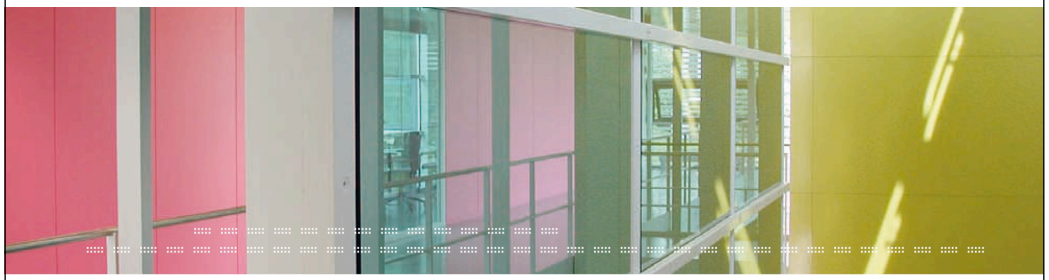
MPI-3.0 Fortran Interface

(Fortran Standardization Meeting WG5/J3, Munich, June 2011)

Rolf Rabenseifner, Jeff Squyres, Craig Rasmussen



Major concerns

- Are all actual buffer arguments that are possible with implicit interfaces still allowed with new explicit TYPE(*), DIMENSION(..)
  - If MPI_Xxxx is written in Fortran (and internally calls a C backend)
  - If MPI_Xxxx is written in C, i.e., defined with BIND(C)
    - Are user bothered with warnings like "REAL may be not interoperable"
    - Are all types allowed, i.e., also
      - CHARACTER strings,
      - assumed-size arrays,
      - derived types, BIND(C) derived types, SEQUENCE derived types,
      - etc.
- Does the *dope-vector* include enough information that a strided array can be copied into a contiguous scratch array without any further information?
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments and they are passed with call by reference.

Major concerns, continued

- Simply contiguous arrays and scalars must be passed to choice buffer dummy arguments with call by reference.
- MPI handles can be defined now with

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

  Are variables defined with such MPI_Comm still allowed at all locations where old-style INTEGER handles are allowed?
  - TYPE(MPI_Comm) :: mycomm or INTEGER :: mycomm
    COMMON /xxx/ mycomm
    (requires Fortran 2003 compilers; only old ifort 6.1 did not pass my Cu1 test)
  - TYPE :: xxx
    SEQUENCE
    TYPE(MPI_Comm) :: mycomm or INTEGER :: mycomm
    END TYPE xxx
    (My Cu3 test failed with all tested compilers)

Major concerns, continued

- OPTIONAL dummy arguments work with BIND(C)
- EXTERNAL and ABSTRACT INTERFACE dummy arguments work
  - If MPI_Xxxx is written in Fortran (and internally calls a C backend)
  - If MPI_Xxxx is written in C, i.e., defined with BIND(C)

Major concerns – the biggest problem area

- The big problem with
  - non-blocking MPI routines, and
  - actual arguments hidden through addresses in MPI datatype handles
- Examples
  - CALL MPI_Isend(xxx, …., rq)
…. (xxx may be still accessed but not modified)
CALL MPI_Wait(rq, …)
    - xxx may be accessed by an MPI progress thread or within MPI_Wait
  - CALL MPI_File_write_all_begin(xxx, …)
…. (xxx may be still accessed but not modified)
CALL MPI_File_write_all_end(xxx, …)
    - xxx may be accessed by an MPI progress thread doing asynchronous I/O
  - CALL MPI_Send(MPI_BOTTOM, 1, mydatatype, …)
    - mydatatype contains the address of xxx and xxx is accessed as if the application would have called MPI_Send(xxx, …)

Major concerns – the biggest problem area – continued

- Our goals
  - **Minimize the burden for the application programmer**
  - **Minimize additional needs for the Fortran Standard**
  - **Minimize drawbacks on compiler optimizations**
  - **Minimize the requirements that are needed that MPI + Fortran guarantees correct execution of portable applications**
  - **Be backward compatible with MPI-2.0**
- **Our solution**
  - **Take all we can get from TR 29113**
  - **Fixing further Fortran-MPI-incompatibilities**
    - At least with advices to users how to use Fortran in combination with MPI
    - Together with an MPI chapter on "**Requirements on Fortran Compilers**"

Major concerns – the biggest problem area – continued

- Three Optimization Problems:
  - Code movement and register optimization (was already discussed in MPI-2.0)
  - Temporary data movement (e.g., when using a GPU)
  - Permanent data movement (e.g., as part of a garbage collection)
- Four usage areas
  - Nonblocking MPI routines
  - One-sided MPI routines
  - Split-collective MPI routines
  - Usage of MPI_BOTTOM, or combining two variables through an MPI datatype

