ISO/IEC JTC1/SC22/WG5 N1211
X3J3 / 96-128

DATE: 1996-07-29

ISO/IEC JTC 1/SC 22/WG 5
Programming languages, their environments and system software interfaces.
Working group Fortran

DOC TYPE: WG5 working document

TITLE: A Fortran 90 interface to MPI version 1.1

SOURCE: Michael Hennecke

PROJECT: 1.22.02.01

STATUS: —

ACTION ID: FYI

DUE DATE: —

DISTRIBUTION: WG5 and X3J3

MEDIUM: Open

NO. OF PAGES: 16

A Fortran 90 interface to MPI version 1.1

Michael Hennecke

Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany

`http://www.uni-karlsruhe.de/~Michael.Hennecke/`
`hennecke@rz.uni-karlsruhe.de`

MPI is the de-facto standard for message passing programming. In MPI version 1.1, the MPI Forum has established language bindings for ISO C and (an extended) Fortran 77, but not for Fortran 90. This report describes the design of a Fortran 90 MODULE implemented on top of an existing MPI library with Fortran 77 bindings. By using this module, Fortran 90 programmers have access not only to the contents of the MPI header file `mpif.h`, but also to a complete specification of the interfaces of all MPI procedures — including type and intent of procedure arguments and generic interfaces to MPI procedures with *choice* arguments. MPI's reliance on non-standard Fortran semantics is particularly troublesome with Fortran 90, these problems and restrictions are also discussed.

1 Introduction

The Message Passing Interface Standard[4] specifies an application programming interface to a set of library routines for message passing programming. It consists of 129 procedures specified in a language-independent form, and a number of defined constants to be used with these procedures. Language bindings are provided for ISO C[1]; there is also a binding for Fortran 77[2] augmented by a number of extensions and assumptions (most of which are summarized in section 2.5 of the MPI standard), but not for Fortran 90[3].

In principle, it should be possible to use a Fortran 77 based MPI implementation from Fortran 90 because Fortran 90 is a superset of Fortran 77. Since MPI documents the type and intent of procedure arguments, and Fortran 90 has the INTERFACE block mechanism to provide such explicit procedure interfaces in a calling program unit, it should also be possible to specify the (Fortran 77) interfaces of MPI procedures by Fortran 90 interface blocks in a MODULE. This would allow for much more type-checking in application programs calling MPI, thus considerably improving program development.

It is the aim of this report to show how such a Fortran 90 interface to MPI can be developed, and to point out the general limitations and problems with Fortran bindings. It is *not* the intent to provide a true Fortran 90 binding which may also benefit from OPTIONAL arguments, assumed-shape arrays, derived types, or other new features of Fortran 90. Fortran 90 binding to MPI is also a subject of the MPI-2 initiative[1] of the MPI Forum,[2] many of the problems discussed there are similar to those addressed in the current work.

The following section summarizes the problems caused by MPI syntax and semantics that are not conforming to the Fortran standards[2,3]. Section 3 shows the design of the proposed MODULE, using a few typical MPI procedures as examples. The complete source is available electronically.[3] Section 4 gives some guidelines for the installation and use of the proposed module, as well as some of the restrictions on a Fortran 90 program using this interface. Some possible extensions are sketched in section 5.

2 MPI and Fortran standard conformance

The Fortran 77 bindings defined in MPI deviate from the Fortran 77 standard in several points. This section discusses the mayor items, and outlines consequences for migration to Fortran 90.

Some minor MPI extensions to Fortran 77 are standardized in Fortran 90:

- MPI requires identifiers to be significant to thirty characters, and also allows underscores in identifiers.
- MPI recommends to use an INCLUDE file for MPI named constants.

