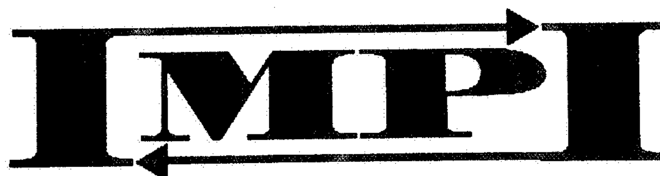
NISTIR 7066

Parallel Programming with Interoperable MPI



William L. George
John G. Hagedorn
Judith E. Devaney
Mathematical and Computational Sciences Division
Information Technology Laboratory

December 2003

U.S. DEPARTMENT OF COMMERCE
Donald L. Evans, Secretary
TECHNOLOGY ADMINISTRATION
Phillip J. Bond, Under Secretary of Commerce for Technology
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arden L. Bement, Jr., Director

Parallel Programming with Interoperable MPI

William L. George
John G. Hagedorn
Judith E. Devaney

1 Introduction

Modern computing centers provide their users with a variety of computing resources ranging from single processor workstations to high-performance parallel computers. Often included in the mix are Beowulf class machines [1], that is, clusters of commodity personal computers (PCs) configured to operate as parallel computers. To implement portable parallel scientific applications for these systems the message-passing library MPI (Message Passing Interface) [2] is typically used. This widely available library provides a C and Fortran interface to routines for sending data (messages) between processors.

Parallel applications are normally run on a single parallel machine such as a shared-memory multiprocessor, a distributed-memory machine such as a Beowulf cluster, or more commonly, a hybid distributed-memory/shared-memory machine consisting of a networked cluster of shared-memory compute nodes. However, there is often a strong desire to harness the resources of two or more such machines, forming what we will call a *multi-cluster*, to perform a single computation. This would be required, for example, for simulations that are too large to be performed on any one of the available parallel machines individually. While MPI programs are supported on most parallel machines, through an MPI library provided specifically for that machine, there is no mechanism within MPI to allow these disparate MPI libraries to cooperate. Interoperable MPI (IMPI) [3, 4] provides a means to accomplish this with minimal effort on the part of the application programmer. IMPI is a set of protocols, implemented within an MPI library, that allow multiple MPI libraries to cooperate, acting like a single MPI library for programs running on a multi-cluster. This article gives an introduction to IMPI including several examples of how it can be used.

2 A crash course in MPI

For those unfamiliar with message-passing, we next describe some basics of this programming style using C and MPI. Assuming we are running a program using P processes[1], each process will be identified in calls to MPI by an integer rank from 0 to $(P-1)$. Figure 1 shows a simple C program that sends an integer from the lowest rank process to the highest rank process.

Once this program is compiled and linked to the MPI library (normally by using the `-lmpi` option on the link command), it can be executed by a command-line utility program provided with the MPI library. Often this utility is named `mpirun`. The command-line to run our program with 8 MPI processes could look like:

```
mpirun -np 8 program1
```

assuming our executable is named `program1` and `-np` is the command-line switch for specifying the number of MPI processes (this syntax varies between MPI implementations).

Examining the source code to this program, the `MPI_Init` and `MPI_Finalize` calls are required in all MPI programs. No calls to MPI routines can be made before the call to `MPI_Init` or after the call to `MPI_Finalize`. To get the rank of the local process we call `MPI_Comm_rank` and to get the total number of processes we call `MPI_Comm_size`.

In most MPI routines, an MPI *communicator* is a required parameter. A communicator describes a set of processes, including the assignment of ranks to those processes, and also defines a separate communications context. A message sent using one communicator can only be received by a call using the same communicator. The predefined communicator `MPI_COMM_WORLD` simply includes all of the processes, however subsets of `MPI_COMM_WORLD` are possible.

In this program, the communication is performed with the most basic MPI communications routines `MPI_Send` and `MPI_Recv`. The parameters to these routines describe the message to be sent/received (`message`, `count`, and `MPI_INT`), the rank of the destination process (`dst` for `MPI_Send`) or source process (`src` for `MPI_Recv`), an arbitrary `tag` value, and an MPI communicator for message matching. The `status` parameter to the `MPI_Recv` routine holds details of the status of the message once it has been received.

