# Building Library Components That Can Use Any MPI Implementation

William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL
gropp@mcs.anl.gov
http://www.mcs.anl.gov/~gropp

**Abstract.** The Message Passing Interface (MPI) standard for programming parallel computers is widely used for building both programs and libraries. Two of the strengths of MPI are its support for libraries and the existence of multiple implementations on many platforms. These two strengths conflict, however, when an application wants to use libraries built with different MPI implementations. This paper describes several solutions to this problem, based on minor changes to the API. These solutions also suggest design considerations for other standards, particularly those that expect to have multiple implementations and to be used in concert with other libraries.

The MPI standard [1, 2] has been very successful (see, for example, [3]). Multiple implementations of MPI exist for most parallel computers [4], including vendor-optimized versions and several freely-available versions. In addition, MPI provides support for constructing parallel libraries. When an application wants to use routines from several different parallel libraries, however, it must ensure that each library was built with the *same* implementation of MPI. The reason rests with the lack of detailed specification in the header files '`mpi.h`' for C and C++, '`mpif.h`' for Fortran 77, or the MPI module for Fortran 90. Because the goals of MPI included high performance, the MPI standard gives the implementor wide latitude in the specification of many of the datatypes and constants used in an MPI program. For each individual MPI program, this lack of detailed specification in the header file causes no problems; few users are even aware that the specific value of, for example, `MPI_ANY_SOURCE` is not specified by the standard. Only the names are specified by the standard.

A problem does arise, however, in building an application from multiple libraries. The user must either mandate that a specific MPI implementation be used for all components or that all libraries be built with all MPI implementations. Neither approach is adequate; third-party MPI libraries (such as those created by ISVs) may be available only for specific MPI implementations, the building and testing each library for each MPI implementation are both time-consuming and difficult to manage. In addition, the risk of picking the wrong version of a library is high, leading to problems that are difficult to diagnose.

Furthermore, while building each library with each version of MPI is possible, this solution doesn't scale to other APIs that have multiple implementations. This is not a hypothetical problem. At least one ISV has already experienced this problem, as has any site that has installed MPICH, LAM/MPI, or MPICH-GM. Thus, a solution that allows an application to use *any* implementation of MPI (and, using similar techniques, other libraries) is needed.

At least two solutions to this problem are feasible. The first is completely generic. It effectively wraps all MPI routines and objects with new versions that work with any MPI implementation by deferring the choice of a specific implementation to link- or even run-time if dynamically-linked libraries are used. For this approach, one can use new tools for building language-specific interfaces from language-independent descriptions of components. The second approach may be applied to those MPI implementations whose opaque objects have the same size (e.g., `MPI_Requests` are four bytes in all implementations). Neither solution is perfect; both require some changes to the source code of an existing component that already uses MPI. In many cases, however, these changes can be automated, making their use almost transparent.

This paper reviews in Section 1 the MPI values and objects that are defined in the '`mpi.h`' header file. In Section 2, the paper describes the issues in creating a generic '`mpi.h`' for the C binding of MPI, with some comments about C++ and Fortran. Section 3 provides an example that shows the same object library used by two different popular MPI implementations.

The approach described here does not address the issues of performance portability, particularly the problem of chosing the MPI routines with the best performance. However, this approach does allow libraries to use the conventional approach of encapsulating performance-critical operations in separate routines and then using runtime configuration variables to choose the best approach. Performance measurement tools such as SkaMPI [6] can be used to help identify the appropriate choice for a particular MPI implemenation.

## 1 Brief Overview of MPI

The MPI specification itself is language independent. The MPI-1 standard specifies bindings to C and Fortran (then Fortran 77); the MPI-2 standard added bindings for C++ and Fortran 90. MPI programs are required to include the appropriate header file (or module for Fortran 90); for C and C++, this is '`mpi.h`'. The file includes: typedefs for MPI opaque objects, definitions of compile-time constants, definitions of link-time constants (see below), and function prototypes. The implementation, however, is given wide latitude in the definition of the named objects. For example, the following definitions for the MPI object used to describe data layouts (`MPI_Datatype`) are used by four common implementations:

| Implementation | Datatype Definition |
|---|---|
| IBM | `typedef int MPI_Datatype;` |
| LAM | `typedef struct _dtype *MPI_Datatype;` |
| MPICH | `typedef int MPI_Datatype;` |
| SGI | `typedef unsigned int MPI_Datatype;` |

While no implementation appears to use a `struct` rather than an `int` or pointer, nothing in the MPI standard precludes doing so.

