Wide-Area Implementation of the Message Passing Interface

Ian Foster, Jonathan Geisler, William Gropp,
Nicholas Karonis, Ewing Lusk, George Thiruvathukal, Steven Tuecke
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
{foster,geisler,gropp,karonis,lusk,thiruvat,tuecke}@mcs.anl.gov

Abstract

The Message Passing Interface (MPI) can be used as a portable, high-performance programming model for wide-area computing systems. The wide-area environment introduces challenging problems for the MPI implementor, due to the heterogeneity of both the underlying physical infrastructure and the software environment at different sites. In this article, we describe an MPI implementation that incorporates solutions to these problems. This implementation has been constructed by extending the Argonne MPICH implementation of MPI to use communication services provided by the Nexus communication library and authentication, resource allocation, process creation/management, and information services provided by the I-Soft system (initially) and the Globus metacomputing toolkit (work in progress). Nexus provides multimethod communication mechanisms that allow multiple communication methods to be used in a single computation with a uniform interface; I-Soft and Globus provided standard authentication, resource management, and process management mechanisms. We describe how these various mechanisms are supported in the Nexus implementation of MPI and present performance results for this implementation on multicomputers and networked systems. We also discuss how more advanced services provided by the Globus metacomputing toolkit are being used to construct a second-generation wide-area MPI.

1 Introduction

Wide area supercomputing or *metacomputing* environments couple geographically distributed resources to provide broader access to supercomputing capabilities or to enable qualitatively new classes of high-performance applications [2, 11, 12, 16]. These environments combine aspects of traditional distributed and parallel computing systems. Metacomputing systems, like distributed systems, must deal with heterogeneity and dynamic behaviors; as in parallel computing, performance requirements often demand careful structuring of computation and communication.

A variety of "nontraditional" programming models (from the parallel computing viewpoint) have been proposed and used for metacomputing applications, including parallel object-oriented programming [23, 28], CORBA, and specialized shared memory models [27]. However, while these models have many useful properties, it is clear that there is a continuing important role for message passing [15, 21]. Message passing provides a higher-level view of communication than the TCP/IP sockets often used in early metacomputing prototypes, while preserving for the programmer a high degree of control over how and when communication occurs.

The wide area environment introduces a number of new problems for the implementor of message-passing systems. Because applications must often execute on heterogeneous collections

of endsystem computers and associated networks, efficient message-passing implementations may need to utilize different communication methods for different communications. For ease of use, it is desirable that the message-passing system be able to negotiate with diverse resource management, process management, and security services in order to start programs on computers that span multiple administrative domains. Key to these *communication* and *startup* challenges is a need for accurate, up-to-date *information* about the structure and state of the various components of a metacomputing system. This information is also needed for efficient implementation of collective operations and specific application structures.

In this article, we report on our experiences implementing the Message Passing Interface (MPI) [26, 21] for metacomputing systems. The MPI standard allows programmers to write message-passing programs without concern for low-level details such as machine type, network structure, low-level protocols, etc. Our implementation addresses communication, startup, and information requirements by adapting the Argonne/Mississippi State MPICH library [20] to use specialized metacomputing services. MPICH provides a portable, high-performance implementation of MPI that incorporates some support for heterogeneous environments (e.g., the p4 device supports heterogeneous networks of workstations), but provides only limited support for wide-area metacomputing environments. Our wide area MPICH uses communication services provided by the Nexus communication library and startup and information services provided by the I-WAY metacomputing environment (in a first phase) and the Globus metacomputing toolkit (in a second phase). The result is a system that allows programmers to use simple, standard commands to run MPI programs in a variety of metacomputing environments (freely combining heterogeneous workstation and massively parallel resources), while making efficient use of underlying networks.

The rest of this article is organized as follows. Sections 2 and 3 introduce Nexus and MPICH, and explain how Nexus services are used to implement MPICH communication structures. Section 4 explains how startup and information issues have been addressed in the I-WAY and Globus contexts. In Section 5 we discuss outstanding issues and ideas for further development. Finally, we review related work in Section 6 and summarize in Section 7.

2 Nexus

We provide a brief introduction to the Nexus communication library that we use to implement 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 languages and communication libraries.