[1] A *process* in this context is a separate asynchronous thread of control that, ideally, runs on a dedicated processor, although it could timeshare a processor with other processes. All processes in an MPI program can run concurrently and synchronize when they need to communicate.

```
//-----------------------------------------------------------
#include <mpi.h>

int main(int argc, char *argv[])
{
  int my_rank, src, dst, tag, message, nprocs, count;
  MPI_Status status;

  count=1;
  tag=100;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  src=0;
  dst=nprocs-1;
  if (my_rank == src) {
    message=42;
    MPI_Send(&message, count, MPI_INT, dst,
             tag, MPI_COMM_WORLD);
  } else if (my_rank == dst) {
    MPI_Recv(&message, count, MPI_INT, src,
             tag, MPI_COMM_WORLD, &status);
  }
  MPI_Finalize();
  return 0;
}
//-----------------------------------------------------------
```

Figure 1: An MPI program that sends an `int` from the process with the lowest rank (0) to the process with the highest rank.

So where does IMPI fit into all of this? At the source code level, an IMPI program is simply an MPI program. Adding IMPI support to an MPI library does not add, remove, or change any user level MPI routines. However, as will be seen in the description of some example IMPI applications later in this article, there can be some additional considerations to take into account when writing an MPI program that is specifically designed to be run on a heterogeneous collection of parallel machines.

3 Starting an IMPI program

When running an MPI program on a multi-cluster with IMPI, each of the clusters or parallel machines in the multi-cluster is referred to as an IMPI *client*. Before running the program, the user must decide on an order for these clients. This ordering determines the ranking of the processes in `MPI_COMM_WORLD` such that the ranks of the processes in client 0 are the lowest ranks, followed by the ranks of the processes in client 1, and so on. This client rank must be a number from zero to one less than the number of clients.

Normally, an MPI program is started with a command such as:

```
mpirun -np <N> program-name <args>
```

where <N> is the number of processes to use and `<args>` are any command-line arguments for the program. To run an MPI program using IMPI on a multi-cluster an IMPI server process must first be started using the command

```
mpirun -server <count>
```

where `<count>` is the number of IMPI clients that will be started. The IMPI server is the rendezvous point for the IMPI clients and acts as a relay between the clients during the startup of the IMPI program. The IMPI server will print to the terminal a string such as `192.168.0.1:12345`, which gives the IP address and the port number of the IMPI server. This information, in this exact form, is needed to start the clients. Once the IMPI server is running, each of the clients can be started with a command of the form:

```
mpirun -client <C> <host:port> <rest>
```

where <C> is the client number, <host:port> is the rendezvous information from the IMPI server, and <rest> is the rest of the standard mpirun command-line[2].

Once an MPI program has started, all of the processes from all of the IMPI clients are included in the MPI communicator MPI_COMM_WORLD and they are ranked according to the ranks given to the IMPI clients.

4 Some IMPI Usage Patterns

Now that we have described the basics of parallel message-passing programming with MPI and how to start an IMPI program, we now show how IMPI can be used to expand the power of MPI programs. We will show several types of applications that we anticipate will use IMPI to great advantage; there are most likely many others we have not yet considered.

We are not describing new classes of parallel programs here; were are instead describing types of parallel programs that are easily supported by IMPI and are likely to be successfully run in a multi-cluster environment.

Case 1: Legacy data-parallel programs.

One immediate use that we anticipate for IMPI is to simply allow legacy MPI programs to be run in a multi-cluster environment. The motivation for doing so would be to either decrease the total execution time of the program or, more likely, to enable the running of larger problems than would be possible on any one of the clusters alone.

There are aspects of this use of IMPI that could require some modifications to the programs in order to obtain reasonable performance. Unless the processing nodes in the clusters of this environment are closely matched in speed, available memory, and I/O (Input/Output) capabilities, some load-balancing that was not needed on a homogeneous set of processing nodes may be required.

