Status Report on the Development of the Interoperable MPI Protocol

William George
John Hagedorn
Judith Devaney
National Institute of Standards and Technology

Abstract— **An industry lead effort to create a protocol for the interoperation of implementations of the Message Passing Interface (MPI) has been ongoing since March of 1997. Meetings have been held several times per year with many implementors of MPI participating to help define the Interoperable MPI protocol (IMPI). This process has resulted in a simple protocol that can be incorporated into existing implementations of MPI to provide for interoperability.**

In this paper we describe IMPI along with some of the reasoning for its design. We also describe an IMPI tester, developed at NIST, that tests implementations for compliance to the protocol. The first complete version of IMPI will be ready by the time this report is published and we expect IMPI compliant implementations of MPI to become available soon after.

Keywords— **MPI, IMPI, interoperability**

I. Introduction

THIS paper describes the status of the on-going effort to define a protocol that enables interoperation of MPI implementations. This protocol, known as Interoperable MPI (IMPI), will enable MPI developers to provide MPI libraries that will interoperate with any other MPI implementation that supports the protocol. Application developers will be able to create MPI jobs that run across heterogeneous computing environments using multiple MPI implementations.

The full description of the IMPI protocol can be found in the report of the IMPI Steering Committee [1]. The first complete version of the IMPI protocol, version 1.0, will be available by the time this paper is published. We want to point out that the work described here is really the work of the IMPI Steering Committee. The committee has been meeting periodically since March of 1997, and the members of the committee have put in a great deal of work into developing this protocol.

The National Institute of Standards and Technology (NIST) has been involved in the IMPI effort from the beginning, playing the part of a facilitator: arranging meetings, running a mailing list, etc. But all technical decisions about the protocol definition have been left to the participating MPI vendors. As part of its role as a facilitator, NIST is developing a software tool that will enable the IMPI developers to test conformance to the protocol.

NIST is also funding a group at the University of Notre Dame to add IMPI to the LAM implementation of MPI [2], `http://www.lsc.nd.edu/research/impi`.

We begin by giving some background on interoperability issues and the approach taken by this effort. Then we describe some details of the protocol including program start up, data transfer, and collective communications algorithms. The NIST IMPI test tool is discussed, and we close with a brief summary and description of possible future work.

II. Background

The message-passing library MPI [3] has been widely adopted as the standard for developing parallel scientific programs for distributed-memory machines. Implementations of MPI exist for all commercial machines that are commonly used for high performance computing. This ensures that a parallel program, written in either C or Fortran, and using MPI for all message passing, will run on most available platforms. The implicit assumption made here is that the nodes of the available platform all run under the same implementation of MPI which is normally true when running on a distributed memory machine specifically designed for this type of computing.

Partly due to the success of MPI, clusters of workstations or PCs have become attractive alternatives to the high-end parallel machines such as the IBM SP2 and the SGI Origin series machines. In particular, *Beowolf*-class systems consisting of clusters of PCs running the Linux operating system and communicating over a commodity, off the shelf (COTS) network have become competitive on some classes of problems [4], [5].

Given a Beowolf-class machine, a cluster of workstations, and a high-end parallel machine, the problem is to find a way to run a single MPI program over all of these nodes. A solution to this problem must address several issues that arise when connecting heterogeneous nodes together as a single parallel machine. For example, each node may use a different format for storing numeric and character data so data must be converted between these formats when passing data. Another issue is program start-up, that is, how does a user start a program that is to run on a variety of machines.

One possible solution to this problem is to choose one of the publicly available implementations of MPI, such as MPICH [6] or LAM [2], and install it on all of the machines. These implementations of MPI are capable of handling the conversion of data between different formats and also have

The authors may be contacted at NIST, 100 Bureau Dr. Stop 8951, Gaithersburg MD 20899-8951. E-mail: william.george@nist.gov, john.hagedorn@nist.gov, judith.devaney@nist.gov

mechanisms for starting MPI programs on a heterogeneous set of nodes. A related effort, a *Globus-enabled* MPICH called MPICH-G [7], is underway which uses the Globus system [8] to connect multiple systems over the Internet to run MPI programs. The main drawback we see to these solutions is the expected loss in message passing performance, especially between homogeneous sets of nodes, due to abandoning the vendor supplied versions of MPI. The IMPI effort offers another solution.

