Some Plebian Extensions to MPI

R. Clint Whaley *

November 6, 1997

Abstract

This paper discusses several extensions the author would like to see made to MPI. The purpose of these extensions is not to add significant new powers to MPI. On the contrary, these proposals may be viewed as filling in certain gaps in what has already been provided. These extensions are of particular interest to the programmer providing libraries based on MPI, as opposed to an application programmer.

<u>DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT</u>

Contents

*Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, `rwhaley@cs.utk.edu`

1 Interoperability of MPI's Interfaces

1.1 Motivation

At present, MPI [3] does not allow the user to portably mix the different interfaces to MPI. In particular, the C and Fortran77 interfaces are not compatible. We will discuss these two particular interfaces, but our comments here should apply more generally to the mixing of any MPI interfaces.

It is relatively common for library builders in particular to mix both C and Fortran77. C is often much easier to use than Fortran77 for system level programming. There are many reasons for this, among them being the relationship between UNIX and C, C's dynamic memory support, and finally the ease with which system level routines may be accessed from C.

It is then common to create a Fortran77 interface over the top of this library. In fact, MPICH[1, 2], a popular MPI implementation, takes this approach.

We therefore believe there are many programmers who mix these two languages. At present, libraries builders have no standard way to mix the C and Fortran77 MPI interfaces. It is not guaranteed that a message sent from one interface can be received by another, or that the interfaces may in any way interact. This leaves such a library builder with the option of making his library available in only one language, writing it separately in each language (if that is possible), or writing non-portable code which makes assumptions about the interoperability of C and Fortran77 which the MPI standard does not guarantee.

1.2 Proposal

We propose to make interface interoperability a part of the MPI standard. In particular, the ranks associated with a particular communicator should refer to the same process regardless of the interface, message tags should have the same meaning, etc. A message may be sent via one interface, and received by another, providing they use the same communicator. To allow this, there must be a standard way to translate between the communicator handles of the languages in question.

We therefore propose that MPI define a neutral communicator handle, which cannot be used for any purpose except communicator translation. To translate a communicator between two languages, the user shall first translate the interface-specific communicator to a neutral communicator, and from there to the second language. We propose that the type of this interface correspond to a Fortran77 INTEGER. Languages with advanced typing will use the type MPI_TRANS_COMM.

We therefore propose that all MPI interfaces have two routines. These routines translate from their communicator handle to a MPI_TRANS_COMM handle, and vice versa. Having a "neutral" handle has the benefit that each interface does not need to be able to directly translate to all other interfaces, thus greatly reducing the number of routines required for this operation.

The proposed routines for the Fortran77 interface are:

```
MPI_TRANS_TO_F77_COMM(transcomm, f77comm)
IN   transcomm
```

```
OUT  f77comm

MPI_TRANS_TO_F77_COMM(TRANSCOMM, F77COMM, IERROR)
   INTEGER TRANSCOMM, F77COMM, IERROR

MPI_TRANS_FROM_F77_COMM(f77comm, transcomm)
IN   f77comm
OUT  transcomm

MPI_TRANS_FROM_F77_COMM(F77COMM, TRANSCOMM, IERROR)
   INTEGER F77COMM, TRANSCOMM, IERROR
```

The proposed routines for the ANSI C interface are:

```
MPI_TRANS_TO_C_COMM(transcomm, ccomm)
IN   transcomm
OUT  ccomm

int MPI_Trans_to_c_comm(MPI_TRANS_COMM transcomm, MPI_Comm *Ccomm)


MPI_Trans_from_c_comm(ccomm, transcomm)
IN   ccomm
OUT  transcomm

int MPI_Trans_from_c_comm(MPI_Comm Ccomm, MPI_TRANS_COMM transcomm)
```

On most platforms, MPI_TRANS_COMM could be created in ANSI C by:

```
typedef int MPI_TRANS_COMM[1];
```

1.3 Comments

The above proposal assumes that MPI will not be changed, only extended. If a change is possible, it might be more natural to constrain all interfaces to having integer communication handles, who's meaning is the same across all interfaces, thus avoiding the necessity of performing any translation.

