MPI on the I-WAY: A Wide-Area, Multimethod Implementation of the Message Passing Interface

Ian Foster, Jonathan Geisler, Steven Tuecke
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
{foster,geisler,tuecke}@mcs.anl.gov

Abstract

High-speed wide-area networks enable innovative applications that integrate geographically distributed computing, database, graphics, and networking resources. The Message Passing Interface (MPI) can be used as a portable, high-performance programming model for such systems. However, the wide-area environment introduces challenging problems for the MPI implementor, because of the heterogeneity of both the underlying physical infrastructure and the authentication and software environment at different sites. In this article, we describe an MPI implementation that incorporates solutions to these problems. This implementation, which was developed for the I-WAY distributed-computing experiment, was constructed by layering MPICH on the Nexus multithreaded runtime system. Nexus provides automatic configuration mechanisms that can be used to select and configure authentication, process creation, and communication mechanisms in heterogeneous systems.

1. Introduction

The I-WAY networking experiment [4] provided the largest testbed developed to date for high-performance distributed computing. Over sixty groups used this testbed to develop applications that connected supercomputers, advanced display devices, storage systems, and/or scientific instruments located across North America. Many of these applications used an implementation of the standard Message Passing Interface (MPI) for process creation, communication, and synchronization. In this article, we describe the techniques used to develop this MPI implementation.

Our goal in developing an MPI implementation for the I-WAY was to shield programmers from such low-level details as the authentication mechanisms, process startup mechanisms, network interfaces, and communication protocols to be used at different sites. That is, we wished to allow programmers to allocate a heterogeneous collection of resources spanning multiple sites and then start an MPI program on these resources by typing a single command. Achieving this degree of transparency is challenging for two principal reasons. First, it requires a low-level infrastructure that supports and permits the coexistence of multiple implementations of low-level mechanisms for authentication, process creation, communication, and so forth. Second, it requires access to correct, up-to-date information about the software and hardware environment at different sites.

We addressed these two challenges by layering the Argonne/Mississippi State MPICH library [14] on top of a runtime library called Nexus [10]. MPICH provides a portable, high-performance implementation of MPI that incorporates some support for heterogeneous environments, but that in its current instantiation is designed primarily for homogeneous massively parallel processing (MPP) systems. Nexus is a portable, multithreaded communication library that we have constructed to support wide-area, heterogeneous computations. Nexus is distinguished from MPI by its support for dynamic resource management, a global address space via global pointers, and a single-sided communication mechanism called a remote service request. In addition, its implementation incorporates automatic configuration mechanisms that allow it to use information contained in resource databases to determine which startup mechanisms, network interfaces, and communication methods to use in different situations. These mechanisms were not designed explicitly to support the MPI communication model but, as we ex-

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

plain in this paper, can be used to construct a high-performance, multithreaded MPI implementation.

In this article, we first provide an overview of the I-WAY experiment and the software environment, I-Soft, that was developed to support application development on the I-WAY. Then, we introduce Nexus and the techniques that it uses to support multimethod communication. Subsequent sections describe the Nexus implementation of MPI and the techniques used to support automatic configuration of MPI computations on the I-WAY.

2. The I-WAY Experiment

The I-WAY, or Information Wide Area Year [4], was a wide-area computing experiment conducted throughout 1995 with the goal of providing a large-scale testbed in which innovative high-performance and geographically distributed applications could be deployed. The I-WAY linked eleven existing national testbeds based on ATM (asynchronous transfer mode) technology to interconnect supercomputer centers, virtual reality research locations, and applications development sites across North America. When demonstrated at the Supercomputing conference in San Diego in December 1995, the I-WAY network connected multiple high-end display devices (including immersive CAVE^{TM} and ImmersaDesk^{TM} virtual reality devices [3]); mass storage systems; specialized instruments (such as microscopes and satellite downlinks); and supercomputers of different architectures, including distributed-memory multicomputers (IBM SP, Intel Paragon, Cray T3D, etc.), shared-memory multiprocessors (SGI Challenge, Convex Exemplar), and vector multiprocessors (Cray C90, Y-MP). These devices were located at seventeen different sites across North America.