The solution we describe here is to allow each set of homogeneous nodes to run under the best available implementation of MPI and only resort to special data handling when a message must pass between nodes running different implementations of MPI. This is a slight simplification since with IMPI it is also possible to run an MPI program on two or more identical parallel machines, both running the same implementation of MPI, which could not normally communicate with each other. The problem of program start-up is also addressed in IMPI.

III. How IMPI Works

A. Startup

There are many problems associated with initiating a single MPI job that spans multiple operating environments and multiple MPI implementations. Problems arise in areas such as differing operating environments, the mechanics of program startup, security, and scheduling. Projects such as GLOBUS [8] and Legion [9] are comprehensive efforts to solve many of these problems.

At this point in its development, IMPI is not attempting to address most of these issues. We are taking a very simple approach. We assume only that TCP/IP is available on at least one machine within each implementation universe. We do not assume that rsh is available or that UNIX (or any other operating system) is in use on any machine. This places a burden on the user to initiate each of the pieces that will fit together into the single interoperating MPI job.

These are the elements used in IMPI to achieve interoperation among multiple MPI implementations:

IMPI Server A rendezvous point for establishing communication among various IMPI Hosts

IMPI Client The initial point of contact that represents a collection of IMPI Hosts within a single MPI implementation.

IMPI Host A process that is responsible for a set of MPI processes within a single implementation. It handles all communications that must travel outside its set of processes.

It is anticipated that the initiation of each of these pieces will be accomplished by the use of an `mpirun` command that has been extended with IMPI functionality. We will use the command name `impirun` instead of `mpirun` to identify this extended command. The user will first start up the IMPI server with a command like this:

```
impirun -server <count>
```

where `<count>` is the number of client connections the server expects to see. This will start up the IMPI server and print out a host:port address of the TCP/IP socket on which the server is listening for client connections.

To start each client, the user will log onto an appropriate system and issue a command of this form:

```
impirun -client <clientRank> <host:port> <cmd>
```

where

`<clientRank>` is a sequence number identifying this client in the collection of clients that will be connecting to the server.

`<host:port>` is the host:port address given by the server.

`<cmd>` is the implementation-specific command line for launching MPI processes under the local implementation. In addition to starting up the client, this command will also start up the necessary IMPI hosts and MPI processes.

Each client then opens a socket connection to the server, executes an authentication protocol, then sends to the server a series of "commands". These commands describe

- the number of IMPI hosts the client represents,
- the number of MPI processes each IMPI host is handling,
- a host:port listening address for each host, and
- a variety of data transfer protocol parameters.

The server collects these data from each client and resends the collected data to all of the clients. In this way, each client is made aware of the information needed for communication among the hosts. It should be noted that the server plays a relatively passive role. It does not try to interpret the information from the clients; it simply receives, collates, and distributes this information.

When this exchange of information through, the server is complete. The hosts may then establish direct socket connections among themselves. It is possible that IMPI hosts that are operating within a common implementation may not need such a connection, but two hosts under different implementations require the socket connection enabled by this startup procedure. The protocol used over these connections is described in subsequent sections.

Note that the server and the clients are used only during the startup phase, while the hosts handle on-going communications throughout the execution of the application. When the startup procedure is complete, a single MPI job spanning multiple MPI implementations with a common MPI_COMM_WORLD has been launched.

B. Data Transfer

The data transfer protocol defines the communications protocol used to transfer data between two MPI implementations. The protocol is designed to support all of MPI. Where possible, the protocol is allows maximum flexibility for the implementor and also allows for future support of MPI-2 implementations.

All communication between implementations is Host-to-Host, over a dedicated channel, with the Hosts handling the multiplexing of messages to and from its processes. The channels between Hosts are assumed to be reliable ordered bidirectional streams which have a finite but unspecified amount of message buffering space. The Host handles the time-shared access to the communications channel by the processes, throttles messages from fast senders to avoid buffer overflows, and provides for the cancellation of messages.

The communications layer upon which the IMPI data transfer protocol is built is not specified, although it is expected that TCP/IP will be used by most implementations. This does not prevent the use of any appropriate communications technology when providing interoperability among a restricted set of IMPI implementations. Similarly, an implementation could support multiple underlying communications technologies and thus support interoperability with an expanded set of implementations.

Also, this protocol does not specify the interaction between the Host and its processes nor the communications medium used between them. Additionally, to support MPI-2 client/server functionality, no parent/child relationship is assumed between the Hosts and their processes.