The MPI standard also requires each routine to be available with both `MPI` and `PMPI` prefixes. For example, both `MPI_Send` and `PMPI_Send` are required. This is called the profiling interface. The intent of the `PMPI` versions is to allow a tool to replace the implementations of the `MPI` routines with code that provides added functionality (such as collecting performance data); the routine can then use the `PMPI` version to perform the MPI operation. This is a powerful feature that can be exploited in many applications and tools, such as the performance visualization tools Jumpshot [7] and Vampir [5].

One solution, mentioned above, creates a new binding for MPI that specifies all of the values and types. A "user" implementation of this binding could make use of the profiling interface. In order to use this new binding, however, applications must be rewritten. This solution will be described in another paper.

## 2 A Generic MPI Header

An alternative solution to a new MPI binding is to find a common version of the header file '`mpi.h`' that can be used by multiple implementations. This solution is less general than the generic binding but has the advantage of requiring fewer changes to existing codes. The approach here is to replace compile-time constants with runtime constants by finding a common data representation for MPI objects. There are some drawbacks in this approach that are discussed below. However, the advantages are that it requires few changes to existing code and most operations directly call the specific MPI routines (with no overhead).

The approach is as follows: Define a new header file that replaces most compile-time values with runtime values. These values are then set when the initialization of MPI takes place. In most cases, this can be accomplished at link time by selecting a particular library, described below, that contains the necessary symbols.

The header file '`mpi.h`' contains the following items that must be handled:

**Compile-time values.** These include the constants defining error classes (e.g., `MPI_ERR_TRUNCATE`) and special values for parameters (e.g., `MPI_ANY_SOURCE`). These can be replaced with runtime values that are initialized by `GMPI_Init`.
A special case are the null objects such as `MPI_COMM_NULL`. Most implementations define all null objects the same way (either zero or $-1$). The MPICH-2 implementation, however, encodes the object type in each object handle, including the null object handles. Thus, these terms must cannot be defined at compiletime.

**Compile-time values used in declarations.** These include constants defining maximum string sizes (e.g., `MPI_MAX_ERROR_STRING`). In many cases, these can be replace by either the maximum or the minimum over the supported implementations, for example,

| Implementation | Size of `MPI_MAX_ERROR_STRING` |
|---|---|
| SGI | 256 |
| IBM | 128 |
| LAM | 256 |
| MPICH | 512 |

Defining `MPI_MAX_ERROR_STRING` as 512 is adequate for all of these MPI-1 implementations, since this value is used only to declare strings that are used as output parameters. Other values describe the sizes of input arrays, such as `MPI_MAX_INFO_KEY`. In this case, the minimum value should be used. While this might seem like a limitation, a truly portable code would need to limit itself to the minimum value used in any supported MPI implementation, so this is not a limitation in practice.

**Init-time constants.** MPI defines a number of items that are constant between `MPI_Init` and `MPI_Finalize`, rather than compile-time constants. The predefined MPI datatypes such as `MPI_INT` belong to this category. For most users, the difference isn't apparent; it comes up only when users try to use one of these in a place where compile-time constants are required by the language (such as case labels in a switch statement). Init-time constants are actually easier to implement than compile-time constants because it is correct to define them as values that are initialized at run time. Fortunately, in most implementations, these are implemented either as compile-time constants or link-time constants (such as addresses of structures) and require no code to be executed to initialize them. Thus, we do not need a special `MPI_Init` routine to handle this case.

**Opaque objects.** These are both the easiest and the most difficult. In practice, most implementations define these as objects of a particular size, and further, most implementations choose to either `int`s or pointers to represent these objects. On many platforms, `int`s and pointers are the same size. For such platforms, we can simply pick one form (such as `int`) and use that.

If the size of these objects is different among different implementations, then there is no easy solution. We cannot pick the largest size because some MPI operations use arrays of opaque objects (e.g., `MPI_Waitsome` or `MPI_Type_struct`). The solution described here does not handle this case, though an approach that follows the technique used for `MPI_Status`, shown below, may be used.