Two main deviations from the Fortran 77 standard need more attention:

- A total of 35 MPI routines use *choice* arguments. These correspond to C's `void *`, their contents is the address of the data object designated by the actual argument. MPI Fortran 77 bindings specify this dummy argument as `<type> BUF(*)`. This design requires a program to be able to associate array actual arguments of all MPI-supported datatypes with this same dummy argument. This is a voilation of the Fortran standards.
  Additionally, MPI assumes that the implementation passes the address of the actual argument. This is not guaranteed by Fortran 77 (neither by Fortran 90), but virtually every `f77` compiler implements a call this way.

[1] URL: `http://parallel.nas.nasa.gov/MPI-2/mpi-bind/`
[2] URL: `http://www.erc.msstate.edu/mpi/`
[3] URL: `http://www.uni-karlsruhe.de/~Michael.Hennecke/Software/`

- MPI has a *non-blocking* communication mode. In this mode, a buffer existing in user space is passed as actual argument to a MPI procedure like MPI_IRECV. The procedure returns at some point in time, but the memory area designated by the actual argument still serves as a buffer to the "MPI system" — MPI will continue writing to that location, although MPI_IRECV has returned and there is no other MPI procedure executing (at least none seen by the Fortran compiler).

The problem of associating actual arguments of different type with one MPI procedure can be circumvented by introducing Fortran 90 generic interfaces for these calls. This is obvious for a complete Fortran 90 re-design of the API. Section 3.2 shows that by clever use of Fortran scoping rules, it is also possible to specify generic interfaces on top of an existing (non-conforming) EXTERNAL procedure with *choice* arguments, hiding the non-standard calls both from the user and the Fortran 90 compiler.

The assumption that assumed-size arrays are passed by their start address is still needed in Fortran 90, but should pose no problems for current compiler implementations.

Non-blocking communication is the most difficult problem in the above list. It is formally violating even the Fortran 77 standard, but severe problems can be expected for Fortran 90: At least if the Fortran 90 interface keeps the specification of BUF arguments as assumed-size arrays, there are many cases in which the Fortran 90 compiler needs to allocate a temporary, copy parts of the actual argument into that temporary (if the dummy argument has not INTENT(OUT)), pass the temporary to the MPI procedure, copy back from the temporary to the actual argument after the call returned (if the dummy argument has not INTENT(IN)), and finally deallocate the temporary. MPI then tries to access a memory region which no longer exists after the RETURN. Passing array sections with non-unit stride is a popular example of this behavior. There is no way to circumvent these problems in Fortran 90 with the current API of MPI. Section 4.3 contains some recommendations on Fortran 90 features which should be avoided when calling MPI procedures.

3 The MODULE design

Much of the code to create explicit interfaces for MPI procedures could be generated automatically: Since the MPI document is available as LaTeX source and contains an annex A with all Fortran 77 bindings, this annex can be used to generate INTERFACE blocks for all MPI procedures. These can be extended by adding suitable INTENT attributes, which are specified in the MPI main document.

What remains to be done is replacing the `<type> BUF(*)` type declarations for *choice* arguments by a generic interface and a set of specific procedures for each generic identifier. This is described in sections 3.2 and 3.3.

Since the resulting MODULEs should be usable for a wide range of implementations, some care must be taken to organize the sources in a way that is valid both for Fortran 90 free and fixed source form. This includes starting comments with an exclamation mark, restricting statements to columns 7–72,[4] and organizing continuation lines in a way acceptable in both formats (that is, a `&` character in column 73 or later of the continuing lines, and a `&` character in column 6 of all continued lines).

3.1 The module MPI1_HEADER

A small module, MPI1_HEADER, is defined which does nothing else than an INCLUDE of the MPI header file `mpif.h`. It is shown in figure 1. There are mainly two reasons to USE such a module instead of separate INCLUDE statements for `mpif.h`:

- A USE statement for a module guarantees consistent interpretation of the header file's contents, and also avoids the code replication inherent in the INCLUDE mechanism.
- Access from the module can be restricted to specific entities by ONLY clauses. An example is shown in the interface body of MPI_WAITANY in figure 4, which needs only the constant MPI_STATUS_SIZE.