2.1 Nexus Overview

Nexus is structured in terms of five basic abstractions: nodes, contexts, threads, communication links, 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 *communication link* provides a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. Communication flows from a communication *startpoint* to a communication *endpoint*. A startpoint is bound to an endpoint to form a *communication link*. Many startpoints can

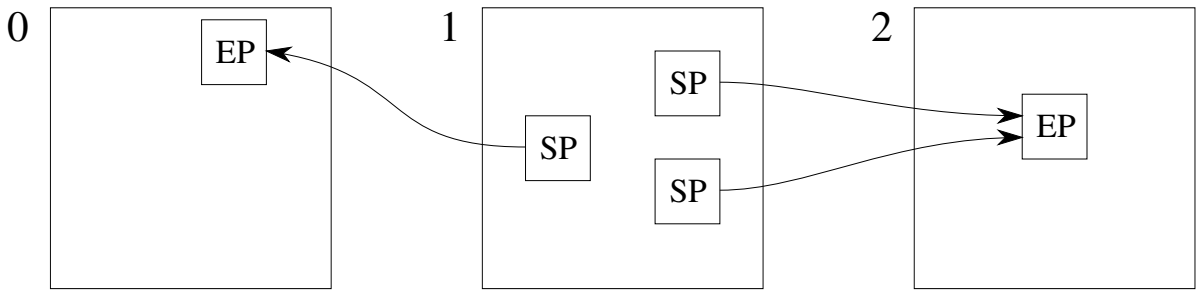

Figure 1: The communication link and its role in communication. The figure shows three address spaces; three startpoints in address space 1 reference endpoint in address spaces 0 and 2.

be bound to a single endpoint, in which case incoming communication is merged as in typical point-to-point message passing systems. Similarly, many endpoints can be bound to a single startpoint, resulting in a multicast communication pattern. Both startpoints and endpoints can be created dynamically; the startpoint has the additional property that it can be moved between processors using the communication operations we now describe.

Communication links are used in conjunction with asynchronous *remote service requests* (RSRs) which invoke actions on remote objects. An RSR is specified by providing a startpoint, an RSR handler identifier and a data buffer, which is constructed using PVM [15] style `put` routines. Issuing an RSR causes the data buffer to be transferred from the startpoint to the bound endpoint, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and endpoint-specific data are available to the RSR handler.

Key to the communication link's utility is the mobility of the startpoint. A process can bind a startpoint to a local endpoint and then communicate that startpoint to other processes, providing the other processes with a handle that they can use to perform RSRs back to the local endpoint. A process can create multiple handles, referring to different endpoints, hence allowing communications intended for different purposes to be distinguished.

The Active Messages (AM) [25] and Fast Messages (FM) [29] communication systems are based on asynchronous handler invocation mechanisms similar to those used in Nexus. The latest AM specification introduces an endpoint construct with some similarities to the Nexus endpoint. However, the AM endpoint is a more heavyweight structure, incorporating both startpoint and endpoint functionality. Also, AM handlers are used in request/reply pairs, rather than in a one-sided fashion as in Nexus.

2.2 Multimethod Communication

The Nexus features that are most important in a metacomputing environment are those that support multimethod communication [8]. These mechanisms are based around the startpoint 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 obtained from an information service 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 within Nexus RSRs results in communications 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 or unreliable transport protocols when appropriate.

Automatic configuration makes sense only if we have access to up-to-date information. We discuss below the techniques used to create and maintain this information.

3 Implementing MPI Communication

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 [26]. 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 [20] is a portable, high-performance implementation of MPI. It is structured in terms of an abstract device interface (ADI). The ADI defines low-level communication-related functions that can be implemented in different ways on different machines [17, 18, 19]. The Nexus implementation of MPI is constructed by providing a Nexus implementation of this device. The following discussion reflects the second-generation ADI, ADI-2, used in the latest Nexus-based implementation of MPICH, rather than the first-generation ADI (ADI-1) used in early implementations [10].

3.1 The MPICH 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 by calling services provided by the common reference ADI implementation. The mapping of MPICH functions to ADI mechanisms is achieved in part via macros and preprocessors, not function calls. Hence, the overhead associated with this organization is often small or nonexistent [20].