One additional complication is the handle conversion functions introduced in MPI-2. These convert between the C and Fortran representations of the handles. For many implementations, these are simply cast operations, and the MPI standard allows them to be implemented as macros. However, for greatest flexibility, the generic header described here implements them as functions, permitting more complex implementations.

**Defined objects (status).** The most difficult case is `MPI_Status`. MPI defines this as a `struct` with some defined members such as `MPI_TAG`. An implementation is allowed to add its own members to this structure. Hence, the size of `MPI_Status` and the layout of the members may be (and is) different in each implementation. The only solution to this problem is to provide routines to allocate and free `MPI_Status` objects and to provide separate routines to access all of the elements. For example, instead of

```
MPI_Status status;
...
MPI_Recv( ..., &status );
if (status.MPI_TAG == 10 || status.MPI_SOURCE == 3) ...
```

one must use special routines such as

```
MPI_Status *status_p = GMPI_Status_create(1);
MPI_Recv( ..., status_p );
if (GMPI_Status_get_tag( status_p ) == 10 ||
    GMPI_Status_get_source( status_p )) ...
GMPI_Status_free( status_p, 1 );
```

Fortunately, many MPI programs don't need or use `status`. These programs should use the MPI-2 values `MPI_STATUS_NULL` or `MPI_STATUSES_NULL`.

**Defined pointers.** MPI also defines a few constant pointers such as `MPI_BOTTOM` and `MPI_STATUS_NULL`. For many implementations, these are just `(void *)0`. Like the most of the other constants, these can be set at initialization time.
`MPI_STATUS_NULL` is a special case. This is not part of MPI-1 but was added to MPI-2. If an MPI implementation does not define it, a dummy `MPI_Status` may be used instead. This will not work for the corresponding `MPI_STATUSES_NULL`, which is used for arguments that require an array of `MPI_Status`, but is sufficient for most uses.

The complete list of functions that must be used for accessing information in `MPI_Status` is as follows:

```
/* Create and free one or more MPI_Status objects */
MPI_Status *GMPI_Status_create( int n );
void GMPI_Status_free( MPI_Status *p );
/* Access the fields in MPI_Status */
int GMPI_Status_get_tag( MPI_Status *p, int idx );
int GMPI_Status_get_source( MPI_Status *p, int idx );
int GMPI_Status_get_error( MPI_Status *p, int idx );
int GMPI_Status_get_count( MPI_Status *p, MPI_Datatype dtype,
                           int idx, int *count );
```

For MPI-2, it is also necessary to add

```
void GMPI_Status_set_error( MPI_Status *p, int idx, int errcode );
```

To reduce the number of function calls, one could also define a routine that returns the tag, source, and count, given a status array and index. The routines shown above are all that are required, however, and permit simple, automated replacement in MPI programs.

These routines, as well as definitions of the runtime values of the various compile- and init-time constants, are compiled into a library with the name `libgmpito`*name*, where *name* is the name of the MPI implementation. For example, a cluster may have `libgmpitompich` and `libgmpitolam`. The definitions of the runtime values are extracted from the '`mpi.h`' header of each supported MPI implementation; special tools have been developed to automate much of this process.

Most C++ and Fortran bindings for MPI are based on wrappers around the C routines that implementations use to implement MPI. Thus, for C++ and Fortran, a variation of the "generic component" or new MPI binding mentioned above can be used. For those objects and items that are defined as constants, the same approach of using variables can be used.

Fortran shares the same issues about `MPI_Status` as C does. For Fortran 90, we can use a similar solution, returning a dynamically allocated array of `MPI_Status` elements and providing a similar set of functions to access the elements of `MPI_Status`.