Fig. 1. The module MPI1_HEADER

```
MODULE MPI1_HEADER
  IMPLICIT NONE
  INCLUDE 'mpif.h'
END MODULE MPI1_HEADER
```

All modules defined in the following sections access MPI1_HEADER by USE, rather than including the header `mpif.h` directly.

[4] For automatic generation of modules from a template source file as described in sections 3.2 and 3.3, it must also be avoided to extend past column 72 when text is substituted. This is the reason to start a continuation line immediately after the subroutine name in Fig. 2 and 3.

As discussed in section 2, one of MPI's violations of the Fortran standards is the use of *choice* arguments: data objects of different type must be associated to the same dummy argument of a procedure like MPI_SEND — typically as MPI buffers. This design is supported by the existence of a generic `void *` in C and the goal to keep the user interface simple: having different SEND procedures for all buffer datatyes is clearly undesirable. In Fortran 90, the solution to this problem is to overload a generic name MPI_SEND with specific SEND procedures for all buffer types required. This keeps the user interface and documentation simple,[5] but is completely standard conforming and allows for strong type checking.

For a native Fortran 90 binding, implementing this concept would be straightforward. Building an interface on top of an existing MPI library containing only one EXTERNAL procedure MPI_SEND is more complicated but possible: Specific procedures may be provided as MODULE PROCEDUREs with a clean interface for each buffer datatype, these may all reference the same EXTERNAL Fortran 77 procedure inside. Two important rules must be followed to put this concept to work:

- To ensure proper resolving of the reference to MPI_SEND inside the module procedures (e.g. MPI_SEND_REAL), these *must* contain an EXTERNAL MPI_SEND statement. Otherwise the reference would be to the generic identifier MPI_SEND, causing an infinite recursive reference.
- Placing a generic interface and *all* specific procedures to MPI_SEND into one module will not work: A quality Fortran 90 compiler is likely to detect that the specific module procedures all reference the same EXTERNAL, and do so in a non-conforming way. So seperately compiled modules which contain only *one* specific procedure must be defined, and afterwards USEd in the base module to make all overloadings accessible. At these USE statements, the compiler only checks the interface of the MODULE PROCEDUREs and thus cannot detect the EXTERNAL references.

Figure 2 shows the module template for overloading procedures which have *choice* arguments: it contains generic interfaces with exactly one specific procedure for type `type` in the specification-part, and the module procedure (which in turn calls the EXTERNAL procedure) in the module-subprogram-part. Replacing the string `type` by a valid datatype like `real` produces a module which overloads all procedures with a specific version for that type.[6]

[5] The work to specify all the specific procedures must still be done, but this is transparent to the user of the interface.

[6] It should be noted that it is dangerous to replace `type` with DOUBLE_PRECISION in identifiers, because this might increase the length of some

Fig. 2. The module template MPI1_type_V, showing only MPI_SEND and MPI_RECV (the full module has 35 generic interfaces, each with one specific procedure).

```
 MODULE MPI1_type_V
   IMPLICIT NONE ; PRIVATE

   PUBLIC :: MPI_SEND
   INTERFACE MPI_SEND
     MODULE PROCEDURE MPI_SEND_T
   END INTERFACE ! MPI_SEND

   PUBLIC :: MPI_RECV
   INTERFACE MPI_RECV
     MODULE PROCEDURE MPI_RECV_T
   END INTERFACE ! MPI_RECV

 CONTAINS

   SUBROUTINE MPI_SEND_T(                                          &
&      BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
     type,    INTENT(IN)  :: BUF(*)
     INTEGER, INTENT(IN)  :: COUNT, DATATYPE, DEST, TAG, COMM
     INTEGER, INTENT(OUT) :: IERROR
     EXTERNAL MPI_SEND
     CALL     MPI_SEND(                                          &
&      BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
   END SUBROUTINE MPI_SEND_T

   SUBROUTINE MPI_RECV_T(                                          &
&      BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
     USE MPI1_HEADER, ONLY: MPI_STATUS_SIZE
     type,    INTENT(OUT) :: BUF(*)
     INTEGER, INTENT(IN)  :: COUNT, DATATYPE, SOURCE, TAG, COMM
     INTEGER, INTENT(OUT) :: STATUS(MPI_STATUS_SIZE)
     INTEGER, INTENT(OUT) :: IERROR
     EXTERNAL MPI_RECV
     CALL     MPI_RECV(                                          &
&      BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
   END SUBROUTINE MPI_RECV_T
 END MODULE MPI1_type_V
```