In order to label the source and destination for each message, each process must be uniquely identified among all of the processes in an MPI job. This is accomplished with a decentralized mapping of each process to a unique *process identifier* (pid). First, each Host determines its unique ID. This will typically be the host's IP address although any identifier that is guaranteed to be unique among the Hosts is sufficient. Next, each process that is handled by that Host is assigned an ID that is guaranteed to be unique among that host's processes. The IMPI pid of a process is the combination of the Host ID and the process ID.

The basic unit of data transfer between processes is a packet. A packet consists of two parts: a fixed length header, and a variable sized payload. The header contains all the information needed to route the packet to the correct destination process and to correctly match the data to an MPI receive call or collective operation on the receiving side. The variable sized payload contains the data to be transferred.

At start-up, the Clients negotiate the value to be used as the maximum payload size, `IMPI_Pk_maxlen`. In effect, the Client that requests the smallest maximum payload size sets the maximum, under the condition that $1 \leq$ `IMPI_Pk_maxlen`. Any message larger that this maximum size is automatically broken-up into multiple packets, each of size `IMPI_Pk_maxlen` except possibly the last packet. These packets are sent in sequential order with the message being reconstructed on the receiving side. The protocol does not mandate at which point in the communication the message is reconstructed, such as in the receiving Host, the receiving process, or somewhere between. This is strictly implementation dependent and does not effect the IMPI protocol.

Support for message cancellation requires that each currently active message be uniquely identified. This identification must be in addition to the standard source, destination, context ID, and tag identifications used to match sends and receives. So, additionally each message is assigned a unique source request identifier which is part of each packet header. The exact message cancellation protocol is specified in the IMPI Standard.

There are several other *types* of packets other than those that are used to send data. To distinguish between these packet types, the header of each packet contains a field which identifies its *type*. The other packets types are:
CANCEL: used to request the cancellation of a message,
CANCELYES: used to respond that the request to cancel was successful,
CANCELNO: used to respond that the request to cancel was denied,
SYNCACK: (synchronization acknowledgment) used to signal the receipt of the first packet of a multi-packet message,
PROTOACK: (protocol acknowledgment) used to support the packet throttling protocol.
All of these packet types carry no data, that is, the payload size is zero.

The purpose of the throttling protocol is to prevent fast senders from overwhelming processes with packets faster than they can process them an therefore requiring ever increasing amounts of buffering space. The throttling protocol stops a sender that sends `IMPI_Pk_hiwater` packets to a process without receiving any intervening `PROTOACK` packets from the receiving process. On the receiving end, the throttling protocol requires that the receiving process sends one `PROTOACK` back to the sending process for each `IMPI_Pk_ackmark` packets it receives. The values for the `IMPI_Pk_hiwater` and `IMPI_Pk_ackmark` are negotiated at startup with the requirement that $1 \leq$ `IMPI_Pk_ackmark` $\leq$ `IMPI_Pk_hiwater`.

Several extra packet header fields are present for *quality-of-service* (QoS) support only and are not required to be used for correct IMPI operation. The QoS fields are:
pk_dtype: The data type of the elements in the message.
pk_count: Number of elements of type `dtype` in the message.
pk_seqnum: A unique sequence number for messages that can no longer be canceled.
These QoS fields can be used to support debuggers or other program development aids such as xmpi from LAM.

C. Collectives

Collective operations require communication among a group of MPI processes, all of which belong to the same communication context, or MPI communicator. When a communicator spans multiple MPI implementations, IMPI must define the pattern of communication that is required between processes in different implementations. Note that IMPI places no restrictions on the implementation of collective operations on communicators that do not span multiple MPI implementations. Thus the IMPI protocol should have no performance impact on such operations.

The object of the IMPI approach to collectives is to make the algorithms simple and robust. Given that communications between processes in different MPI implementations will typically be much slower that communication within an implementation, these collective algorithms attempt to minimize the number and size of IMPI managed messages where possible without making the algorithms overly complex. In general, simplicity was favored over optimizations although some effort was made to provide reasonable performance. In the future, the IMPI collective algorithms may further optimized.

The IMPI collective communication algorithms are built on top of the point-to-point communications defined in section III-B and on the collective operations within a given implementation. IMPI collective communication uses the concept of a *master process* for each participating MPI implementation. The master process is the lowest rank process within a given implementation and communicator. All IMPI-defined collective communications are between master processes; when data needs to be passed from one implementation to another, it is done via point-to-point communications between the master processes in each implementation.

