A Message Passing Standard for MPP and Workstations

Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker

THE Message Passing Interface (MPI) is a portable message-passing standard that facilitates development of parallel applications and libraries. MPI defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77 or C. The standard also forms a possible target for such language compilers as High Performance Fortran [7]. Commercial and free, public-domain implementations of MPI have been available since 1994 (see the sidebar, "MPI Implementations"), running on both tightly coupled, massively parallel processing (MPP) machines and on networks of workstations (NOWs).

The MPI standard was developed over a 12-month period in 1993-1994 of intensive meetings involving more than 80 people from approximately 40 organizations, mainly from the U.S. and Europe. The meetings were announced on various bulletin boards and mailing lists and were open to the technical community. The MPI meetings operated on a tight budget (actually no budget when the first meeting was announced). DARPA provided partial travel support for U.S. academic participants through the National Science Foundation. Support for several European participants was provided by the European Commission through its Esprit program. Formal voting at the meetings was by a single vote per organization; in order to vote, an organization needed to have had at least one representative at two of the last three meetings. To provide guidance for preparing formal proposals, frequent informal votes including all those present were held. Many vendors of concurrent computers were involved, as were researchers from universities, government laboratories, and industry.

This effort culminated in the 1994 publication of the MPI specification [8]. Other sources of information on MPI are available [10] or are under development (see the sidebar, "More MPI Assistance").

Researchers incorporated into MPI the

The MPI standard defines a core of communication library routines for the processes forming concurrent programs on MPP machines and on networks of workstations.

most useful features of several systems, rather than choosing a single system as the standard. MPI has roots in PVM [4, 5], Express [9], P4 [1], Zipcode [10], and PARMACS [2], and in systems sold by IBM, Intel, Meiko Scientific, Cray Research, and nCube.

MPI is used to specify the communication among a set of processes forming a concurrent program. The message-passing paradigm is attractive because of its wide portability and scalability. It is easily compatible with both distributed-memory multicomputers and shared-memory multiprocessors, with NOWs, and with combinations of these elements. Message passing will not be made obsolete by increased network speeds or by architectures combining shared and distributed-memory components.

Though much of MPI standardizes the common practice of existing message-passing systems, MPI goes further to define such advanced features as user-defined datatypes, persistent communication ports, powerful collective communication operations, and scoping mechanisms for communication. No previous system incorporated all these features.

In considering MPI, it is important to understand the constraints implied by such an endeavor and the practical constraints under which the committee operated, as well as the following goals:

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication, avoiding memory-to-memory copying and supporting overlap of computation and communication and offload to communication coprocessors where available.
- Allow implementations in heterogeneous environments.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface so the user need not cope with communication failures dealt with by the underlying communication subsystem.
- Define an interface not too different from current practice, such as Express, Intel's NX, PVM, and P4, while providing extensions allowing greater flexibility.
- Define an interface that can be implemented on many vendors' platforms with no significant changes in the underlying communication and system software.
- Design interface semantics to be language independent.
- Design the interface to allow for thread safety.

What MPI Does and Does Not Specify

The standard specifies the form of the following:

- *Point-to-point communications.* Messages between pairs of processes.
- *Collective communications.* Communication or synchronization operations involving entire groups of processes.
- *Process groups.* How process groups are used and manipulated.
- *Communicators.* A mechanism for providing separate communication scopes for modules or libraries. Each communicator specifies a distinct name space for processes and a distinct communication context for mes-

MPI Implementations

MPI is available on parallel computers from Convex Computer, Cray Research, IBM, Intel, Meiko Scientific, nCube, NEC, and Silicon Graphics. A number of public-domain MPI implementations are available and can be found at the following locations:

• Argonne National Laboratory/Mississippi State University implementation. Available by anonymous ftp at info.mcs.anl.gov/pub/mpi.

• Edinburgh Parallel Computing Centre CHIMP implementation. Available by anonymous ftp at ftp.epcc.ed.ac.uk/pub/chimp/release/chimp.tar.Z.

• Mississippi State University UNIFY implementation. The UNIFY system provides a subset of MPI within the PVM environment without sacrificing the PVM calls already available. Available by anonymous ftp at ftp.erc.msstate.edu/unify.

• Ohio Supercomputer Center LAM implementation. A full MPI standard implementation for LAM, a Unix cluster computing environment. Available by anonymous ftp: tbag.osc.edu/pub/lam.