One MPI1_TYPE_V module needs to be generated for each supported buffer datatype, these must be separately compiled to prevent a `f90` compiler from

MPI identifiers beyond 31 characters. Therefore, DBLE is used in identifiers for the DOUBLE PRECISION version.

detecting the non-standard EXTERNAL references in the module procedures for different `type`. Note that the names of all specific procedures are PRIVATE to each module, so they can all be suffixed with `_T` (rather than the full `_type` name) without name clashes. The resulting modules are USEd in the main module MPI1. which is described in section 3.4.

3.3 Scalar MPI buffer objects

MPI defines all buffers as assumed-size arrays, `<type> BUF(*)`. The problem of associating objects of different type with such *choice* buffers was solved in section 3.2 by introducing generic procedures with specific versions for each `<type>`. However, there is another subtle point where the MPI specification of the Fortran binding does not conform to the Fortran standards: It is illegal in Fortran to associate a scalar data object that is not an array element designator (like `A(1)` is) with an assumed-size array. Nevertheless, MPI uses the Fortran 77 procedures this way, see Example 4.15 on page 116 of [4] for an example. Most `f77` compilers do not complain about such "common practice", but `f90` compilers are likely to flag an error in this situation, especially when an explicit interface is visible.

To allow the possibility to pass scalars as MPI buffers (which is an extension of the original MPI specification), another set of specific procedures with scalar `BUF` arguments must be provided. In principle, two implementations are possible:

- The specific procedure takes a scalar `BUF` argument, copies the value of this argument to a temporary `TMP_BUF(1)`, and passes this array of size one to the EXTERNAL procedure These specific procedures may be included in the same MPI1_TYPE_V module shown above, since the call to the EXTERNAL passes an array as the original specific procedure does.
- The specific procedure passes the scalar `BUF` directly to the EXTERNAL procedure. This needs another set of seperately compiled modules (as described in section 3.2), because the fact that a scalar (and not a rank–1 array as in Figure 2) is passed to the EXTERNAL must be hidden from the compiler.

Although the first alternative seems to be easier, it cannot work with the *non-blocking* calls: the temporary array `TMP_BUF(1)` is automatic, and thus deallocated on return from the procedure. This leads to the problems described in section 4.3. The module template MPI1_TYPE_S for the second alternative is shown in figure 3. As in the preceding section, one version is needed for each supported buffer datatype.

These modules allow to pass scalars to BUF arguments. However, scalar buffers

Fig. 3. The module template MPI1_type_S, showing only MPI_ALLTOALL (the full module has 32 generic interfaces, each with one specific procedure).

```
  MODULE MPI1_type_S
    IMPLICIT NONE ; PRIVATE

    PUBLIC :: MPI_ALLTOALL
    INTERFACE MPI_ALLTOALL
      MODULE PROCEDURE MPI_ALLTOALL_T
    END INTERFACE ! MPI_ALLTOALL

  CONTAINS

    SUBROUTINE MPI_ALLTOALL_T(                                          &
&       SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,               &
&       RECVTYPE, COMM, IERROR)
      type, INTENT(IN)  :: SENDBUF
      INTEGER, INTENT(IN)  :: SENDCOUNT
      INTEGER, INTENT(IN)  :: SENDTYPE
      type, INTENT(OUT) :: RECVBUF
      INTEGER, INTENT(IN)  :: RECVCOUNT
      INTEGER, INTENT(IN)  :: RECVTYPE
      INTEGER, INTENT(IN)  :: COMM
      INTEGER, INTENT(OUT) :: IERROR
      EXTERNAL MPI_ALLTOALL
      CALL     MPI_ALLTOALL(                                            &
&       SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,               &
&       RECVTYPE, COMM, IERROR)
    END SUBROUTINE MPI_ALLTOALL_T
  END MODULE MPI1_type_S
```