The I-WAY distributed supercomputing environment was used by over sixty application groups for experiments in high-performance computing (e.g., [18]), collaborative design, and the coupling of remote supercomputers and databases into local environments (e.g., [17]). A primary thrust was applications that use multiple supercomputers and virtual reality devices to explore collaborative technologies in which shared virtual spaces are used to perform computational science. For simplicity, the I-WAY standardized on the use of TCP/IP running over ATM Adaptation Layer 5 (AAL5) for application networking; in future experiments, alternative protocols will undoubtedly be explored. The need to configure both IP routing tables and ATM virtual circuits in this highly heterogeneous environment was a significant source of implementation complexity.

An innovative aspect of the I-WAY project was the development of a system management and application programming environment called I-Soft [8] that provided uniform authentication, resource reservation, process creation, and communication functions across I-WAY resources. A novel aspect of this approach was the deployment of a dedicated I-WAY Point of Presence, or I-POP, machine at each participating site. These machines provided a uniform environment for deployment of management software and also simplified validation of system management and security solutions by serving as a "neutral" zone under the joint control of I-WAY developers and local authorities.

3. Nexus

We next give a brief introduction to the Nexus communication library used to construct the I-WAY implementation of MPI. Nexus provides a low-level interface to multithreading and communication mechanisms in homogeneous and heterogeneous systems. It is designed for use by library writers and compiler writers; in addition to MPI, systems that use Nexus facilities include parallel object-oriented languages (for example, CC++ [2] and Fortran M [6]), parallel scripting languages (nPerl [11]), and communication libraries (CAVEcomm [5] and a Java library).

3.1. Nexus overview

Nexus is structured in terms of five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called—confusingly for MPI users—a *context*. (For the purposes of this article, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. The *global pointer* (GP) provides a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. A GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between nodes by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint. The remote service request mechanism allows point-to-point communication, remote memory access, and streaming protocols to be supported within a single framework.

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

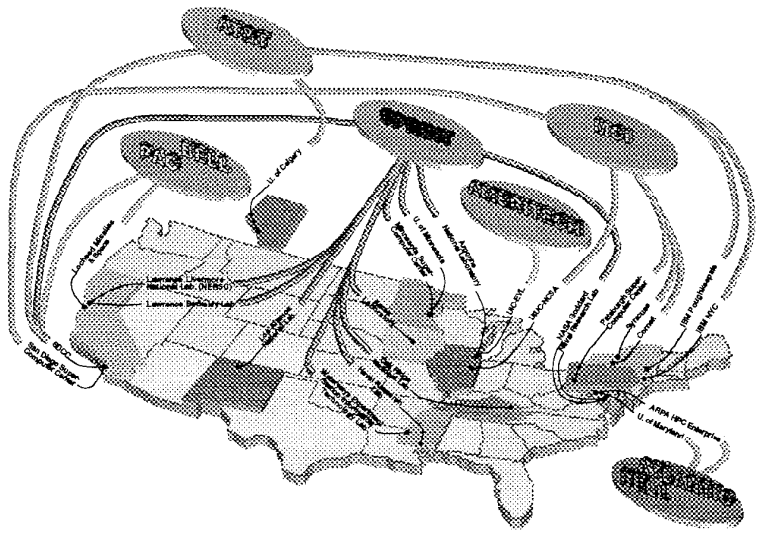

Figure 1. The I-WAY network. Figure produced by Linda Winkler and Richard Foster.

3.2. Multimethod communication

From the point of view of the I-WAY, the Nexus features that are most interesting are those that support multimethod communication [7]. These mechanisms are based around the global pointer construct, which is used to maintain information about the methods that can be used to perform communications directed to a particular remote location. Simple protocols allow this information to be propagated from one node to another and provide a framework that supports both automatic and manual selection from among available communication methods.

Nexus incorporates automatic configuration mechanisms that allow it to use information contained in resource databases to determine which startup mechanisms, network interfaces, and communication methods to use in different situations. These mechanisms allow Nexus programs to execute unchanged in different environments, with communication methods selected according to default rules, depending on the source and destination of the message being sent. For example, automatic selection in Nexus RSRs being performed with IBM's Message Passing Library (MPL) within an IBM SP2 and with TCP/IP between computers. Manual selection is also supported, for example allowing selection of specialized ATM protocols when appropriate.

Automatic configuration makes sense only if resource databases contain up-to-date information. We discuss below the techniques used to create and maintain resource databases in the I-WAY.

4. The I-WAY Implementation of MPI

We now address the question of how Nexus mechanisms were used to construct an MPI implementation. In this section, we explain how MPICH was layered on top of Nexus; in the next section, we discuss I-WAY–specific issues.