More MPI Assistance

The book by W. Gropp, E. Lusk, and A. Skjellum [6] is a tutorial-level explanation of MPI. An expanded and annotated reference manual for MPI is the book *MPI: The Complete Reference* by Snir, Otto, Hess-Lederman, Walker, and Dongarra [11].

An MPI-specific newsgroup is accessible through comp.parallel.mpi, and an abundance of information about MPI is available through the World-Wide Web. The following list includes URLs containing MPI-related information:

- Netlib Repository at the University of Tennessee and Oak Ridge National Lab, http://www.netlib.org/mpi/index.html
- Argonne National Lab, http://www.mcs.anl.gov/mpi
- Mississippi State University, Engineering Research Center, http://www.erc.msstate.edu/mpi
- Ohio Supercomputer Center, LAM Project, http://www.osc.edu/lam.html
- Australian National University, file://dcssoft.anu.edu.au/pub/www/dcs/cap/mpi/mpi.html

A current version of errata for the specification document [8] is available at ftp://www.netlib.org/mpi/errata.ps. The complete email associated with the MPI Forum is archived in netlib. Send a message to netlib@ornl.gov with the message "send index from mpi". You can also get them via ftp from netlib2.cs.utk.edu/mpi. At least one software vendor, PALLAS, in Brühl, Germany, which specializes in high-performance computing, offers professional support and consulting for MPI; see info@pallas-gmbh.de.

sages; it may also carry additional scope-specific information.

- *Process topologies.* Functions that allow convenient manipulation of process labels when the processes are regarded as forming a particular topology, such as a Cartesian grid.
- *Bindings for Fortran 77 and ANSI C.* MPI was designed so that versions of it in both C and Fortran had straightforward syntax. In fact, the detailed form of the interface in these two languages is specified and is part of the standard.
- *Profiling interface.* The interface is designed so that run-time profiling or performance-monitoring tools can be joined to the message-passing system. It is not necessary to have access to the MPI source to do performance monitoring and instrumentation; hence portable profiling systems are easily constructed.
- *Environmental management and inquiry functions.* These functions provide a portable timer, some system-querying capabilities, and the ability to influence error behavior and error-handling functions.

However, many relevant issues of parallel programming are not covered by the standard, including:

- Shared-memory operations
- Interrupt-driven messages, remote execution, and active messages
- Program construction tools
- Debugging support
- Thread support
- Process or task management
- Input and output functions

The main reasons for not addressing these issues were the committee's self-imposed time constraint and by the view that many of them are system dependent. Meetings focusing on extending MPI began in 1995 and will end in 1997. The rest of this article discusses some of MPI's more interesting features, including point-to-point communication, user-defined datatypes, collective communications, and groups, contexts, and communicators.

Point-to-Point Communication

MPI provides a set of send and receive functions that allow the communication of typed data with an associated tag. Typing of the message contents is necessary for the heterogeneous support needed for the performance of correct data representation conversions as data is sent from one architecture to another architecture. The tag allows selectivity of messages at the receiving end; a user's program can receive on a particular tag, or the program can wildcard, or self-define, this quantity, allowing reception of messages with any tag. Also provided is message selectivity on the source process of the message.

The fragment of code in Figure 1 for the example of process 0 sending a message to process 1 executes on both process 0 and process 1, sending a character string. `MPI_COMM_WORLD` is a default communicator provided upon startup. Among other things, a communicator defines the allowed set of processes involved in a communication operation. Process ranks are integers, serving to label processes and are discovered through inquiry to a communicator (see the call to `MPI_Comm_rank()` in Figure 1). The typing of the communication is evident by the specification of `MPI_CHAR`. The receiving process specified that the incoming data was to be placed in `msg` and that it had a maximum size of 20 elements of type `MPI_CHAR`. The variable status, set by `MPI_Recv()`, gives information on the source and tag of the message and the number of elements actually received. For example, the receiver can examine this variable to find out the actual length of the character string received.

This example employs blocking send and receive functions. The send call blocks until the send buffer can be reclaimed; that is, after the send, process 0 can safely overwrite the contents of `msg`. Similarly, the receive function blocks until the receive buffer actually contains the contents of the message.

MPI also provides nonblocking send and receive functions that allow the overlap of message transmittal with computation or the overlap of multiple message transmittals with one another. Nonblocking functions always come in two parts: the posting functions, which begin the requested operation, and the test-for-completion functions, which allow the application program to discover whether the requested operation has been completed.