Note that process rank 0 within a communicator is always a master process for its MPI implementation. This process plays a distinguished role in many of the IMPI collective communication algorithms.

The IMPI collective algorithms are expressed in a C-like pseudo-code. This pseudo-code is simply a tool for describing what should be accomplished within each implementation and the data that must be passed between implementations. Even though it must specify some operations that are entirely local to a single implementation, it is not intended to dictate how to implement these local operations.

For example, in the description of the collective algorithms, we use *local_comm* to refer to a logical intra-communicator that consists of all processes in a communicator that are local to a give MPI implementation. Some of the algorithm descriptions refer to collective operations such as a barrier or broadcast on local_comm. An implementation may choose to implement these via internal realizations of local_comm or by some other means.

The set of processes over which a collective operation occurs is determined by a communicator. In the IMPI protocol an integer context ID is associated with a communicator. This context ID must agreed upon by all participating implementations. Therefore, the creation of a new communicator requires a collective operation over the communicator from which the new one is being derived. The new context ID is generated by finding the maximum context ID currently in use across the existing communicator, then incrementing it by one. Note that this context ID, which is used in the IMPI protocol, need not be (and probably will not be) identical to any context ID representation used elsewhere in an MPI implementation. All that is necessary is that the implementation be able to map between the IMPI context ID and any other identifier that the implementation may use.

Let's look at the algorithms for two of the simpler collective operations. Figure 1 shows the algorithm for MPI_Barrier, one of the simplest of the collective operations. As mentioned above, this C-like pseudo-code used here is only intended to describe an algorithm, it is not intended as compilable code. The barrier operation begins with a barrier over the local_comm. Zero-length messages are then used to synchronize the master processes on each implementation. Finally there is another barrier of each local_comm.

As another example, consider the broadcast operation (figure 2). This algorithm uses point-to-point communication to move the data to each of the non-local master processes, then it uses the local broadcast operation to broadcast among the processes within the local_comm.

Most of the other collective algorithms are too lengthy for inclusion here. These two examples give a flavor for the approach. The complete set of algorithms can be found in the IMPI document [1].

IV. Testing

In an effort to assist the MPI development community, NIST has developed software to aid developers in testing conformance to the IMPI protocols. The tester does not try to test an MPI implementation against the MPI specification; it only tests that an IMPI implementation adheres to the defined IMPI protocols.

We will refer to the implementation of the IMPI protocols to be tested as the *Implementation Under Test* (IUT). The IUT will be running on a *System Under Test* (SUT). The SUT refers to all the hardware and software needed to run the IUT. The person running the tests will be called the *tester*.

The NIST IMPI test tool uses a web-based approach in conjunction with an interpreter that runs test scripts selected by the tester. This approach is used in order to simplify testing, to provide for easy modification and augmentation, and to reduce version control problems.

There are three major components of the test software:

Test Tool Applet This is the tester's primary interface to the test tool.

Test Interpreter This is an MPI program that is downloaded by the tester and linked to the IMPI implementation to be tested.

Test Manager A server that runs on a NIST host. This software coordinates the test software components and plays the role of IMPI server, client, host, and process during startup and test script execution.

The Test Manager provides a simulated IMPI implementation against which the IUT is tested. The Test Manager, while not being an MPI implementation, does implement the full IMPI communication protocol, and is carefully instrumented to detect and report errors in the protocol. Errors are reported from the Test Manager to the Test Tool Applet where they are displayed to the tester.

To perform tests using the NIST IMPI test tool, the tester must first download the Test Interpreter (from the NIST IMPI web site), compile it, and link it with the IUT. The test interpreter is a C program that that makes MPI calls. It does not embody any specific test scenarios in its code. Rather it interprets scripts that the tester selects during a test session. These test scripts are written in a language similar to C and make MPI calls.

