CMP
United Business Media

New Series! Rethinking Software Security

#357 FEBRUARY 2004

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

http://www.ddj.com

WEB SERVICES

- WS-I & Interoperable Web Services
- Integrating XML Web Services With VB6 Apps
- UDDI & Dynamic Web Service Discovery

I/O Multiplexing & Scalable Socket Servers

Regular Expression Mining

Applying the Overtake & Feedback Algorithm

Parallel Programming & Interoperable MPI

Communicating with Your Manager

ASP.NET Forms Authentication

Eudora Mailbox Classes

Multitasking on the Cheap

Parallel Programming with Interoperable MPI

Building portable applications for diverse systems

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

Modern computing centers typically provide users with a variety of computing resources, ranging from single-processor workstations to high-performance parallel computers. Increasingly, this mix also includes Beowulf class machines—clusters of commodity PCs configured to operate as parallel computers. The Message Passing Interface (MPI) library, which provides C and Fortran interfaces to routines for sending data (messages) between processors, was designed to implement portable applications for diverse systems such as these.

Although parallel computations are normally run on single parallel computers, there is often a need to harness the resources of multiple clusters and parallel computers, forming what we call "multiclusters" to perform a single computation. (For information on related technologies, see the accompanying text box entitled "Multicluster Environments.") This might be required, for example, for simulations that are too large to be performed on any available individual parallel machine. Interoperable MPI (IMPI) provides a means of accomplishing this with minimal effort on the part of application programmers. IMPI is a set of protocols—implemented within an MPI library—that let multiple MPI libraries cooperate, acting like a single MPI library for programs running on a multicluster. The IMPI protocol specification is available at http://impi.nist.gov/impi-report/index.html. In this article, we'll examine IMPI and provide examples of how it can be used. For background on parallel architectures and programming, see the accompanying text box entitled "Basic Parallel Architectures and Programming."

William and John are computer scientists and Judith is group leader in the Scientific Applications and Visualization Group at the National Institute of Standards and Technology. They can be contacted at william.george@nist.gov, john.hagedorn@nist.gov, and judith.devaney@nist.gov, respectively.

A Crash Course in MPI

For readers unfamiliar with message passing, we'll briefly describe some basics of this programming style using C and MPI. Assuming you are running a program using *P* processes, each process will be identified in calls to MPI by an integer rank from 0 to (*P*–1). Listing One, for instance, sends an integer from the lowest rank process to the highest rank process.

Once this program is compiled and linked to the MPI library (*-lmpi*), it can be executed by a command-line utility program provided with the MPI library. Often this utility is named "mpirun." 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), the command line to run our program with eight MPI processes could look like: *mpirun -np 8 program1*.

In Listing One, 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, you call *MPI_Comm_rank*. To get the total number of processes, 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 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 Listing One, 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 message once it has been received.

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, there can be some additional considerations to take into account when writing an MPI program that is specifically designed to run on a heterogeneous collection of parallel machines.

Starting an IMPI Program

When running an MPI program on a multicluster with IMPI, each cluster or parallel machine in the multicluster is referred to as an IMPI client. Before running the program, users 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 0 to 1 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. To run an MPI program using IMPI on a multicluster, 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.

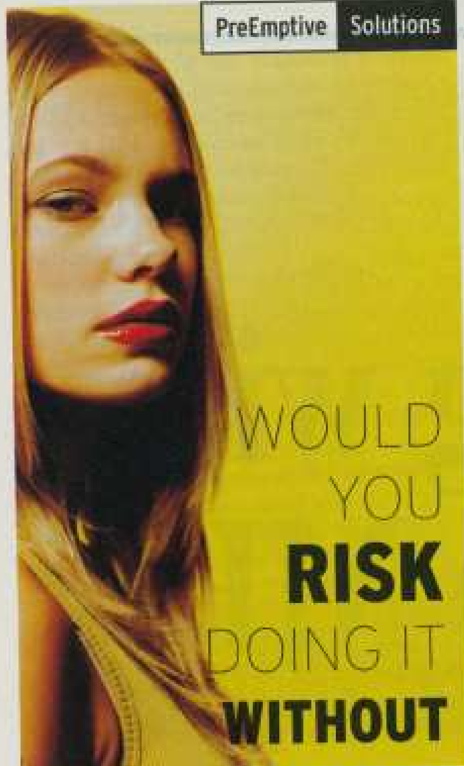
Once an MPI program has started, all of the processes from all of the IMPI clients are included in the MPI communicator MPI_COMM_WORLD, and they are ranked