While this description may already seem like rather a lot to say about a simple transmittal of data from one process to another, there is more. To understand why, we examine two aspects of the communication: the semantics of the communication primitives and the underlying protocols that implement them. Consider the earlier example on process 0 after the blocking send has completed. If the send has completed, does this tell us anything about the receiving process? Can we know that the receive has finished, or even that it has begun?

Such questions of semantics are related to the nature of the underlying protocol implementing the operations. If a user wishes to implement a protocol minimizing the copying and buffering of data, the most natural semantics might be the rendezvous version, in which completion of the send implies the receive has been initiated (at least). On the other hand, a protocol that attempts to block processes for the minimal amount of time necessarily ends up doing more buffering and copying of data.

The trouble is that one choice of semantics is not best for all applications, nor is it best for all architectures. Because MPI's primary goal is to standardize message-passing operations yet not sacrifice performance, the decision was made to include all the major choices for point-to-point semantics in the standard.

An additional complicating factor is that the amount of space available for buffering is always finite; on some systems, that space may be small or nonexistent. Therefore, MPI does not mandate a minimal amount of buffering, and the standard is very careful about the semantics it requires.

These complexities are manifested in MPI by modes for point-to-point communication. Both blocking and nonblocking communications have modes. The mode allows the user to choose the semantics of the send operation and, in effect, to influence the underlying protocol for the transfer of data.

In standard mode, the completion of the send does not necessarily mean that the matching receive has started, and no assumption should be made in the application program about whether the outgoing data is buffered by MPI. In buffered mode, the user can guarantee that a certain amount of buffering space is available. The catch is that the space must be explicitly provided by the application program. In synchronous mode, a rendezvous semantics between sender and receiver is used. Finally, ready mode allows the user to exploit extra knowledge to simplify the protocol and potentially achieve higher performance. In a ready-mode send, the user asserts that the matching receive is already posted.

Figure 1. C code. Process 0 sends a message to process 1.

```
/* static variable used as "key" for library */
/* Only one per process is necessary, even if multiple */
/* library invocations can be concurrently active. */
extern int lib_key;
/* library init. Need to invoke once by each process, */
/* before library is used. */
void lib_init ()
{
/* allocate a process-unique key */
MPI_Keyval_create(MPI_NULL_FN, MPI_NULL_FN, &lib_key, (void *)NULL);
}
void lib_call( MPI_Comm comm, … )
{
int flag;
/* private communicator for library-internal communication */
MPI_Comm *private_comm;
/* retrieve private communicator */
MPI_Attr_get( comm, lib_key, &private_comm, &flag );
if (!flag) {
/* get failed; this is first call and private_comm */
/* has not yet been allocated. So, do it. */
/* Make new communicator, with same process group as comm. */
private_comm = (MPI_Comm *)malloc(sizeof(MPI_Comm));
MPI_Comm_Dup( comm, private_comm );
/* Cache private communicator with public one. */
MPI_Attr_put( comm, lib_key, (void *)private_comm );
}
/* Execute library code, using private_comm for */
/* internal communication. */
…
}
char msg[20] ;
int myrank, tag = 99;
MPI_Status status;
…
MPI_Comm_rank( MPI_COMM_WORLD, &myrank ); /* find my rank */
if (myrank == 0) {
strcpy( msg, "Hello there");
MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1,  tag, MPI_COMM_WORLD);
} else {
MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status );
}
```

User-Defined Datatypes

All MPI communication functions take a datatype argument. In the simplest case, the datatype is a primitive type, such as an integer or floating-point number. An important and powerful generalization results from allowing user-defined types wherever the primitive types can occur. These are not "types" as far as the programming language is concerned. They are types only in that MPI is made aware of them through type-constructor functions describing the layout in memory of sets of primitive types. Through user-defined types, MPI supports communication of complex data structures, such as array sections and structures containing combinations of primitive datatypes. Fig-

ure 2 gives an example of using a user-defined type to send the upper-triangular part of a matrix.

Collective Communications

Collective communications transmit data among all the processes specified by a communicator object. One function, the barrier, serves to synchronize processes without passing data. MPI provides the following collective communication functions:

- Barrier synchronization across all processes
- Broadcast from one process to all
- Gathering data from all processes to one
- Scattering data from one to all
- Allgather, like a gather followed by a broadcast of the gather output
- Alltoall, like a set of gathers in which each process receives a distinct result
- Global reduction operations, such as sum, max, min, and user-defined functions
- Scan (or prefix) across processes

Figure 3 shows broadcast, scatter, gather, allgather, and alltoall. Many of the collective functions also have "vector" variants, whereby different amounts of data can be sent to or received from different processes. For these vector variants, the simple grids in Figure 3 become more complex.