There are some advanced topics of language interoperability that MPI will be unable to enforce. The most important of these is data type. For instance, there is no guarantee that C's int is the same as Fortran77's INTEGER. MPI cannot enforce such things, nor should the user expect it to. Of MPI's data types, only `MPI_BYTE` and `MPI_PACKED` should be guaranteed to be the same across all languages. When the platform allows, the language correspondence will be as shown in table 1.

If these translations are not accurate on a particular platform, it will be up to the user to receive (probably as `MPI_PACKED`) and translate these data types.

Note that we do not propose a translation for all handles, e.g., there is no way to translate a data type (basic or derived) from one interface to another. This is also true of, for instance, of `MPI_Op`. These details will be left up to the programmer.

C	Fortran77
MPI_INT	MPI_INTEGER
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION

Table 1: Assumed, but not guaranteed, Fortran77 and C data type correspondence

The same is true of information that has been cached on a particular language's communicator. It is proposed that the only guaranteed way to access cached information is to access it from the same interface that it was stored in. This seems logical because there is no guarantee that an arbitrary data type in one language is even representable in another. Therefore, if the user is now in C, and wants information he has stored while in Fortran77, he should do a communicator translation, call a Fortran77 routine which retrieves the information, and pass the information back to C.

Note that since we are talking about communicator translation (not duplication or copying), the data is still there after translation, it's just that it cannot be standardly retrieved from any interface except the one it was originally cached in.

Since KEYVAL is an integer in both standard interfaces, it seems logical to constrain it to be an integer in all new interfaces. If this is done, there should be no problem with allowing the user to use KEYVAL to retrieve cached information from a different language than it was cached in. However, such retrieval is not guaranteed, and so the user would be required to make sure the information thus retrieved is handled in a meaningful way.

When the user calls MPI_ATTR_DELETE, the routine will always call the callback functions of the original definition. I.e., if the user calls MPI_ATTR_DELETE from Fortran77, but cached the information in C, the original C callback functions will still be called.

2 Refining MPI_INIT

2.1 Motivation

Presently, MPI_INIT is rather ill-defined. There are two issues here. First, since we hope to have language interoperability, we must define how the various interfaces interact with regards to initialization.

The second issue involves the C interface routine, MPI_Init, which takes pointers to `argv` and `argc` as parameters. There are a couple of problems with this. First, if your MPI calls are done by a library, they may not be made from `main()`. In this case, there will be no standard way to access `argv` and `argc`. This leaves the library builder with no way to hide the MPI calls, as the library user will be required to make the MPI_Init call explicitly in his main routine (or to pass `argv` and `argc` to the library in some way). The second problem is that it is not defined which processes actually have access to `argv` and `argc` when running an MPI program. This means that is not possible for the user standardly access `argv` and `argc`, since he does not know which processes posses the information. Thus he has the burden of passing these arguments to MPI_Init, with no assurance he can actually use them.

2.2 Proposal

We propose solutions to each of these problems. First, we propose that each language be segregated in its initialization. This means that no Fortran77 MPI calls can be made until MPI_INIT has been called, and no C MPI calls made until MPI_Init has been called. Both routines will have their versions of MPI_INITIALIZED for testing for this.

In practice, we foresee that the first call to MPI_INIT, regardless of the language, will actually have the most significance, as it will set up the basic environment used by all. However, as the number of language bindings grows, we find it plausible that having one initialization call set up all of them may become problematic. User's of multiple interfaces should not find it too onerous to call the appropriate test and initialization routines.

With this approach to interface initialization, it becomes more important than ever to ensure that MPI_Init can be called at any arbitrary point, where `argv` and `argc` may not be accessible. We therefore propose that if the user passes in `NULL` pointers in place of pointers to `argv` and `argc`, that the user will not be accessing `argv` and `argc`, but simply wishes to set up the C interface.

Finally, we believe it is important to state which processes will have access to `argv` and `argc`, assuming they are passed to MPI_Init. We propose that after a call to MPI_Init in which pointers to `argv` and `argc` are passed, that all processes will have access to these variables.