One other consideration that needs to be addressed in running legacy data-parallel applications in a multi-cluster environment is the handling of file I/O. Depending on the configuration of the networks connecting the clusters, and whether disk volumes are cross-mounted with some form of networked file system, some pre-processing and post-processing to move input and output files to where they are needed may also be required.

[2]The command name "mpirun" is used in this example, but this name is not mandated by MPI (or IMPI). Refer to the documentation for your MPI library for the correct command name.

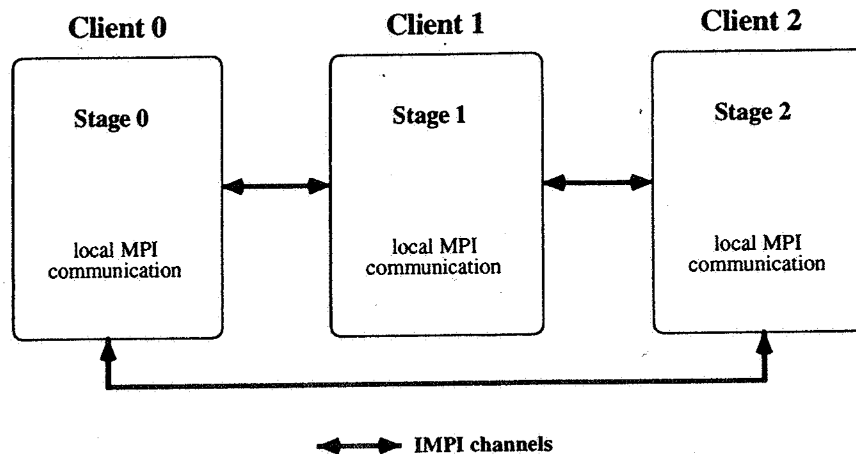


Figure 2: Using IMPI in a 3-stage large-grained parallel application. One IMPI client is assigned to each stage of the computation.

Case 2: Large-grained parallel programs.

Another anticipated use of IMPI is in support of applications designed as large grain data-flow algorithms. A simple case of this is a computation comprised of several large computational stages, structured such that each can be executed on a separate parallel machine. One example of this type of application is a global climate simulator. This simulator could include separate models (computational stages) for the ocean, the lower atmosphere, and the upper atmosphere, with defined physical boundaries between each of these modeled environments. Each of these models could be run on separate parallel machines with the coupling between the models enabled by communication over the *IMPI channels*, that is, the network connections between the IMPI clients (see Fig. 2).

In this type of application, each MPI process will need to know not only its rank within the `MPI_COMM_WORLD` communicator, but also to which stage of the computation it belongs and possibly which stage each of the processes in `MPI_COMM_WORLD` belongs to. IMPI provides this information to the application at run-time through the use of an existing MPI facility called *attribute caching*. This allows for arbitrary information to be associated with an MPI communicator for each process. For IMPI support, each process can determine which IMPI client it belongs to by retrieving a cached attribute called the `IMPI_CLIENT_COLOR` which is simply an integer. For the communicator `MPI_COMM_WORLD`, this integer will be identical to the client rank given to the `mpirun` command.

The term COLOR is used to match the terminology used in the MPI routine MPI_Comm_Split(MPI_Comm comm, int color, ...), a routine that creates a set of new communicators, each of which consists of all of the MPI processes that share the same color. For our large-grained parallel application, each MPI process would pass in its IMPI client color. This is likely to be one of the first operations completed in this type of IMPI application so that the processes in each stage can obtain their own private communicator to use within its stage of the computation. Here are the MPI calls needed to create the communicators for each stage of the computation:

```
int *stage, stat, stage_rank;
MPI_Comm stage_comm;

MPI_Attr_get(MPI_COMM_WORLD, IMPI_CLIENT_COLOR,
             &stage, &stat);
MPI_Comm_split(MPI_COMM_WORLD, *stage, 0, &stage_comm);
MPI_Comm_rank(stage_comm, &stage_rank);
```

Once these calls are completed, each MPI process will know to which stage it belongs (*stage), have an MPI communicator for communications within the set of processes that comprise that stage (stage_comm), and will know its rank within that set of processes (stage_rank).