The syntax and semantics of the MPI collective functions were designed to be consistent with point-to-point communications. However, to keep the number of functions and their argument lists at a reasonable level of complexity, the MPI committee made collective functions more restrictive than the point-to-point functions in several ways. One restriction is that, in contrast to point-to-point communication, the amount of data sent must exactly match the amount of data specified by the receiver. This restriction was imposed to avoid the need for an array of status variables as an argument to the functions that would otherwise be necessary for the receiver to discover the amount of data actually received.

A major simplification is that collective functions come only in blocking versions. Though a standing joke at committee meetings concerned the nonblocking barrier, such functions can be quite useful[1] and may be included in a future version of MPI.

A final simplification of collective functions concerns modes, which come in only one type that may be regarded as analogous to the standard mode of point-to-point. Specifically, the semantics are: A call collective function (on a given process) can return to the user's program as soon as its participation in the overall communication is complete. As usual, the completion indicates that the caller is now free to access and modify locations in the communication buffer. The completion does not indicate that other processes have completed—or even started—the operation.

```
double a[100][100];
int disp[100] ,blocklen[100] , i;
MPI_Datatype upper;
…
/* compute start and size of each row */
for (i=0; i<100; ++i) {
disp[i] = 100 * i + i;
blocklen[i] = 100 – i;
}
/* create datatype for upper triangular part */
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit( &upper );
/* . . and send it */
MPI_Send( a, 1, upper, dest, tag, MPI_COMM_WORLD );
```

Figure 2. Using a user-defined datatype, a single send transmits the upper-triangular part of a matrix.

Thus, a collective communication may or may not have the effect of synchronizing all calling processes. The barrier, of course, is the exception to this statement.

These semantics were chosen to allow a variety of implementations. And the MPI user must keep these issues in mind. For example, even though a particular implementation of MPI may provide a broadcast with the side effect of synchronization (the standard allows this side effect), the standard does not require it, and hence, any program that relies on synchronization is nonportable. On the other hand, a correct and portable program must allow a collective function to be synchronizing. Though one should not rely on synchronization side effects, one must program to allow for them.

Though these issues and statements may seem unusually obscure, they are merely a consequence of the desire of MPI to do two things:

- Allow efficient implementations on a variety of architectures
- Be clear about exactly what is and what is not guaranteed by the standard

Groups, Contexts, and Communicators

A key feature needed to support robust, parallel libraries is a guarantee that communication within a library routine does not conflict with communication extraneous to the routine. The concepts encapsulated by an MPI communicator provide this support.

A communicator is a data object that specifies the scope of a communication operation, that is, the group of processes involved and the communication context. Contexts partition the communication space. A message sent in one context cannot be received in another context. Process ranks are interpreted with respect to the process group associated with a communicator. MPI applications begin with a default communicator, `MPI_COMM_WORLD`, which has as its process group the entire set of processes (of

[1]The nonblocking barrier would, of course, block at the test-for-completion call.

this parallel job). New communicators are created from existing communicators; creation of a communicator is a collective operation.

Communicators are especially important for the design of parallel software libraries. Suppose we have a parallel matrix-multiplication routine as a member of a library. We would like to allow distinct subgroups of processes to concurrently perform different matrix multiplications. A communicator provides a convenient mechanism for passing into the library routine the appropriate group of processes, and within the routine, process ranks are interpreted relative to this group. The grouping and labeling mechanisms provided by communicators are useful, and communicators are typically passed into library routines that perform internal communications.

Such library routines can also create their own unique communicators for internal use. For example, consider an application in which process 0 posts a wildcarded, nonblocking receive just before entry to a library routine. Such "promiscuous" posting of receives is a common technique for increasing performance. If an internal communicator is not created, incorrect behavior may result, since the receive may be satisfied by a message sent by process 1 from within the library routine—if process 1 invokes the library ahead of process 0. Another example is a case in which a process sends a message before entry into a library routine, but the destination process does not post the matching receive until after it exits the library routine. In this case, the message may be received incorrectly within the library routine.

These problems are avoided by proper design and use of parallel libraries. One workable design is for the application program to pass communicators into the library routine specifying the group and ensuring a safe context. Another design has the library create a "hidden" and unique communicator in a library initialization call, again leading to correct partitioning of the message space between application and library.