## 3 Using the Generic Header

Applications are compiled in the same way as other MPI programs, using the header file '`mpi.h`'. Linking is almost the same, except one additional library is needed: the implementation of the GMPI routines in terms of a particular MPI implementation. For example, consider an application that uses GMPI and a library component that is also built with GMPI. The link line looks something like the following:

```
# Independent of MPI implementation (generic mpi.h in /usr/local/gmpi)
cc -c myprog.c -I/usr/local/gmpi/include
cc -c mylib.c -I/usr/local/gmpi/include
ar cr libmylib.a mylib.o
ranlib libmylib.a
# For MPICH
/usr/local/mpich/bin/mpicc -o myprog myprog.o -lmylib \
      -L/usr/local/gmpi/lib -lgmpitompich
# For LAM/MPI
/usr/local/lammpi/bin/mpicc -o myprog myprog.o -lmylib \
      -L/usr/local/gmpi/lib -lgmpitolam
```

With this approach, only one compiled version of each library or object file is required. In addition, for each MPI implementation, a single library implementing GMPI in terms of that implementation is required.

As an illustration, shown below is a simple library that can be used, in compiled form, with two different MPI implementations on a Beowulf cluster.

This simple library provides two versions of a numerical inner product: one that uses `MPI_Allreduce` and is fast and one that preserves evaluation order for the allreduce operation (unlike real numbers, floating-point arithmetic is not associative, so the order of evaluation can affect the final result), at a cost in performance. The code for the library routine is as follows:

```
#include "mpi.h"

double parallel_dot( const double u[], const double v[], int n,
                     int ordered, MPI_Comm comm )
{
   int    rank, size, i;
   double temp = 0, result;

   if (ordered) {
      MPI_Comm tempcomm;
      /* A good implementation would cache the duplicated communicator */
      MPI_Comm_dup( comm, &tempcomm );
      MPI_Comm_rank( tempcomm, &rank );
      MPI_Comm_size( tempcomm, &size );
      if (rank != 0)
          MPI_Recv( &temp, 1, MPI_DOUBLE, rank-1, 0, tempcomm,
                    MPI_STATUS_NULL );
      for (i=0; i<n; i++)
          temp += u[i] * v[i];
      if (rank != size-1)
          MPI_Send( &temp, 1, MPI_DOUBLE, rank+1, 0, tempcomm );
      MPI_Bcast( &temp, 1, MPI_DOUBLE, size-1, tempcomm );
      MPI_Comm_free( &tempcomm );
      result = temp;
   }
   else {
      for (i=0; i<n; i++)
          temp += u[i] * v[i];
      MPI_Allreduce( &temp, &result, 1, MPI_DOUBLE, MPI_SUM, comm );
   }
   return result;
}
```

This code uses the communicator that is passed into the routine. The following shell commands show how easy it is to build this library so that it may be used by either MPICH or LAM/MPI. The main program is in `myprog` and includes the call to `MPI_Init` and `MPI_Finalize`.

```
% cc -c -I/usr/local/gmpi/include dot.c
% ar cr libmydot.a dot.o
% ranlib libmydot.a
% cc -c -I/usr/local/gmpi/include myprog.c
% /usr/local/mpich/bin/mpicc -o myprog myprog.o \
```

```
        -lmydot -L/usr/local/gmpi/lib -lgmpitompich
% /usr/local/mpich/bin/mpirun -np 4 myprog
% /usr/local/lammpi/bin/mpicc -o myproc myprog.o \
        -lmydot -L/usr/local/gmpi/lib -lgmpitolam
% /usr/local/lammpi/bin/mpirun -np 4 myprog
```

This example has been run as shown with an implementation of the libraries `libgmpitompich` and `libgmpitolam` built for an IA32 cluster.

## 4 Conclusion

This paper has outlined an approach that allows libraries and object files to use several different MPI implementations without requiring recompilation and without requiring significant changes in the source for those components. In fact, some applications will require no changes at all. An implementation of this approach for MPICH and LAM/MPI on IA32 platforms has been constructed and demonstrated. In addition, source code transformation tools have been developed that aid in constructing the `libgmpito`*name* library for other MPI implementations. These tools and libraries will be available at
`http://www.mcs.anl.gov/mpi/tools/genericmpi`.

## Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

## References

1. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
2. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.
3. List of papers that use MPI. `http://www.mcs.anl.gov/mpi/mpiarticles`.
4. MPI implementations. `http://www.mcs.anl.gov/mpi/implementations.html`.
5. Vampir 2.0 – Visualization and Analysis of MPI Programs. `http://www.pallas.de/pages/vampir.htm`.
6. R. Reussner, P. Sanders, L. Prechelt, and M Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998.
7. Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.