Multicluster Environments

The desire to run MPI programs across heterogeneous sets of clusters has been around since the introduction of MPI, and other projects have provided this capability in different ways. For example, two portable and freely available implementations of MPI, MPICH (see "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," by William Gropp, Ewing Lusk, N. Doss, and Anthony Skjellum, *Parallel Computing*, September 1996, http://www-unix.mcs.anl.gov/mpi/mpich), and LAM (http://www.lam-mpi.org/) are capable of running programs across heterogeneous clusters of machines, so long as you use the same library (MPICH or LAM) on each of the clusters. Another MPI library, MagPIe (see "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems," by Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A.F. Bhoedjang, *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* [PPoPP'99], May 1999), is based on the MPICH source code and contains updated implementations of the MPI collective communications routines that are optimized for operation over WANs. All of these solutions, however, preclude you from using the vendor-tuned MPI libraries, thus sacrificing performance within each parallel machine or cluster.

The MPI-Connect project (see "MPI Inter-connection and Control," by G.E. Fagg and K.S. London, *Technical Report Tech Rep. 98-42*, Corps of Engineers Waterways Experiment Station Major Shared Resource Center, 1998) used another message passing library—PVM, short for "portable virtual machine" (*PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, by Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, MIT Press, 1994) to connect processes under the control of different MPI libraries. While this lets you use the vendor MPI libraries on each of the machines, MPI-Connect does not allow the use of any of the MPI collective operations, such as broadcast or reduce. On a larger scale, the Global Grid Forum (http://www.gridforum.org/) is coordinating a set of projects that will make computers, and other resources such as large databases, telescopes, wind tunnels, particle accelerators, and the like, available for use remotely and in concert. Many issues such as user authentication, resource scheduling, and security are being investigated by this forum.

—W.L.G., J.G.H., J.E.D.



PROTECTION?

I DON'T TAKE RISKS.
I **DOTFUSCATE** MY .NET CODE;
I **DASHO** MY JAVA CODE.

PreEmptive's Dotfuscator® for .NET and DashO™ for Java help protect your programs against reverse engineering while making them smaller and more efficient. Both have an easy to use GUI and command line interface for seamless integration with your build process. The benefits are superior intellectual property protection, decreased application size, and better program performance.

It's your code. Protect it.™

dotfuscator®
dashO™
www.preemptive.com
800.996.4556

according to the ranks given to the IMPI clients.

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 turn to how IMPI can be used to expand the power of MPI programs. There are several types of applications that we anticipate will use IMPI to great advantage, but most likely there

Basic Parallel Architectures and Programming

At the highest level, most parallel scientific programs can be characterized as either task parallel or data parallel. These terms describe how the program obtains parallelism.

A task-parallel program consists of multiple independent tasks that can be completed with little or no communication between the tasks. Typically, one process is in charge of assigning the tasks to the available processors and collecting the results. The SETI@home project is one example of this model of parallel processing (http://setiathome.ssl.berkeley.edu/). The amount of parallelism available in a task-parallel program increases as the number of independent tasks increases.

A data-parallel program typically operates on large multidimensional arrays with a main loop that updates these arrays once per iteration while converging toward a solution. Each iteration of the main loop completes identical, or nearly identical, calculations to update each of the elements in one or more of the large arrays. The amount of parallelism available in a data-parallel program increases with the size of the arrays. Data-parallel programs often require communication between the processing nodes during the update calculations. Therefore, the distribution of the data among the processing nodes is an important consideration when designing these programs.