for MPI_BUFFER_ATTACH and MPI_BUFFER_DETACH are not supported because this would not be useful anyway. MPI_ADDRESS is also not overloaded with a scalar BUF version to avoid problems with possible call-by-value implementations for scalar dummy arguments with INTENT(IN). For procedures with *two* BUF arguments like MPI_ALLTOALL, it is also not possible to associate a scalar with one buffer and a vector with the second one. This would require additional specific procedures, and will normally be unreasonable.

3.4 The main module MPI1

The user should be able to access the MPI header file, all the generic interfaces described in the preceding sections, and all interface blocks for procedures without *choice* arguments by a single USE statement. This is the aim

Fig. 4. The main module MPI1 (showing only two of 94 interface bodies).

```
  MODULE MPI1
    USE MPI1_HEADER

!   ... generic overloadings for <choice> argument procedures ...

    USE MPI1_integer_V    ;  USE MPI1_integer_S
    USE MPI1_real_V       ;  USE MPI1_real_S
    USE MPI1_dble_V       ;  USE MPI1_dble_S
    USE MPI1_complex_V    ;  USE MPI1_complex_S
    USE MPI1_logical_V    ;  USE MPI1_logical_S
    USE MPI1_character_V  ;  USE MPI1_character_S

    IMPLICIT NONE

!   ... A.9  Fortran Bindings for Point-to-Point Communication ...

    INTERFACE
      SUBROUTINE MPI_WAITANY(                                      &
 &        COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
        USE MPI1_HEADER, ONLY: MPI_STATUS_SIZE
        INTEGER, INTENT(IN)    :: COUNT
        INTEGER, INTENT(INOUT) :: ARRAY_OF_REQUESTS(*)
        INTEGER, INTENT(OUT)   :: INDEX
        INTEGER, INTENT(OUT)   :: STATUS(MPI_STATUS_SIZE)
        INTEGER, INTENT(OUT)   :: IERROR
      END SUBROUTINE MPI_WAITANY
    END INTERFACE

!   ... A.10  Fortran Bindings for Collective Communication ...
!   ... A.11  Fortran Bindings for Groups, Contexts, etc. ...
!   ... A.12  Fortran Bindings for Process Topologies ...
!   ... A.13  Fortran Bindings for Environmental Inquiry ...

    INTERFACE
      FUNCTION MPI_WTIME()
        DOUBLE PRECISION :: MPI_WTIME
      END FUNCTION MPI_WTIME
    END INTERFACE

!   ... A.14  Fortran Bindings for Profiling ...
  END MODULE MPI1
```

of the MPI1 module, which makes the MODULEs defined above accessible by suitable USE statements, and contains interface blocks for the remaining MPI procedures. It is shown in figure 4.

4 Installation and use

4.1 Installation

To install the set of modules described in section 3, they must be compiled with the Fortran 77 header file `mpif.h` visible.[7] This typically produces one object file and one module file (*file*`.mod`) for each MODULE source file.

The module files must be moved to a location which is in the search path for module files (often the INCLUDE path), they are required at compile-time. On UNIX systems, all objects can be combined into one archive library, e.g. `libmpi_f90.a`, which must be moved to a location in the library search path. This library is required at link time, on UNIX systems the `-lmpi_f90` option must *precede* the `-lmpi` option needed for the original Fortran 77 implementation of MPI.