Optimization …	… may cause a problem when using:			
	Nonblocking	1-sided	Split-coll.	MPI_BOTTOM
Code movement and register optimization	YES	YES	no	YES
Temporary data movement	YES	YES	YES	no
Permanent data movement	YES	YES	YES	YES

Major concerns – the biggest problem area – continued

Code movement and register optimization (was already discussed in MPI-2.0)

Optimization …	… may cause a problem when using:			
	Nonblocking	1-sided	Split-coll.	MPI_BOTTOM
Code movement and register optimization	YES	YES	no	YES

Solutions: — **Overhead may be**

- new **TARGET attribute** — **low-medium**
- new **Calling MPI_F_SYNC_REG** — **low**
- **or a user defined routine (see DD in MPI-2.0)** — **low**
- **Using module variables or COMMON blocks** — **low-medium**
- **VOLATILE** — **high-huge**

An MPI library + Fortran compiler is only **MPI-3.0 compliant**
if this problem is solved when the application uses one of these methods!

Wrong solution:

- **ASYNCHRONOUS attribute** — **medium-high**

Major concerns – the biggest problem area – continued

Temporary data movement (e.g., when using a GPU)

new

Optimization ...	… may cause a problem when using:			
	Nonblocking	1-sided	Split-coll.	MPI_BOTTOM
Temporary data movement	YES	YES	YES	no

Overlapping communication and computation

Solutions:

- **None !!!!**

Current Fortran TR29113 proposals J3/11-183 - 192 and J3/11-202 - 205 about extending the ASYNCHRONOUS attribute do not remove this restriction → not better than current situation

Alternative (***this is a hard restriction for the users !!!***):

wow

- **Never use parts of a variable for communication / parallel I/O and another part for overlapping computation**

Wrong solution: **Overhead may be**

- **VOLATILE (*too expensive !!!*)** **high-huge**
- **ASYNCHRONOUS attribute (*does not work !!!*)** **medium-high**

Major concerns – the biggest problem area – continued

Permanent data movement (e.g., as part of a garbage collection)

new

Optimization ...	… may cause a problem when using:			
	Nonblocking	1-sided	Split-coll.	MPI_BOTTOM
Permanent data movement	YES	YES	YES	YES

Solutions:

- **None !!!!**

Okay?

Alternative (***this is a reasonable restriction for the implementors !!!***):

- **An MPI library + Fortran compiler is only MPI-3.0 compliant if this problem is solved!**

Wrong solution:

- **VOLATILE** (***too expensive !!!***)
- **ASYNCHRONOUS attribute** (***does not work !!!***)

Overhead may be

high-huge

medium-high

MPI_STATUS(ES)_IGNORE with function overloading

With USE mpi_f08, the user can freely choose

- CALL MPI_Recv(buf,cnt,datatype,src,tag,comm,status,ierror)
- CALL MPI_Recv(buf,cnt,datatype,src,tag,comm, ierror)
- CALL MPI_Recv(buf,cnt,datatype,src,tag,comm,status)
- CALL MPI_Recv(buf,cnt,datatype,src,tag,comm)

- Some routines are often in the critical path:
  - Function overloading is at compile-time

  → no conditional branch at run-time

  → Function overloading is more efficient
- Only 36 routines with status output argument
- Same API cannot be done with OPTIONAL status argument, i.e., with OPTIONAL status, users must write
  - CALL MPI_File_write(fh,buf,count,datatype, IERROR=ierror)

  instead of
  - CALL MPI_File_write(fh,buf,count,datatype, ierror)
- Also MPI_ERRCODES_IGNORE and MPI_UNWEIGHTED

Note that here, ierror may be needed, because in all I/O routines, ERORS_RETURN is the default!