2.3 Comments

We can obviously not require that MPI distributes `argc` and `argv` to all processes, but we find little to recommend this idea. MPI_Init is called only once, so the time required to send `argv` and `argc` should certainly not be an issue. Further, if `argv` and `argc` were to be standardly accessible at all in such a scheme, we would have to assure the user that, for instance, MPI_COMM_WORLD's rank 0 always had `argv` and `argc`, and then the user could perform the send if necessary himself. This strikes us as rather an unwieldy way to handle the problem.

3 Non-blocking communication

3.1 Motivation

An MPI programmer wishing to provide a flexible library cannot always prevent user misuse. In our own libraries, we have seen situations where users perform operations that require a large amount of outstanding non-blocking communication. At present, MPI does not specify how many non-blocking communications may be safely used, nor is there a way to detect when internal resources have been overwhelmed.

As a matter of fact, an implementation could adhere to the standard even if only one non-blocking communication could be carried out at a time.

3.2 Proposal

We propose to add an error class to MPI. This error class is MPI_ERR_NORESOURCE. This error class will be guaranteed to be non-fatal: i.e., this error does not leave the MPI system in an undefined state as do other errors. For instance, if the user calls MPI_ISEND, and there are not enough internal system resources to issue the non-blocking send, MPI_ERR_NORESOURCE will be the error class of the returned error. The user may then wait for some previous non-blocking communication to complete, and try the send again.

3.3 Comments

The author sees that this error class will have obvious use in the routines MPI_ISEND, MPI_IBSEND, MPI_ISSEND, MPI_IRSEND, and MPI_IRECV. It may have use in other routines as well. Note that this error class should only be used to indicate a recoverable error has occurred.

4 Communicator Formation

4.1 Motivation

At present, all members of a defining group must call the communicator formation routine, even if the new group contains only a small subset of the defining group. For instance, if the MPI user defines a communicator based on MPI_COMM_WORLD, all processes in the system must call MPI_COMM_CREATE. We believe that this is an unusually restrictive requirement. For instance, if the user wishes to form communicators based on MPI_COMM_WORLD, he must not have any processes branch to specialized code until all required communicators have been formed.

This problem will be most readily apparent in libraries which work on top of other message passing interfaces as well as MPI. Here the user may pass to the library a list of processes which the library should use. When using message passing interfaces that do not possess grouping abilities, the library itself will provide groupings. In MPI, it would be convenient to get added safety by forming a private communicator which only the library can use.

Herein lies the problem. If the user passes in ranks based on MPI_COMM_WORLD, all processes will have to make the library call, whereas on another system, only those processes involved in the operation need make the library call. This will require substantial re-ordering of the user's code. In complex codes where groupings are done on the fly, in response to run-time information, this may be difficult to achieve.

4.2 Proposal

We propose to add the routine MPI_COMM_CREATE2. This routine functions just as MPI_COMM_CREATE, except that it blocks only on the processes defined to be in the group argument `group`. The proposed interface is therefore:

```
MPI_COMM_CREATE2(comm, group, newcomm)
IN  comm
IN  group
OUT newcomm

int MPI_Comm_create2(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

MPI_COMM_CREATE2(COMM, GROUP, NEWCOMM)
   INTEGER COMM, GROUP, NEWCOMM
```

4.3 Comments

The MPI standard offers the following three reasons that MPI_COMM_CREATE was defined to be blocking on the entire group of comm:

1. It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
2. It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
3. It permits implementations sometimes to avoid communication related to context creation.

Points 1 and 3 above should be addressed by warning the user that MPI_COMM_CREATE2 may be substantially slower than MPI_COMM_CREATE. The author feels that for most applications, MPI_COMM_CREATE is called rarely enough in respect to other MPI routines that its speed is not a real issue.

Point 2 depends on the how the communicator creation is implemented, and thus is not open to obvious rebuttal or discussion. However, it could be argued that this requirement makes some codes unworkable, and thus additional safety is not helpful.

References

[1] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. "Users' Guide to mpich, a Portable Implementation of MPI", 1995. Available via world wide web from URL = http://www.mcs.anl.gov/mpi/mpich/index.html.

[2] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. "Installation Guide to mpich, a Portable Implementation of MPI", 1995. Available via world wide web from URL = http://www.mcs.anl.gov/mpi/mpich/index.html.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard . *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994. Special issue on MPI. Also available electronically, the url is ftp://www.netlib.org/mpi/mpi-report.ps.