4.2 Using the interface from Fortran 90 programs

Access to the Fortran 90 interface described in this report is provided by a USE MPI1 statement in each scoping unit requiring such access,[8] possibly including an ONLY clause to document which entities are actually needed.

A Fortran 90 program can then use all MPI calls and defined constants exactly the same way as a Fortran 77 program. The main advantage is that the `f90` compiler can now check MPI calls against the explicit interfaces specified in the module. As long as the program uses only Fortran 77 features, the only difference to the original Fortran 77 binding should be the overhead caused by calling through the additional layer of specific procedures for *choice* arguments. This effect should be small compared to the expected communication overhead.

It is also possible to use keyword arguments, with the argument names documented in the MPI standard. However, this is of limited use: OPTIONAL arguments, which are the main application of keyword arguments, are not

[7] The rules for source form given in the beginning of section 3 should also be followed for `mpif.h` — not all MPI implementations on systems where the Fortran compiler supports free source form include a `mpif.h` suitable for free source form...
[8] Note that an interface body has its own scope, and consequently has no access to the USEd modules in its surrounding scoping unit; a separate USE statement in the interface body for MPI_WAITANY in figure 4 is needed to access MPI_STATUS_SIZE.

supported by the current proposal. Entities of derived type cannot be used as arguments, e.g. as MPI buffers. This will be checked by the compiler, which knows the explicit interfaces. Specifying suitable components of derived type objects is possible, of course.

4.3 Features to avoid in Fortran 90 programs

When other Fortran 90 features not present in Fortran 77 are used, severe complications arise for all arguments which, by the MPI specification, are interpreted as the "address of a data object". This concept does not exisit in Fortran, and although virtually all implementations of Fortran 77 are compatible with this interpretation, Fortran 90 is not. Consider the following examples:

Example 4.1 Array sections:

```
USE MPI1, ONLY: MPI_REAL, MPI_SEND
REAL :: A(10)
! ...
CALL MPI_SEND( BUF=A(1:10:2),5,MPI_REAL,dest,tag,comm )
```

Because the BUF dummy argument to MPI_SEND is an assumed-size array and the callee might rely on sequence association, the caller must supply the actual argument for BUF as a dense memory region. Passing an array section with non-unit stride like `A(1:10:2)` invariably forces the compiler to allocate a temporary, copy the array section to/from that temporary, and pass this (dense) array to MPI_SEND. [9]

Example 4.2 POINTER arrays:

```
INTEGER       :: i
REAL, POINTER :: A(:)
REAL, TARGET  :: B(10)
! ...
A => B(2:9:i)
CALL MPI_SEND( A,SIZE(A),MPI_REAL,dest,tag,comm )
```

At the call to MPI_SEND, the POINTER actual argument `A` will be automatically de-referenced because the BUF dummy argument does not have the POINTER attribute. However, since the target of the pointer may again be an array section which is not dense in memory, the compiler may need to copy to a temporary as above. Since run-time costs to check if this is necessary may be large, some implementations may always do this copying — even if the target is dense.

[9] For contiguous array sections like `A(:,i)`, it is compiler-dependent if a copy is made or not.

Example 4.3 Array expressions:

```
REAL :: A(10), B(10)
! ...
CALL MPI_SEND( A+B,10,MPI_REAL,dest,tag,comm )
```

Fortran 90 allows general array expressions as actual arguments to a dummy argument that is an array and is not modified by the procedure. So sending the sum of `A` and `B` as shown above seems natural. Obviously, the compiler has to allocate temporary space to hold the result of the addition `A+B`, this temporary is passed to the procedure and deallocated on return (possibly later if the compiler can safely re-use the result `A+B` after the call).