The sidebar, "Library Communicator and Caching," reveals how one might implement the second type of design. Some thought reveals that as one creates separate communicators for libraries, it is convenient to associate these new communicators with the old communicators from which they were derived. The MPI caching mechanism provides a way to set up such an association. Though one can associate arbitrary objects with communicators using caching, the ability to perform such associations for library internal communicators is one of the most important uses of caching.

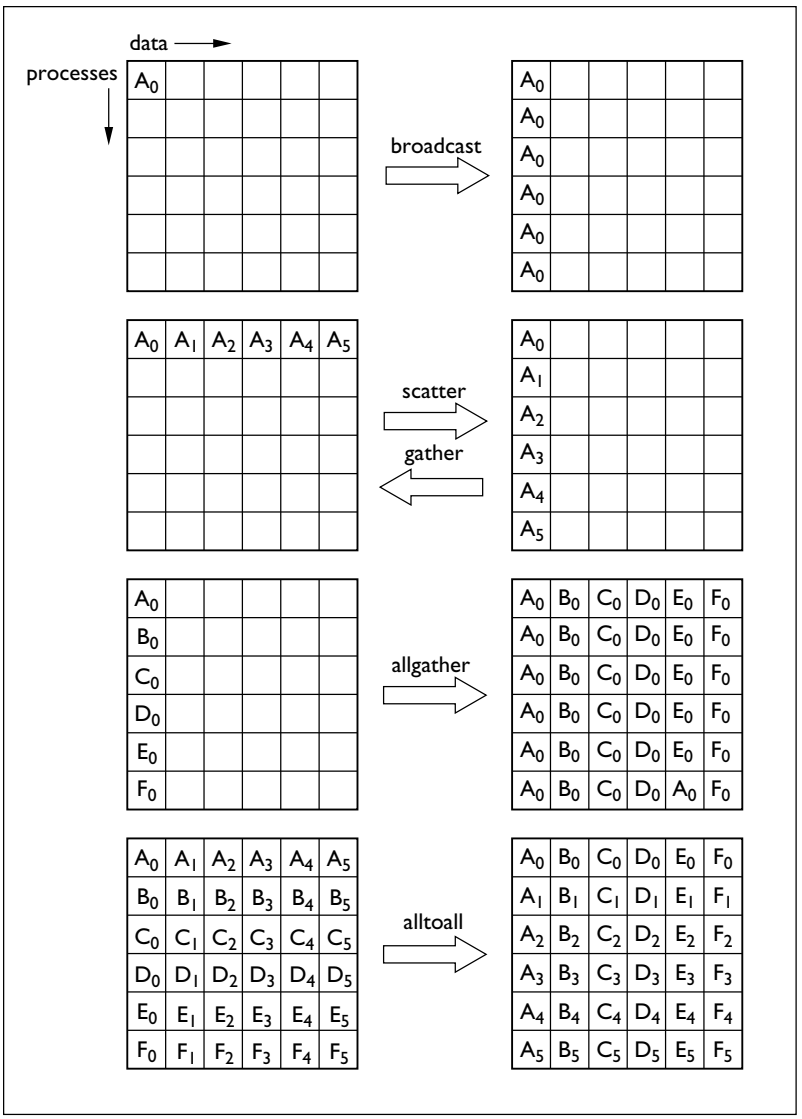

Figure 3. Collective move functions for a group of six processes. For each process, each row of boxes represents data locations in one process. Thus, in the broadcast, initially only the first process contains the data A_0, but after the broadcast all processes contain the data.

Conclusions

A pleasant surprise for participants in the MPI effort was the interesting intellectual issues that kept coming up. This article has concentrated on some of them, but for most cases, programming in MPI is straightforward and similar to programming with other message-passing interfaces.

MPI does not claim to be the definitive answer to all message-passing needs. Indeed, our insistence on simplicity and timeliness of the standard precludes MPI being the definitive passing system. The MPI interface provides a useful basis for developing software for message-passing environments. Besides promoting the

Library Communicator and Caching

We wish to give a parallel library its own communicator with a unique context. The strategy is to pass in—at each invocation of the library—a communicator that describes the process group to be used. The library function "duplicates" the process group, getting a similar communicator but one with a unique communication context. This context becomes the private, library-internal communicator.