4.1. MPI and MPICH

We first review important features of MPI and of the MPICH implementation on which this work is based.

The Message Passing Interface defines a standard set of functions for interprocess communication [15]. It defines functions for sending messages from one process to another (point-to-point communication), for communication operations that involve groups of processes (collective communication, such as reduction), and for obtaining information about the environment in which a program executes (enquiry functions). The communicator construct combines a group of processes and a unique tag space and can be used to ensure that communications associated with different parts of a program are not confused.

MPICH [14] is a portable, high-performance implementation of MPI. It is structured in terms of an

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

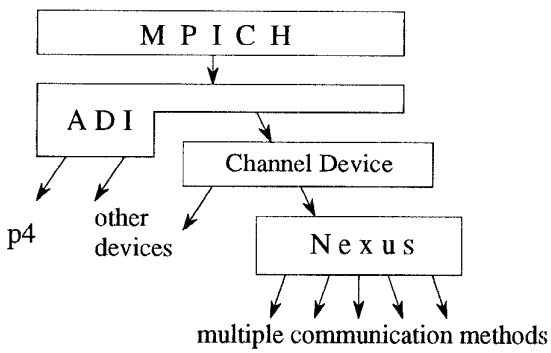

Figure 2. The Nexus implementation of MPI is constructed by defining a Nexus instantiation of the MPICH channel device, a specialization of the abstract device interface.

abstract device interface (ADI) that defines low-level communication-related functions that can be implemented in different ways on different machines [12, 13]. The Nexus implementation of MPI is constructed by providing a Nexus implementation of this device. The use of the ADI simplifies implementation but has some performance implications, which we discuss below.

4.2. The Abstract Device Interface

Figure 2 illustrates the structure of the MPICH implementation of MPI. Higher-level functions such as those relating to communicators and collective operations are implemented by a device-independent library, defined in terms of point-to-point communication functions provided by the ADI. To achieve high performance, the ADI provides a rich set of communication functions supporting different communication modes. A typical implementation of the ADI will map some functions directly to low-level mechanisms and implement others via library calls. The mapping of MPICH functions to ADI mechanisms is achieved via macros and preprocessors, not function calls. Hence, the overhead associated with this organization is often small or nonexistent [14].

The ADI provides a fairly high-level abstraction of a communication device: for example, it assumes that the device handles the buffering and queuing of messages. The lower-level channel interface defines simpler functions for moving data from one processor to another. For example, it defines `MPID_SendControl` and `MPID_SendChannel` functions that can be used to implement the MPI function `MPI_Send`. On the destination side, the test `MPID_ControlMsgAvail` and function `MPID_RecvAnyControl` are provided and can be used to implement `MPI_Recv`. Different protocols can be selected; the best in many circumstances sends both the message envelope (tag, communicator, etc.) and data in a single message, up to a certain data size, and then switches to a two-message protocol so as to avoid copying data.

The Nexus implementation of the channel device establishes a fully connected set of global pointers linking the processes involved in the MPI computation. Then, it implements channel device send functions as RSRs to "enqueue message" handlers; these handlers place data in appropriate queues or copy it directly to a receive buffer if a receive has already been posted. As this brief description shows, the mapping from ADI to Nexus is quite direct; the tricky issues relate mainly to avoiding extra copy operations. The principal overheads relative to MPICH comprise an additional 32 bytes of Nexus header information, which must be formatted and communicated; the decoding and dispatch of the Nexus handler on the receiving node; and a small number of additional function calls. We quantify these costs below. Most are artifacts of version 1 of the MPICH channel device; we are currently working with the MPICH developers to investigate a tighter integration of MPICH and Nexus, which we expect to eliminate most remaining overheads.

Finally, we observe that the Nexus implementation of MPI is structured so that Nexus thread management functions and MPI communication functions can both be used in the same program. This coexistence is simplified by the fact that the MPI specification is *thread safe*. That is, there is no implicit internal state that prevents the execution of MPI functions from being interleaved. The Nexus library addresses other thread safety issues, ensuring that only one thread at a time accesses nonthread-safe system components such as communication devices and I/O libraries on many systems.