Once the test interpreter has been downloaded, compiled, and linked, a test session proceeds as follows:

```
MPI_Barrier (comm)
{
   MPI_Barrier (local_comm);

   if (rank == 0) {
      for (i = each other master) {
         MPI_Recv (MPI_BOTTOM, 0, MPI_BYTE, i, IMPI_BARRIER_TAG, comm, &status);
      }

      for (i = each other master) {
         MPI_Send (MPI_BOTTOM, 0, MPI_BYTE, i, IMPI_BARRIER_TAG, comm);
      }
   }
   else if (rank is a master) {
      MPI_Send (MPI_BOTTOM, 0, MPI_BYTE, 0, IMPI_BARRIER_TAG, comm);
      MPI_Recv (MPI_BOTTOM, 0, MPI_BYTE, 0, IMPI_BARRIER_TAG, comm, &status);
   }

   MPI_Barrier (local_comm);
   return (MPI_SUCCESS);
}
```

Fig. 1. Algorithm for barrier

```
MPI_Bcast (buf, count, dtype, root, comm)
{
   if (rank == root) {
      for (i = each non-local master) {
         MPI_Send (buf, count, dtype, i, IMPI_BCAST_TAG, comm);
      }
      MPI_Bcast (buf, count, dtype, local_rank of root, local_comm);
   }
   else {
      if (local to root) {
         MPI_Bcast (buf, count, dtype, local_rank of root, local_comm);
      } else {
         if (master) {
            MPI_Recv (buf, count, dtype, root, IMPI_BCAST_TAG, comm, &status);
         }
         MPI_Bcast (buf, count, dtype, 0, local_comm);
      }
   }

   return (MPI_SUCCESS);
}
```

Fig. 2. Algorithm for broadcast

- The tester accesses the NIST IMPI web site with a Java-enabled browser and launches the Test Tool Applet.
- The applet opens a TCP/IP connection with the IMPI Test Manager running on the NIST IMPI web server. Test requests and results will be exchanged over this socket.
- Through the applet's interface the tester specifies the configuration of clients, hosts, and processes to be tested. This includes choosing where the IMPI server process will run.
- The tester initiates the startup protocol through the Test Tool Applet user interface. The applet displays a sample `impirun -client` command that includes the specific `host:port` needed.
- The tester then runs the IMPI Test Interpreter on the SUT using this command line or its equivalent. The startup protocol will be executed and its success or failure will be reported to the tester through the applet.
- If the startup protocol is successfully completed, the tester may then select and execute test scripts through the applet's interface.

• The test scripts are sent by the Test Manager to the Test Interpreter. The scripts are interpreted both by the Test Interpreter and by an interpreter embedded in the Test Manager. The IMPI communications required by the test script are monitored by the Test Manager for conformance to the protocol. Any problems are reported by the Test Manager to the Test Tool Applet where results are displayed to the tester.

Figure 3 shows this testing scheme graphically. The numbers in this figure indicate the order in which the communications channels are established initially. The dashed lines are HTTP Web communication (connection-less communications, stateless). The solid lines are connection-oriented TCP/IP sockets. The dotted line is direct tester interaction with the SUT.

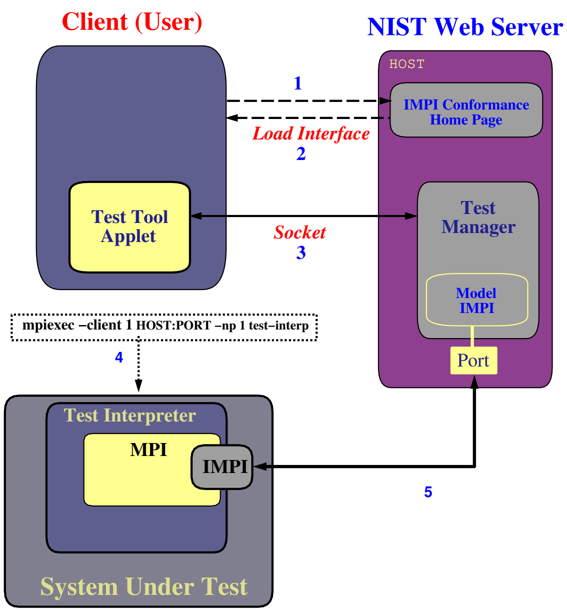

Fig. 3. IMPI Test Architecture

The Test Manager includes both an *upper-tester* and a *lower-tester* which monitor the execution of the test scripts. The upper-tester works at the level of the MPI calls to make sure the contents of the messages are correct. The lower-tester works at the level of the IMPI packets to make sure each packet header is valid. The placement of these testers in relation to the communication protocol stack is shown in figure 4.

This test scheme has several advantages, most importantly:

• It simplifies the delivery of the test capability to the tester.
• Tests are easily modified and added.
• Software configuration problems are minimized for the tester.