The MPI caching mechanism is used to make the process of attaching arbitrary pieces of information work well. The private communicator is associated (cached) with the communicator passed in by the application. This association means that the private communicator needs to be created only the first time the library is invoked with that particular communicator as argument. The caching hides the internal communicator from the application, so the application need not explicitly manage the internal communicators. See the sources in the second sidebar for details concerning the caching mechanism.

emergence of parallel software, a message-passing standard gives vendors a clearly defined base set of routines they can implement efficiently. Hardware support for parts of the system is also possible and may greatly enhance parallel scalability.

The final MPI 1 Forum meeting in February 1994 decided that plans for extending MPI should wait for more experience with the current version. The MPI 2 Forum now under way is moving in the following directions:

- Parallel input/output
- Remote store/access
- Active messages
- Process startup
- Dynamic process control
- Nonblocking collective operations
- Fortran 90 and C++ language bindings
- Graphics
- Real-time support

For more information, see the MPI-specific newsgroup at comp.parallel.mpi. The official version of the specification document can be obtained from netlib [3] by sending an email message to netlib@www.netlib.org with the message: "send mpi-report.ps from mpi". A postscript file will be mailed back to you by the netlib server. The document may also be obtained via anonymous ftp from www.netlib.org/mpi/mpi-report.ps. A hypertext version is available on the World-Wide Web at http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html. ◻

References

1. Butler, R., and Lusk, E. Monitors, messages, and clusters: The P4 parallel programming system. *Parallel Comput. 20* (April 1994), 547–564.
2. Calkin, R., Hempel, R., Hoppe, H., and Wypior, P. Portable programming with the PARMACS Message-Passing Library. *Parallel Comput., Special Issue on Message-Passing Interfaces 20* (April 1994), 615–632.
3. Dongarra, J., and Grosse, E. Distribution of mathematical software via electronic mail. *Commun. ACM 30*, 5 (July 1987), 403–407.
4. Dongarra, J., Geist, A., Manchek, R., and Sunderam, V. Integrated PVM framework supports heterogeneous network computing. *Comput. Phys. 7*, 2 (April 1993), 166–175.
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994. The book is available electronically; see ftp://www.netlib.org/pvm3/book/pvm-book.ps.
6. Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Mass., 1994.
7. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Jr., and Zosel, M. *The High Performance Fortran Handbook*. MIT Press, Cambridge, Mass., 1994.
8. Message Passing Interface Forum. MPI: A message-passing interface standard. *Int. J. Supercomput. Appl. and High Performance Comput., Special Issue on MPI 8*, 3/4 (1994). Also see ftp://www.netlib.org/mpi/mpi-report.ps.
9. Parasoft Corp., Monrovia, Calif. *Express User's Guide, Version 3.2.5*, 1992. See parasoft@parasoft.com.
10. Skjellum, A., and Leung, A. Zipcode: A portable multicomputer communication library atop the reactive kernel. In *Proceedings of the 5th Distributed Memory Concurrent Computing Conference*, D.W. Walker and Q.F. Stout, eds. (Charleston, S. Car, Apr. 1990) IEEE Press, 1990, pp. 767–776.
11. Snir, M., Otto, S., Hess-Lederman, S., Walker, D., and Dongarra, J., *MPI: The Complete Reference*. MIT Press, Cambridge, Mass., 1996. Available electronically; see http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

About the Authors:

JACK J. DONGARRA holds a joint appointment as Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and as Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory under the UT/ORNL Science Alliance Program. **Author's Current Address:** Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301; email: dongarra@cs.utk.edu and http://www.netlib.org/utk/people/jackdongarra.html

STEVE W. OTTO is on the faculty of the Computer Science and Engineering Department of the Oregon Graduate Institute and also recently joined Intel Development Laboratories. **Author's Current Address:** Oregon Graduate Institute, Computer Science Department, 19600 NW Von Neumann Drive, Beaverton, OR 97006-1999; email: otto@cse.ogi.edu

MARC SNIR is a senior manager at the IBM T. J. Watson Research Center. **Author's Current Address:** IBM T. J. Watson Research Center, Route 134, P.O. Box 218, Yorktown Heights, NY 10598; email: snir@watson.ibm.com

DAVID WALKER is a senior research staff member in the Mathematical Sciences Section at Oak Ridge National Laboratory and an Adjunct Associate Professor in the Department of Computer Science at the University of Tennesseee, Knoxville. **Author's Current Address:** Building 6012, MS-6367, P. O. Box 2008, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367; email: walker@rios2.epm.ornl.gov