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. At the same time, it permits a high degree of control over how messages are constructed, avoiding unnecessary copying that resulted in ADI-1 when assembling messages from several components, translating between different data representations, or implementing "posted" receives. The key to this flexibility and efficiency is that the device is made responsible for datatype management. For example, the device is required to implement (either by using the reference implementation or through custom routines) `MPID_SendDatatype` and `MPID_IsendDataType` functions that implement blocking and non-blocking send forms, respectively. These routines are responsible for handling MPI datatypes, which are

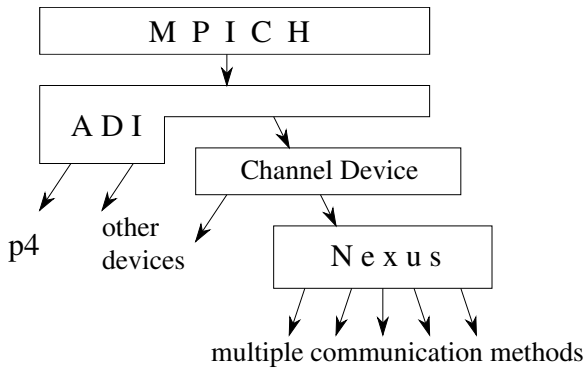

Figure 2: The Nexus implementation of MPI is constructed by defining a Nexus instantiation of the MPICH abstract device interface. In the first version of MPICH/Nexus, this was achieved via an intermediate channel device, as shown here; more recently, a tighter integration of MPICH and Nexus has been achieved by providing a full Nexus implementation of ADI-2.

represented by tree-structured descriptors that the device traverses to perform buffer translation when sending and receiving messages. They can represent noncontiguous data structures. ADI-2 also extends ADI-1 with support for MPI's user-packed buffers.

The Nexus implementation of ADI establishes a fully connected set of communication links connecting the processes involved in the MPI computation. Then, it implements ADI 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. Implementations of the device on different computers may use different protocols to perform the data transfer. The best strategy in many circumstances is to send both the message envelope (tag, communicator, etc.) and data in a single message, up to a certain data size, and then switch to a two-message protocol so as to avoid copying data.

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 for the MPICH/Nexus initial implementation, which was based on ADI-1 and the MPICH channel device, a simple interface designed for quick ports. We expect the latest version of MPICH/Nexus to achieve better performance due to its tighter integration of the two systems, however we were not able to obtain performance results from that system in time for this paper.

3.2 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. As noted above, they reflect the earlier ADI-1 implementation of MPICH/Nexus. These processors are connected via a high-speed multistage crossbar switch and are organized by software into disjoint partitions. Processors in the same partition can communicate

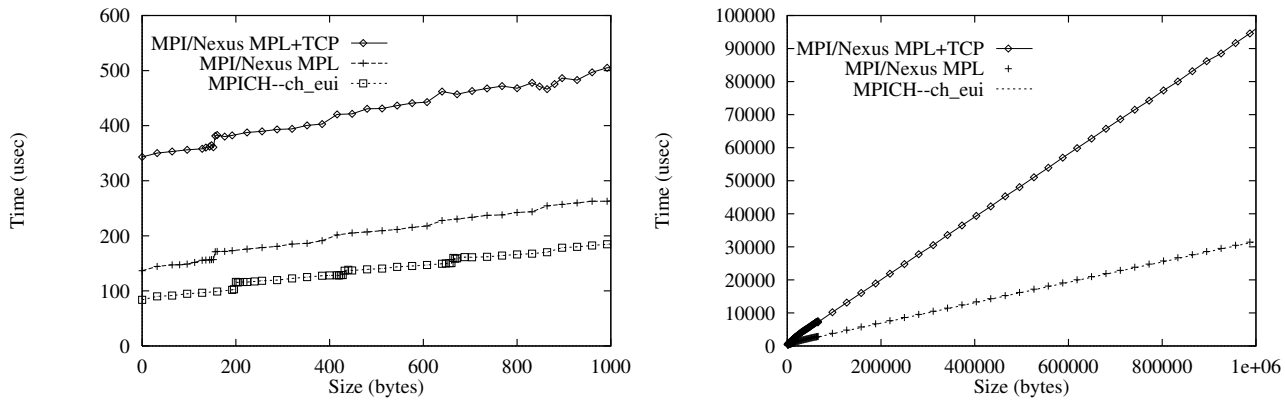

Figure 3: One-way message latency as a function of message size, for various implementations of MPI described in the text. The two graphs show results for small and large messages, respectively.

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 [14], 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) [14].

We evaluated the performance of the Nexus implementation of MPI by using the ping-pong benchmark provided by the MPI `mpptest` program [20]. 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 3.1; as we note there, we believe that many are eliminated by the use of ADI-2. 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 in-

vokes 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 [8].

The results presented in this section are for a nonthreaded 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 Startup and Information Services

We now consider the related problems of initiating MPI computations ("startup") and providing the information required to select communication and startup mechanisms. We first describe the techniques adopted in the I-WAY software environment and then discuss how difficulties encountered in that environment can be overcome by using the more sophisticated services provided by the Globus metacomputing infrastructure toolkit.