Same decisions as in C++

MPI_ALLOC_MEM and Fortran

- How to use MPI_ALLOC_MEM together with C-Pointers in Fortran. (instead of non-standard Cray-Pointers)

new

Also available in the mpi module, not in mpif.h

```
SUBROUTINE MPI_Alloc_mem(size, info, baseptr, ierror)
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(C_PTR), INTENT(OUT) :: baseptr ! overloaded with the following…
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: baseptr ! …type
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
END
```

new

```
SUBROUTINE MPI_Free_mem(base, ierror)
  TYPE(*), DIEMSION(..) :: base
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
END
```

- New Example 8.1:

new

```
USE mpi_f08   !  or USE mpi   (not guaranteed with INCLUDE 'mpif.h')
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_PTR) :: p
REAL, DIMENSION(:,:), POINTER :: a                  ! no memory is allocated
INTEGER, DIMENSION(2) :: shape
INTEGER(KIND=MPI_ADDRESS_KIND) :: size
shape = (/100,100/)
size = 4 * shape(1) * shape(2)                      ! assuming 4 bytes per REAL
CALL MPI_Alloc_mem(size,MPI_INFO_NULL,p,ierr) ! memory is allocated and
CALL C_F_POINTER(p, a, shape)                       ! now accessible through a
A(3,5) = 2.71;
CALL MPI_Free_mem(a, ierr)                          ! memory is freed
```

- New interface that can be used together with ALLOCATABLE arrays

Not done

Thanks to Dieter an Mey, who gave me an example in Feb. 2004

Section 16.2.16 Requirements on Fortran Compilers

- The compliance to MPI-3.0 (and later) Fortran bindings is not only a property of the MPI library itself, but is always a property of an MPI library together with the Fortran compiler it is compiled for.
  - *Advice to users.* Many MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc). These scripts start the compiler probably together with special options to guarantee this compliance. *(End of advice to users.)*
- An MPI library is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 16.2.3 to 16.2.11 work correctly.

Summary on such requirements (slide 1) New

- Assumed-type and assumed-rank from Fortran 2008 TR 29113 is available;
  - Otherwise preliminary MPI-3.0 library with Fortran 2003 work-arround.
- Simply contiguous arrays and scalars must be passed to choice buffer dummy arguments with call by reference.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments and they are passed with call by reference. Derived type buffers
- The TARGET attribute solves code movement problems.
- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (as MPI_F_SYNC_REG and user-written DD) solve code movement problems.
- The problems with temporary data movement are solved as long as the application uses different sets of variables for the nonblocking communication and the computation when overlapping communication and computation.
- Problems caused by automatic and permanent data movement (e.g., within a garbage collection) are resolved without any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in calling MPI operations.

Rules about Correct-ness

Summary on such requirements (slide 2) New

- 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 handle and status types in mpi_f08 (i.e., sequence derived types with INTEGER elements) are (handle) or can be (status) identical to one numerical storage unit or a sequence of those. These types must be valid at every location where an INTEGER and a fixed-size array of INTEGERs (i.e., handle and status in the mpi module and mpif.h) is valid, especially also within SEQUENCE derived types defined by the application.
  - *Rationale.* This is not yet part of the draft N1845 of TR 29113 [36], but may be part of the final version of this TR 29113 [35].
- Further requirements apply if the MPI library internally uses BIND(C) routine interfaces.

Major rules about back-ward compa-tibility

Open questions – … for users okay?
… for the implementors okay? … and technically okay?

- Is the decision “BIND(C) derived types for handles and status” okay?
- It is not expected that our new handles can be used officially in SEQUENCE dervied types within the application data. Is this okay?
- Is the decision “explicit callback prototypes for buffer-free routines” okay?
- Is the decision “implicit callback prototypes for routines with buffers” okay?
- Is the “wording about derived type user buffers and MPI_Type_create_struct” okay?
- Are the “solutions about code movement” together with the “requirements” okay?
- Is the restrictive solution for “temporary data movement” okay?
- With “permanent data movement”, is it okay to put the burden on the implementors?
- Are there link-time optimizations that still can produce wrong execution?

Problem with MPI_ALLOC_MEM

- Application 1 – using the a standard Fortran pointer TYPE(C_PTR) with MPI-2.2
  - Calls MPI_ALLOC_MEM that has an implicit interface (maybe within **mpif.h** or the **mpi** module)
  - This user has ignored the Example 8.1 because it uses a non-standard pointer
- Application 2 – Using Cray-Pointer together with a Fortran 95
  - Calls MPI_ALLOC_MEM with an implicit interface
  - Or having a compiler that maps Cray-Pointer with INTEGER (KIND=MPI_ADDRESS_KIND)
- With Fortran mpif.h, only the INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR is required. With the mpi module, a second, overloaded subroutine is required if the Fortran compiler supports ISO_C_BINDING:

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER :: INFO, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
  TYPE(C_PTR) :: BASEPTR
```

Acknowledgement

- Thanks to the Fortran Standardization Committees WG5 and J3
  for their working together to solve the MPI-Fortran incompatibility problems.
  We really appreciate your hard work on these topics.