The third parameter in the call to MPI_Comm_split can be used to allow the re-ordering (re-ranking) of the processes in stage_comm if the MPI implementation would like to do so (presumably for performance reasons); 0 here means do not re-order the processes.

Thus, using the IMPI supplied attribute IMPI_CLIENT_COLOR in addition to the standard MPI routines for creating new communicators, you can implement a large-grained parallel application that adapts to whatever size clusters (IMPI clients) you have available. More work would be needed if you wanted, for load balancing purposes, to either assign more than one client to one of the computational stages or more than one stage to a single client. For finer control of the load balance, for example by allowing a single cluster to be split internally between two stages, further work on this basic software architecture would be needed.

Example code for enabling communication between the three computational stages, using the MPI routine MPI_Intercomm_create to create special MPI communicators, can be found in the MPI 1.1 document, Section 5.6.3 Intercommunication Examples [5]. This is a standard MPI programming technique that is not affected by the use of IMPI.

Case 3: Computational Steering/Interactive applications

The ability to monitor the progress of a large simulation, especially during its initial development, can be of great help in debugging the code and in determining experimentally a set of reasonable simulation parameters. In this case, we can use IMPI to run two or three sub-programs, all aware of each other and connected via MPI. These extra programs are used for monitoring and controlling the main simulation. This is close to the model-view-controller (MVC) style of program [6, 7], for those familiar with developing graphical applications, except that the coupling between the model, view, and controller is much looser. With the size and computational complexity of the models (simulations), the time between view updates may be from minutes to hours or even longer. Figure 3 shows a configuration of IMPI clients for this type of parallel application. In this figure, MPI processes are shaded to indicate the various values of the `IMPI_CLIENT_COLOR` attribute. Code similar to that shown for the large-grained parallel program could be used to create MPI communicators for each of the distinct parts of the program (simulator, monitor, and controller), and then, assuming the simulator is itself a large-grained parallel program, create the communicators for each of the stages of that simulator. The outline for this type of IMPI application is shown in Figure 4.

Note that each of the clients shown in Figure 4 can, and will likely, be running on a different CPU architecture and operating system as well as using a different implementation of MPI. To make this work, IMPI handles the conversion of data types between these systems automatically when, for example, sending an integer or floating point value between machines with different internal data formats. Also, *collective* communication operations, such as the broadcasting of a value from one MPI rank to all of the other MPI ranks, are implemented in IMPI in such a way so as to take advantage of the vendor tuned MPI libraries as much as possible. That is, the "IMPI channels", as shown in Figure 3, are assumed to be slower (lower bandwidth, higher latency) than the internal networks on the individual client clusters, and so their use is minimized in the IMPI collective communications routines (such as broadcast).

As with the large-grained parallel program case, communication between the the viewer, controller, and the simulator is enabled by creating special MPI communicators using the MPI routine `MPI_Intercomm_create`.

So, the model part of this program contains the simulation that is to be run on one or more clusters. This part of the program can be a data-parallel or large-grain pipelined program as previously described or any other type of MPI program. It is also possible for this simulator to be a multi-threaded program that runs on a large shared-memory machine which uses MPI only to communicate with the view and controller parts of the IMPI program.