4.3. Performance experiments

We have conducted a variety of performance experiments to evaluate the performance of both our multimethod communication mechanisms and the Nexus implementation of MPI. All experiments were conducted on the Argonne IBM SP2, which is configured with Power 1 rather than the more common Power 2 processors. These processors are connected via a high-speed multistage crossbar switch and are organized by software into disjoint partitions. Processors in the

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

same partition can communicate by using either TCP or IBM's proprietary Message Passing Library (MPL), while processors in different partitions can communicate via TCP only. Both MPL and TCP operate over the high-speed switch and can achieve maximum bandwidths of about 36 and 8 MB/sec, respectively. TCP communications incur the high latencies typically observed in other environments, and so multiple SP partitions can be used to provide a controlled testbed for experimentation with multimethod communication in networked systems.

Nexus performance experiments, reported elsewhere [10], reveal that on the Argonne SP2, a "ping-pong" benchmark that performs RSRs back and forth between two processors obtains a one-way cost of 82.8 μsec for a zero-length message; in contrast, the SP2's low-level MPL communication library takes 61.4 μsec. The principal sources of the 21.4 μsec difference between NexusLite and MPL are the setup and communication of the 32-byte header contained in a Nexus message (about 8 μsec) and the lookup and dispatch of the handler on the receive side (about 7 μsec) [10].

We evaluated the performance of the Nexus implementation of MPI by using the ping-pong benchmark provided by the MPI `mpptest` program [14]. We executed this program using both "native" MPICH and the Nexus implementation of MPI, in the later case comparing performance both with MPL support only and with MPL and TCP support. Figure 3 shows our results.

The graph on the left shows that MPICH takes 83.8 μsec for a zero-length message. This is comparable with the 82.8 μsec achieved by Nexus alone, suggesting that MPICH and Nexus are implemented at a similar level of optimization. The Nexus implementation of MPI incurs an overhead of around 60 μsec for a zero-length message; the graph on the right shows that for larger messages, the overhead becomes insignificant. We have outlined the sources of these overheads in Section 4.2; as we note there, we believe that most can be eliminated by improving the MPICH ADI. The jump in the MPICH numbers at 200 bytes is an artifact of the protocols used in the low-level MPL implementation. Notice the corresponding jump in the Nexus plots at around 170 bytes; the offset is due to the additional header information associated with a Nexus RSR.

The MPL+TCP results illustrate some performance issues that can arise when multiple communication methods must be supported. The Nexus implementation used in these experiments detects incoming communications by using a simple integrated polling scheme. This scheme invokes a method-specific poll operation for each communication method supported within a process. This approach can perform badly when the polling operation for one method is much slower than the others. For example, on many MPPs, the probe operation used to detect communication from another processor is cheap, while a TCP `select` is expensive. On the SP2, the `mpc_status` call used to detect an incoming MPL operation costs 15 microseconds, while a `select` costs around 100 microseconds. This sort of cost differential allows an infrequently used, expensive method to impose significant overhead on a frequently used, inexpensive method. These overheads can be reduced by using optimizations that, for example, perform TCP polls less frequently [7].

The results presented in this section are for a non-threaded implementation of Nexus. The results for the threaded version of Nexus are similar, except that we see an additional 29.6 μsec overhead on a zero-length message due to locking needed for thread safety and the use of a probe rather than a blocking receive to detect incoming messages.

4.4. Discussion

The Nexus implementation of MPI provides three benefits over and above those provided by MPICH: multimethod communication, interoperability with other Nexus applications, and multithreading.

The automatic selection of communication methods is supported directly in the Nexus implementation of MPI. An interesting question is how to support manual control of method selection in an MPI framework. We propose that this be achieved via MPI's caching mechanism, which allows the programmer to attach to communicators and subsequently modify and retrieve arbitrary key/value pairs called *attributes*. An MPI implementation can be extended to recognize certain attribute keys as denoting communication method choices and parameter values. For example, a key `TCP_BUFFER_SIZE` might be used to specify the buffer size to be used on a particular communicator.

A second benefit that accrues from the Nexus implementation of MPI is interoperability with other Nexus-based tools. For example, on the I-WAY, numerous applications used the CAVEcomm [5] client-server package to transfer data among one or more virtual reality systems and a scientific simulation running on a supercomputer. When the simulation itself was developed with MPI, the need arose to integrate the polling required to detect communication from either source. This integration is supported within Nexus.