Example 4.4 Assumed-shape actual arguments:

```
SUBROUTINE USER_SEND( A )
  REAL, DIMENSION(:) :: A
  ! ...
  CALL MPI_SEND( A,SIZE(A),MPI_REAL,dest,tag,comm )
END SUBROUTINE USER_SEND

REAL :: X(100)
! ...
CALL USER_SEND( X(::3) )
```

In this example, the array `A` is an assumed shape array. Some compilers will copy the array section `X(::3)` to a temporary which is then passed to USER_SEND. Now `A` is a dense array, and no problems need to arise when `A` is passed to MPI_SEND. However, a quality Fortran 90 compiler may be able to pass `X(::3)` without the need to copy to a dense temporary because an assumed shape array may also be implemented by a descriptor mechanism. In such cases, a temporary must be allocated for the call to MPI_SEND since BUF is *not* an assumed shape array.

The obvious drawback of the examples above (and similar situations like actual arguments that are general array expressions) is the performance degradation (and memory consumption) caused by allocating and deallocating the temporary array and copying data to and from that temporary. However, a more serious problem shows up with MPI's *non-blocking* communication calls: these will most likely cause not only slow but *erroneous* results in situations like those above. The reason is that MPI calls like MPI_ISEND and MPI_IRECV initiate asynchroneous read and write operations to memory locations given by their BUF argument, which persist even after the call has returned. Since the temporary arrays will be deallocated by the compiler immediately after the return from the corresponding MPI call, strange behavior is to be expected for non-blocking communications.

An actual argument used as a buffer in non-blocking calls *must* be a whole array data object, and shall not be an assumed shape dummy or have the POINTER attribute. If array sections are needed, the user may copy them to temporary arrays manually (again, POINTER should be avoided), and exercise great care that these do not go out of scope before the non-blocking operation has completed. For whole array buffers, the MPI datatype construction features can be used to specify the array sections to be actually transferred. Since the rules to avoid copying to temporaries may be quite complicated, the safest way to avoid erroneous behavior is not to use non-blocking communication from Fortran 90 at all. Obviuosly, abandoning these calls makes latency hiding quite difficult.

5 Possible extensions

Most extensions to the simple Fortran 90 interface described above will be complicated – support for Fortran 90 derived types or assumed shape arrays as MPI buffers are examples. However, some extensions are possible with only modest impact on the MPI environment (and existing MPI codes). this section outlines some of them.

5.1 OPTIONAL arguments

A number of arguments of MPI procedures might be declared as OPTIONAL, because many applications do not need them. Examples include:

- the communicator `COMM`, which in many applications is always the global communicator MPI_COMM_WORLD,
- the error indicator `IERROR`, because the default MPI behavior when errors are detected is MPI_ERRORS_ARE_FATAL,
- the `STATUS` field of receive calls, typically required only when wildcards like MPI_ANY_TAG are used.
- the `DATATYPE` handle of communication calls, this can be derived from the type of BUF (except for MPI_BYTE and MPI_PACKED).

Such features can be added without any portability problems for existing Fortran 77 programs: figure 5 shows how the COMM, STATUS and IERROR arguments of MPI_RECV can be made optional. If INTENT(IN) arguments are not given, the module procedure uses suitably defined defaults, INTENT(OUT) arguments which are provided by the EXTERNAL procedure are simply not passed up if the corresponding argument of the module procedure is not present.

Fig. 5. The specific MPI_RECV_T procedure with OPTIONAL arguments.

```
   SUBROUTINE MPI_RECV_T(                                              &
&      BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
     USE MPI1_HEADER, ONLY: MPI_STATUS_SIZE, MPI_COMM_WORLD
     type,    INTENT(OUT) :: BUF(*)
     INTEGER, INTENT(IN)  :: COUNT, DATATYPE, SOURCE, TAG
     INTEGER, INTENT(IN),  OPTIONAL :: COMM
     INTEGER, INTENT(OUT), OPTIONAL :: STATUS(MPI_STATUS_SIZE)
     INTEGER, INTENT(OUT), OPTIONAL :: IERROR
     EXTERNAL MPI_RECV
     INTEGER :: COMM_O, STATUS_O(MPI_STATUS_SIZE), IERROR_O

     IF ( PRESENT(COMM) ) THEN
       COMM_O=COMM
     ELSE
       COMM_O=MPI_COMM_WORLD
     END IF
     CALL MPI_RECV(                                                    &
&      BUF, COUNT, DATATYPE, SOURCE, TAG,                              &
&      COMM_O, STATUS_O, IERROR_O)
     IF ( PRESENT(STATUS) )  STATUS=STATUS_O
     IF ( PRESENT(IERROR) )  IERROR=IERROR_O
   END SUBROUTINE MPI_RECV_T
```

