interesting question to ask is whether we can have the convenience of subarrays without losing performance. Noticing that the shape of halos is actually fixed in stencil codes, we wonder whether we can take advantage of this fact and do MPI data type caching in order to avoid the overhead from data type creation and freeing. Also, in our code, Fortran wrappers are outside of a module, thus eliminating some advantages of using modern Fortran (e.g., inlining). Can they be put within a module without breaking the MPI profiling interface? Answering those questions is our future work.

## 6. ACKNOWLEDGMENTS

This work used the Cray machine Edison at NERSC, which is supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC02-05CH11231. The work is supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

## 7. REFERENCES

- [1] MPI. <http://www.mpi-forum.org>.
- [2] MPICH. <http://www.mpich.org>.
- [3] ISO/IEC/SC22/WG5. TS 29113 further interoperability of Fortran with C, 2012.
- [4] C. E. Rasmussen and J. M. Squyres. A case for new MPI Fortran bindings. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 183–190. Springer, 2005.

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.  
(This page will be removed before publication and shall not count in the page count)