The third benefit that accrues from the use of Nexus is access to multithreading. The concurrent execution

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

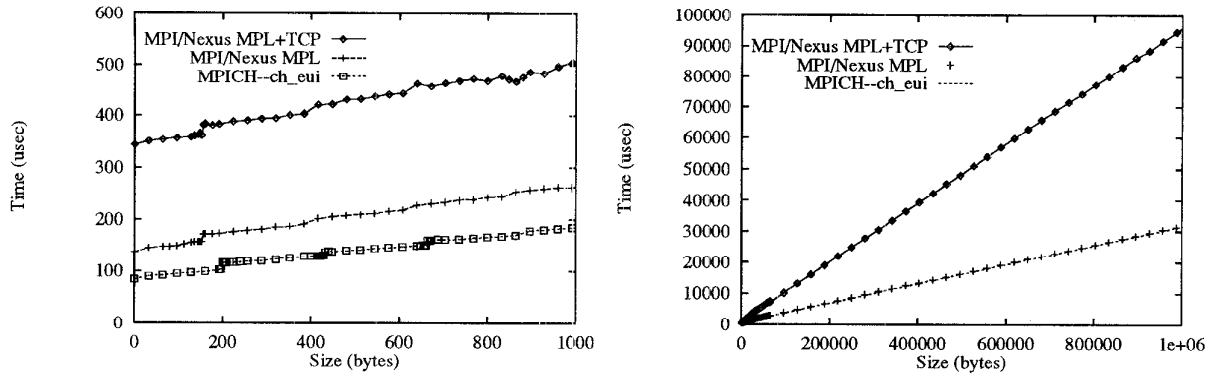

Figure 3. One-way message latency as a function of message size, for various implementations of MPI described in the text. The two graphs show results for small and large messages, respectively.

of multiple lightweight threads within a single process is a useful technique for masking variable latencies, exploiting multiprocessors, and providing concurrent access to shared resources. Various approaches to the integration of multithreading into a message-passing framework have been proposed (see [10] for a discussion). The Nexus implementation of MPI supports a particularly simple and elegant model that does not require that explicit thread identifiers be exported from MPI processes. Instead, threads are created and manipulated with Nexus functions, and interthread communication is performed by using standard MPI functions, with tags and/or communicators being used to distinguish messages intended for different threads. The MPI/Nexus combination can be used to implement a variety of interesting communication structures. For example, we can create two communicators and communicate independently on each from separate threads, using either point-to-point or collective operations. Or, several threads can receive on the *same* communicator and tag value. In a multiprocessor, the latter technique allows us to implement parallel servers that process requests from multiple clients concurrently.

The multithreaded MPI also has its limitations. In particular, it is not possible to define a collective operation that involves more than one thread per process. This functionality requires extensions to the MPI model [9, 16, 19].

Finally, we note that Nexus support for dynamic resource management and multithreading also provides a framework for implementing new features proposed for MPI-2, such as dynamic process management, single-sided communication, and multicast.

5. Nexus, MPI, and the I-WAY

The I-WAY implementation of MPI was constructed by extending the MPICH/Nexus system described in the preceding section to support startup mechanisms provided by the I-WAY software environment. The I-WAY scheduler was configured so that, when scheduling resources to users, it would also generate database entries describing the resources and the network configuration [8]. Nexus (and hence MPI) could then use this information when creating a user computation. This support made it possible for a user to allocate a heterogeneous collection of I-WAY resources and then start a program simply by typing "`impirun`."

The Nexus implementation of MPI was used extensively for I-WAY application development. Experiences emphasized the advantages of the Nexus automatic configuration mechanisms. In many cases, user were able to develop applications in MPI (or in other high-level Nexus-based tools such as CC++ or CAVEcomm) without any knowledge of low-level details relating to the compute and network resources included in a computation. These applications would then execute in heterogeneous environments. For example, in a virtual machine connecting IBM SP and SGI Challenge computers with both ATM and Internet networks, Nexus uses three different protocols (IBM

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

proprietary MPL on the SP, shared-memory on the Challenge, and TCP/IP or AAL5 between computers) and selects either ATM or Internet network interfaces, depending on network status. Other systems used Nexus mechanisms in the same manner, notably the parallel language CC++ [1] and the parallel scripting language nPerl [11], used to write the I-WAY scheduler.