A classification scheme also exists for describing hardware that supports parallel programs. This classification scheme focuses on one of the most important considerations in parallel programming—access to main memory by the processors. At the top level, a parallel machine can be described as either a distributed-memory machine or a shared-memory machine.

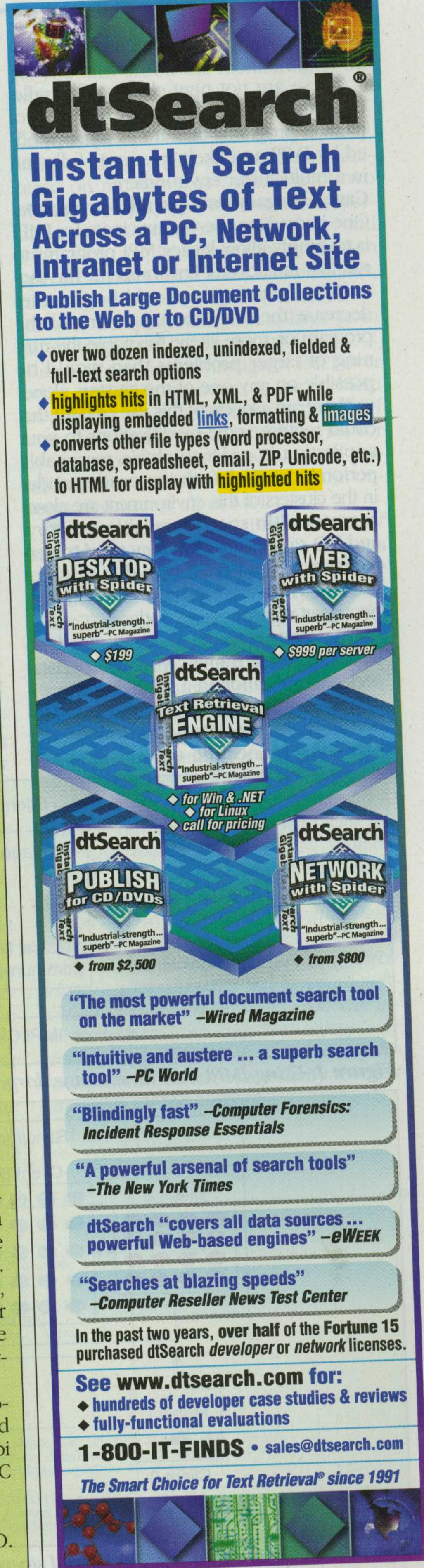
In a distributed-memory machine each processor has its own local main memory that is not directly accessible by other processors. Sharing data between processors is accomplished via message passing; that is, explicitly sending data from one processing node to another. These messages are sent over a network that connects the processors.

In a shared-memory machine, all processors have equal access to all available memory. Like distributed-memory machines, shared-memory machines also need an interconnection network; however, in this case the network connects the processors to the main memory. For best performance, applications must avoid contention on this network by avoiding multiple simultaneous requests for data stored in the same area of memory. Even on a shared-memory machine, message-passing style programs, using MPI, perform well. In this case, each process only directly accesses portions of the memory that hold its data. Message passing is implemented (within MPI) using standard memory-to-memory moves. Unlike most multithreaded shared-memory applications, this results in very scalable parallel applications due to the low contention on the processor-to-memory interconnection network.

Of course in the real world, these classification schemes are blurred. Currently, many high-performance parallel machines available from manufacturers such as IBM, Sun, Hewlett-Packard, and SGI are distributed-memory machines with high-speed interconnection networks, in which each processing node in the network is a 2- to 16-processor shared-memory multiprocessor. Additional hardware and software is sometimes available for these machines that provide you with a shared-memory API on top of the basic distributed-memory architecture. Luckily, with vendor-tuned MPI libraries, all of these machines can still run your C/Fortran MPI programs without change and with good communications performance.

For an introduction to parallel programming with MPI, including books and online tutorials, see http://www.lam-mpi.org/tutorials/ and http://www.ERC.MsState.Edu/labs/hpcl/projects/mpi/.