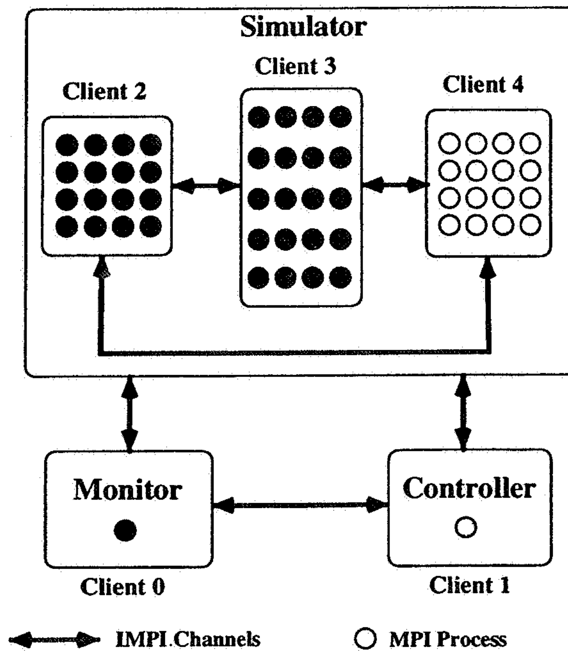


Figure 3: Using IMPI for Computational Steering. One client is assigned to the monitor, one to the controller, and three to the simulator. Each MPI process, represented in this figure by a circle, has a value for the IMPI_CLIENT_COLOR attribute that is cached onto MPI_COMM_WORLD. These attribute values, which match the associated client numbers shown in the figure, are emphasized here by mapping each value to a separate shade of grey.

```
//-----------------------------------------------------------
  int *color, stat, rank;
  MPI_Comm comm;

  MPI_Attr_get(MPI_COMM_WORLD, IMPI_CLIENT_COLOR,
               &color, &stat);
  /* Simulator gets all clients > 1 */
  if (color > 1) color=2;
  MPI_Comm_split(MPI_COMM_WORLD, *color, 0, &comm);
  switch (color) {
  case 0: /* Call the Controller */ break;
  case 1: /* Call the Monitor */ break;
  case 2: /* Call the Simulator */; break;
  }
//-----------------------------------------------------------
```

Figure 4: Using the `IMPI_CLIENT_COLOR` attribute.

The second program is a monitor program (the view portion of MVC) that performs the following steps in a loop: accept image data from the simulation, possibly once every iteration of its main loop; render this data into a form suitable for the target display; and display the image, either on your workstation or other suitable device. If the simulation is not working as expected, you will know this as early as possible. To minimize the effect of this monitoring on the performance of the simulator, the communication between the simulation and the monitor can be reduced by decimating the image data or reducing the frequency of image updates.

The third program, if needed, allows for some amount of interactivity with the simulation, perhaps allowing you to modify the controlling parameters of the simulation or, more drastically, allowing you to kill it or restart the simulation from within the main simulation. This control could also allow you to turn on and off the monitoring of the simulation as needed.

5 Conclusion

IMPI allows legacy MPI programs to be run unaltered on *multi-clusters* consisting of two or more computing resources such as parallel machines, clusters, workstations, and single or multiprocessor PCs. Also, applications can be written specif-

ically to be run in such a multi-cluster allowing greater control over various aspects of the application such as load-balancing and file I/O. One major design advantage of IMPI over other available solutions to the problem of running on a multi-cluster is that IMPI uses the vendor-tuned MPI libraries for optimum communication within each parallel machine while still allowing the unrestrained use of all of MPI, including optimized collective communications operations that involve all of the processes in the multi-cluster.

If you wish to experiment with IMPI, the freely available MPI library LAM/MPI [8] currently supports IMPI. Full implementations of IMPI are included in the MPI libraries from Hewlett-Packard Co., MPI Software Technology, Inc., and Pallas GmbH (for Fujitsu). Other implementations of IMPI are anticipated in the future.

Disclaimer: Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to Build a Beowulf.* MIT Press, 1999. <http://www.beowulf.org>.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Scientific and Engineering Computation Series. MIT Press, 1999.

[3] William George, John Hagedorn, and Judith Devaney. IMPI: Making MPI interoperable. *Journal of Research of the National Institute of Standards and Technology*, 105(3), 2000. May-June issue. Article includes the entire IMPI specification as an appendix. <http://nvl.nist.gov/pub/nistpubs/jres/jres.htm>.

[4] William George, John Hagedorn, and Judy Devaney. A Java-based tool for testing interoperable MPI protocol performance. In *Proceedings of the First International Conference and Exhibition on the Practical Application of Java*, pages 111–124. The Practical Application Company Ltd., April 1999. London, UK.

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>.

[6] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Orientated Software*. Addison-Wesley, 1995.

[8] Jeffrey M. Squyres, Andrew Lumsdaine, William L. George, John G. Hagedorn, and Judith E. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000.