However, a MODULE PROCEDURE must be created also for the non-*choice* procedures (which have only an interface block in figure 4) because the code to handle OPTIONAL arguments cannot appear in an interface block. Since a module procedure cannot have the same name as the EXTERNAL procedure it calls, such an implementation would require the same hierarchy as figure 2, e.g. a generic interface named MPI_INIT containing a specific procedure MPI_INIT_O with OPTIONAL argument IERROR, which in turn calls the EXTERNAL procedure MPI_INIT. This is not very elegant, and for this reason was not included in the proposed Fortran 90 interface.

5.2 Derived types for MPI handles

It would be desirable to enable type-checking for MPI handles, too. In the Fortran 77 binding to MPI, all handles are of type INTEGER. The safest way to protect handles from being corrupted in Fortran 90 is to introduce a derived datatype with a PRIVATE component for each MPI handle type, and making only the type name (and the assignment and comparison operations required by the MPI document) available to the application program.

Implementing this is also possible in a module on top of an existing Fortran 77 implementation: the MODULE PROCEDUREs can pass the integer component of a derived type handle object to the corresponding EXTERNAL procedure. However, since application programs would need to be adapted to declare handles with the new derived types, this implementation would invalidate existing MPI programs.[10]

Another problem with such an implementation is that all the MPI-defined constants for a handle type must be re-declared to be of the corresponding derived type. This can be done with some tricky renaming of the original include file's constants, but this is clearly not the ideal solution. And for similar reasons as with OPTIONAL arguments, each EXTERNAL procedure needs to be called from a MODULE PROCEDURE, which is quite an effort to implement. Therefore, derived type handles were deferred to a native Fortran 90 binding which could implement them much easier.

6 Summary

This report has shown that designing a Fortran 90 interface to the existing MPI Fortran binding is possible without invalidating any existing Fortran application using MPI. A set of module files has been provided which can be used on top of an existing Fortran 77 implementation of MPI.

Some of the problems which must be expected when more advanced features of Fortran 90 are used have been discussed. Since MPI relies on some non-standard implementation details of the Fortran compiler, these features can be very dangerous to use. It must be noted that the probles exposed in section 4.3 are present even without the explicit interfaces provided by the Fortran 90 binding – without an explicit interface the Fortran 90 compiler assumes that all arrays arguments are assumed-size arrays.

Extensions to the proposed (minimal) binding are possible and desirable, but most of them are too difficult to build them on top of an existing Fortran 77 implementation, would invalidate existing programs, or both. It is likely that the MPI–2 initiative will standardize some of these more advanced features.

[10] A simple way to document different types of handles in the interfaces (without adding any extra functionality) would be to define KIND type parameters like MPI1_COMM in the MPI1_HEADER module, and declare handles like COMM of type INTEGER(MPI1_COMM) instead of simply INTEGER. Of course, all such KIND type parameters must specify the default INTEGER kind to be compatible with the original version.

References

[1] International Organization for Standardization. Information technology – Programming languages – C (ISO/IEC 9899:1990). Also ANSI X3.159-1989.

[2] International Organization for Standardization. Information technology – Programming languages – FORTRAN (ISO 1539:1980). Also ANSI X3.9-1978.

[3] International Organization for Standardization. Information technology – Programming languages – Fortran (ISO/IEC 1539:1991). Also ANSI X3.198-1992.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 1.1. June 12, 1995.
URL: `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps`.