—W.L.G., J.G.H., J.E.D.



are many others we have not yet considered.

These are not new classes of parallel programs we are describing, but types of parallel programs that are easily supported by IMPI and likely to successfully run in a multicluster environment.

Case 1: Legacy data-parallel programs. One immediate use we anticipate for IMPI is to simply allow legacy MPI programs to run in a multicluster 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 your application in order to obtain reasonable perfomance. Unless the processing nodes in the clusters of this environment are closely matched in speed, available memory, and I/O capabilities, you may need to perform some load balancing that was not needed when you ran only on a homogeneous set of processing nodes.

One other consideration that needs to be addressed in running your legacy data-parallel application in a multicluster 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 filesystem, you may need to add some preprocessing and postprocessing to move input and output files where they are needed.

Case 2: Pipelined programs. Another anticipated use of IMPI is in support of applications designed as large-grain dataflow algorithms. A simple case of this is a pipelined computation comprised of several large-grain stages. Each stage of the computation 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 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 Figure 1).

In this type of application, each MPI process needs 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 runtime 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 pipelined application, each MPI process would pass in its IMPI client color. This is likely to be one of the first operations completed in a pipelined IMPI application so that the processes in each stage of the pipeline obtain their own private communicator to use within its stage of the computations. Listing Two presents the MPI calls needed to create the communicators for each stage of the computation.

Once these calls are completed, each MPI process knows to which stage it belongs (*stage*), has an MPI communicator for communications within the set of processes that comprise that stage (*stage_comm*), and knows 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 reordering (reranking) of the processes in *stage_comm* if the MPI implementation would like to do so (presumably for performance reasons); 0 here means do not reorder 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 pipelined application that adapts to whatever size clusters (IMPI clients) you have available. More work would be needed if you wanted to either assign more than one client to one of the computational stages or more than one stage to a single client.

Example code for enabling communication between the three pipeline stages, using the MPI routine *MPI_Intercomm_create* to create special MPI communicators, is provided in the MPI 1.1 document, Section 5.6.3 Intercommunication Examples (http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html). This is a standard MPI programming technique not affected by the use of IMPI.

Case 3: Computational steering and interactive applications. The ability to monitor the progress of large simulations,