A significant difficulty revealed by the I-WAY experiment related to the mechanisms used to generate and maintain the configuration information used by Nexus. While resource database entries were generated automatically by the scheduler, the information contained in these entries (such as network interfaces) had to be provided manually. The discovery, entry, and maintenance of this information proved to be time consuming, in particular because I-WAY network status proved to be highly changeable. Clearly, this information should be discovered automatically whenever possible. Automatic discovery would make it possible, for example, for a parallel tool to use dedicated ATM links if these were available, but to fall back automatically to shared Internet if the ATM link was discovered to be unavailable. The development of such automatic discovery techniques remains a challenging research problem.

The Nexus communication library provides mechanisms for querying the resource database, which users could have used to discover some properties of the machines and networks on which they were executing. In practice, few I-WAY applications were configured to use this information; however, we believe that this situation simply reflects the immature state of practice in this area and that users will soon learn to write programs that exploit properties of network topology, etc. Just what information users will find useful remains to be seen, but presumably enquiry functions that reveal the number of machines involved in a computation and the number of processors in each machine will be required. Our MPI implementation could use information about network topology to optimize collective operations, which are currently performed by using algorithms designed for multicomputer environments; presumably, communication costs can often be reduced by using communication structures that minimize intermachine communication.

6. Summary

We have describe an implementation of the Message Passing Interface designed to execute in wide area, heterogeneous environments. This implementation was used by numerous groups to develop applications for the I-WAY networking experiment. We developed this implementation by layering MPICH on the Nexus communication library and by integrating Nexus into the I-WAY software environment. This produced a system that can deal with heterogeneous authentication, process creation, and communication mechanisms. In particular, support for multimethod communication allowed an MPI application to use different communication mechanisms depending on where it was communicating. In future work, we expect to extend our MPI system so that programmers can use existing and future Nexus mechanisms to vary method selection according to what is being communicated or when communication is performed.

Microbenchmark studies provide insights into the costs associated with the Nexus implementation of MPI. The results presented here are promising in that they show that overheads associated with multimethod communication are small and manageable. However, we know that these overheads can be reduced further. The only unavoidable overheads associated with the Nexus implementation of MPI seem to be the few microseconds associated with handler lookup and the use of probe rather than blocking receive.

Acknowledgments

Our work on multimethod communication is a joint effort with Carl Kesselman and has also benefited greatly from discussions with Steve Schwab. Our MPI implementation was made possible by the outstanding MPICH implementation constructed by Bill Gropp, Ewing Lusk, Nathan Doss, and Tony Skjellum; we are grateful for their considerable help with this project. This work was supported by the National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[2] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press, 1993.

[3] C. Cruz-Neira, D. Sandin, T. DeFanti, R. Kenyon, and J. Hart. The CAVE: Audio visual experience automatic virtual environment. *CACM*, 35(6):65–72, 1992.

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE

[4] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *Intl J. Supercomputer Applications*, 1996. in press.

[5] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens. Sharing visualization experiences among remote virtual environments. In *International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237. Springer-Verlag, 1995.

[6] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 25(1), 1994.

[7] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Multimethod communication for high-performance networked computing systems. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[8] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high-performance distributed computing experiment. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*. IEEE, 1996.

[9] I. Foster, C. Kesselman, and M. Snir. Generalized communicators in the Message Passing Interface. In *Proc. MPI Developers Conf.* IEEE, 1996.

[10] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, 1996. To appear.

[11] I. Foster and R. Olson. A guide to parallel and distributed programming in nPerl. Technical report, Argonne National Laboratory, 1995. `http://www.mcs.anl.gov/nexus/nperl/`.

[12] W. Gropp and E. Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report MCS-P342-1193, Argonne National Laboratory, 1994.

[13] W. Gropp and E. Lusk. MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1996.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[16] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for collective operations among distributed threads. Technical Report 95-36, ICASE, 1995.

[17] C. Lee, C. Kesselman, and S. Schwab. Near-real-time satellite image processing: Metacomputing in CC++. *Computer Graphics and Applications*, 1996. to appear.

[18] M. Norman et al. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *Intl J. Supercomputer Applications*, 1996. in press.

[19] A. Skjellum, N. Doss, K. Viswanathan, A. Chowdappa, and P. Bangalore. Extending the message passing interface. In *Proc. 1994 Scalable Parallel Libraries Conf.* IEEE Computer Society Press, 1994.

Proceedings of the Second MPI Developers Conference (MPIDC '96)
0-8186-7533-0/96 $10.00 © 1996 IEEE