Note that the Test Interpreter is the only part of the test system that the tester must download and retain on their system. Because the test interpreter does not hard-wire any test cases it requires updates very infrequently. The actual test scenarios are embodied in the test scripts. These

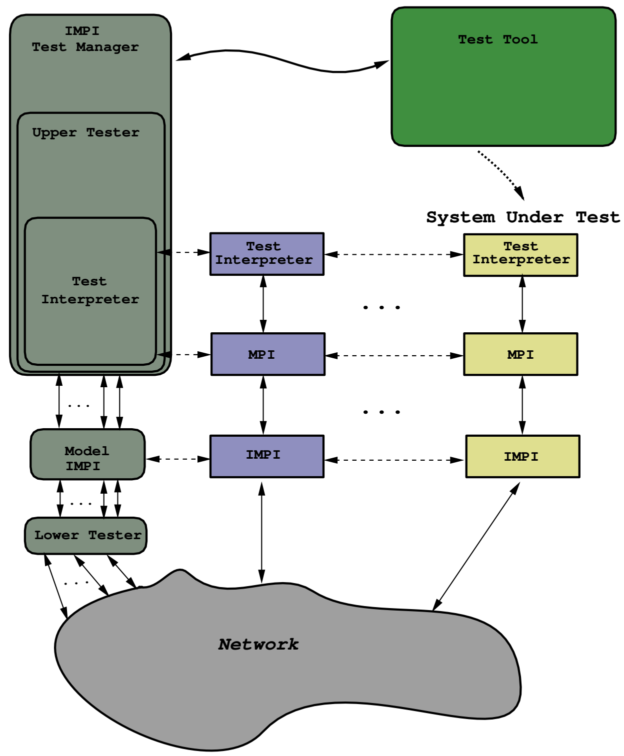

Fig. 4. IMPI Protocol Stack

can be modified, deleted, and new test scripts can be added without revision to the Test Interpreter. Similarly, modifications can be made to the Test Applet and Test Manager without bother to the tester. Each time the tester connects to the NIST IMPI Test Tool web site, they gain access to the newest version of the software. By making the Test Interpreter a relatively generic interpreter of MPI scripts we minimize the headaches associated with distributing and documenting software and the version conflicts that can occur.

V. Conclusions and Future Work

Interoperable MPI provides a capability that is critical for the development of high-performance applications across heterogeneous system. The design of the IMPI protocols enables MPI developers to add interoperability without hurting performance within an implementation.

From the user's point of view, IMPI has minimal impact at the application level. The MPI library calls retain their syntax and semantics; no changes to an MPI application are required when porting to a heterogeneous computing environment using IMPI.

Future development of the IMPI protocol may occur in the following areas:

• Improved security features.
• Optimized algorithms for collective operations.
• Automated methods for starting up cross-platform applications.
• Support of MPI-2 functionality.

Although Interoperable MPI does not solve all of the problems faced when constructing parallel applications to be run across heterogeneous computing resources, we feel that it supplies an important piece to the puzzle.

Disclaimer

Certain commercial equipment and software may be identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment or software is necessarily the best available for the purpose.

References

[1] IMPI Steering Committee, "Impi - interoperable message-passing interface," 1998, ftp://ftp.nist.gov/pub/hpss/interop/impi-report.current.ps.

[2] D. Burns and Raja B. Daoud, "Lam: An open cluster environment for mpi," in *Supercomputing Symposium '94*, June 1994, Toronto, Canada, code available at http://www.lsc.nd.edu/lam/.

[3] Message Passing Interface Forum, "MPI: A message-passing interface standard," *International Journal of Supercomputing Applications*, vol. 8, no. 3/4, 1994.

[4] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling, "Beowolf: Harnessing the power of parallelism in a pile-of-pcs," *Proceedings, IEEE Aerospace*, 1997.

[5] J. Salmon and C. Stein, "Scaling of beowolf-class distributed systems," *SC98 Proceedings*, November 1998, http://beowulf.gsfc.nasa.gov/papers/papers.html.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept. 1996.

[7] I. Foster and N. Karonis, "A grid-enabled mpi: Message passing in heterogeneous distributed computing systems," *SC98 Proceedings*, November 1998.

[8] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *Intl J. Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, Summer 1997.

[9] A. Grimshaw and W. Wulf, "Legion - a view from 50,000 feet," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.