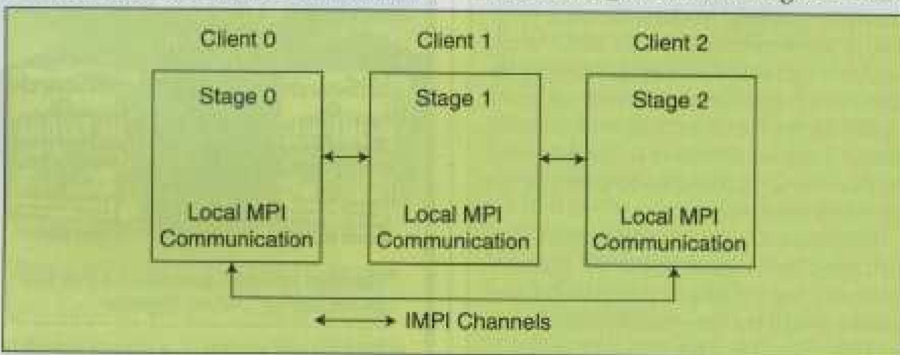


Figure 1: *Using IMPI in a three-stage, large-grained parallel application.*

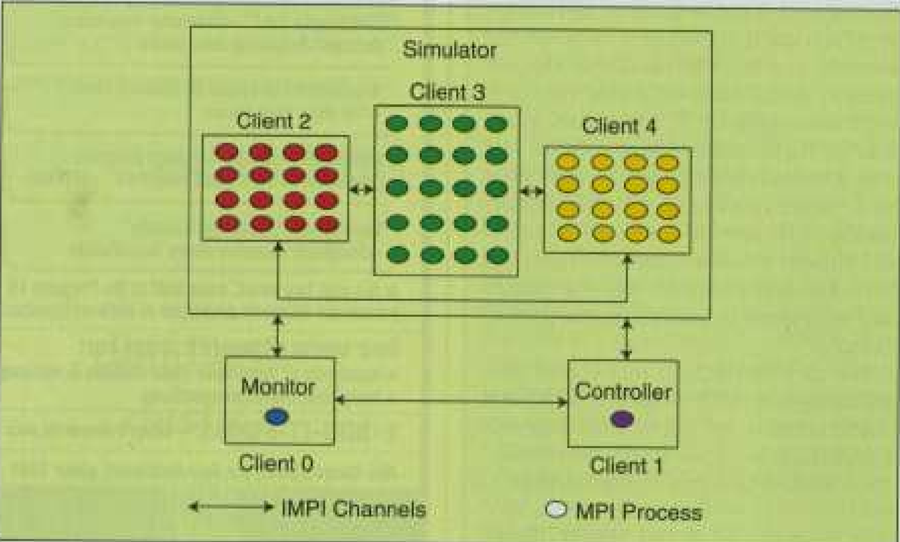


Figure 2: *Using IMPI for computational steering.*

especially during initial development, can be of great help in debugging the code and in experimentally determining a set of reasonable simulation parameters. In this case, we can use IMPI to run two or three subprograms, all aware of each other and connected via MPI. These extra programs are used for monitoring and controlling the main simulation. This is akin to the model-view-controller (MVC) style of program, except that the coupling between the model/view/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 2 shows a configuration of IMPI clients for this type of parallel application. In Figure 2, MPI processes are colored to indicate the various values of the IMPI_CLIENT_COLOR attribute. Code similar to that shown for pipelined programs 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 a pipelined program, create the communicators for each of the stages of the pipeline. The outline for this type of IMPI application is as follows:

One client is assigned to the monitor, one to the controller, and three to the simulator. Each MPI process, represented in Figure 2 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 in Figure 2, are emphasized here by mapping each value to a separate color; see Listing Three.

As with the pipelined program case, communication between the viewer, the controller, and the simulator is enabled by creating special MPI communicators using the MPI routine *MPI_Intercomm_create*.

So, the model part of Listing Three 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 multithreaded program that runs on a large shared-memory machine that uses MPI only to communicate with the view and controller parts of the IMPI program.

The second program referenced in Listing Three 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 letting you modify the controlling parameters of the simulation or, more drastically, allowing you to kill or restart the simulation from within the main simulation. This control could also let you turn on/off the monitoring of the simulation as needed.

> *There are several types of applications that will use IMPI to great advantage*

Conclusion

IMPI lets legacy MPI programs run unaltered on multiclusters consisting of two or more computing resources such as parallel machines, clusters, workstations, and PCs. Also, applications can be written specifically to run in such a multicluster, allowing greater control over various aspects of the application such as large-grain pipelining, load balancing, and file I/O. One major design advantage of IMPI over other available techniques to the problem of running on a multicluster 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.

The freely available MPI library LAM/MPI (http://www.lam-mpi.org/) supports IMPI. Full implementations of IMPI are available from Hewlett-Packard, MPI Software Technology, and Pallas GmbH (for Fujitsu). Other implementations of IMPI are anticipated in the future. Furthermore, the National Institute of Standards and Technology (NIST) IMPI test tool (http://impi.nist.gov/ImpiTT.html) lets you test IMPI implementations for conformance to the IMPI protocol standard. For a detailed background on MPI, see *Using MPI: Portable Parallel Programming with the Message Passing Interface*, by William Gropp, Ewing Lusk, and Anthony Skjellum (MIT Press, 1999).

Note: 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.

DDJ

Listing One

```
#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;
}
```

Listing Two

```
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);
```

Listing Three

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

MPI_Attr_get(MPI_COMM_WORLD, IMPI_CLIENT_COLOR, &color, &stat);
if (color > 1) color=2; /* Simulator gets all clients > 1 */
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;
}
```

DDJ