4.1 The I-WAY Software Environment

The I-WAY [5] 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™ and ImmersaDesk™ 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., [28]), collaborative design, and the coupling of remote supercomputers and databases into local environments (e.g., [23]). 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. The need to configure both IP routing tables and ATM virtual circuits in this highly heterogeneous environment was a significant source of implementation complexity.

As part of the I-WAY project, we and others developed a management and application programming environment called I-Soft that provided uniform authentication, resource reservation, process creation, and communication functions across I-WAY resources [9]. These services took advantage of dedicated I-WAY Point of Presence (I-POP) machines deployed 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.

The I-WAY implementation of MPI was constructed by extending the MPICH/Nexus system described in the preceding section to use I-Soft services. The I-WAY scheduler was configured so that, when scheduling resources to users, it would also generate a text file describing the resources and the network configuration [9]. 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 MPI implementation described here was used extensively for I-WAY application development. Experiences emphasized the advantages of the Nexus automatic configuration mechanisms. Users could develop MPI applications 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 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.

4.2 The Globus Metacomputing Toolkit

While successful in the sense that real applications were able to operate in a large-scale wide area environment, the I-WAY software environment and the I-WAY implementation of MPI also had significant weaknesses. Many of these issues are being addressed in the Globus project [11, 13], a multi-institutional effort that is developing a set of core services (the *Globus toolkit*, for which Nexus is the communication service) for metacomputing applications. In this section, we outline briefly how Globus mechanisms can be used to address a number of deficiencies noted in our I-WAY implementation of MPI. We consider in turn information services, security, resource management, and process management.

Information Services. 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 program 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 Globus Metacomputing Directory Service (MDS) [7] is designed to address these issues. MDS provides a uniform interface to a wide variety of information about the structure and state of a metacomputing system. MDS uses the API and data representation defined by the Lightweight Directory Access Protocol (LDAP) [22] to construct a framework within which can be represented static and dynamic information about computers, networks, etc. In contrast to the text file used to communicate information within the I-WAY experiment, MDS provides for a richer set of information, dynamic update of data, and distributed maintenance of information. Information about resources at a particular site can be discovered automatically and/or specified by site administrators, and maintained locally at that site. Services such as the Network Weather Service [30] can be used to determine the instantaneous status of network links.

Authentication and Resource Management. I-Soft provided single sign-on authentication and a centralized global scheduler for I-WAY resources. However, while effective, the implementations of these services had significant deficiencies. In particular, they enforced the use of particular policies at individual sites, such as the use of an I-POP machine and dedicated allocation of resources to the I-Soft scheduler [9].

The Globus toolkit seeks to overcome these deficiencies by defining more flexible interfaces that can then be implemented in terms of diverse local policies [13]. The Globus Security Services use public-key mechanisms to map a "globus id" to local ids representing users at different sites. The Globus Resource Access Manager provides a uniform interface to diverse low-level schedulers and process management mechanisms [4]. Our MPI implementation has been modified to use these services to support the creation, monitoring, and management of computations that span multiple sites.

5 Future Issues

The Nexus/Globus implementation of MPI makes possible a number of extensions to the conventional MPI interface and programming model that may be useful in metacomputing systems. We discuss these here.

Further Optimizations. The Globus implementation of MPI uses system information to select from among alternative communication mechanisms but does not otherwise seek to optimize program execution for a wide area environment. A variety of other optimizations are possible. For example, MPI topologies can be used to guide the allocation of ranks to processes, network structure information can be used to optimize the implementation of collective operations [24], and multicast mechanisms incorporated in Nexus can be used to optimize broadcast operations.

Multimethod Communication. Support for multimethod communication can be extended to support manual control of method selection in an MPI framework, for example to allow for programmer selection of specialized methods that use unreliable transport or that compress data. One promising approach to the specification of this selection is to use MPI's attribute-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, while a key `UNRELIABLE` could be used to indicate that communication over a certain communicator can be performed with an unreliable protocol, if this is more efficient.

User Access to Structure Information. The Globus MDS can be used to provide to the programmer a wide range of information about the structure and state of the machines and networks on which they are executing. In the I-WAY experiment, few 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.

MPI-2 Dynamic Support. The MPI-2 standard defines a number of functions (e.g., `MPI_SPAWN`, `MPI_CONNECT`) for creating processes dynamically and connecting previously independent computations. Globus mechanisms can be used to provide portable, secure implementations of these functions. MPI-2 also defines an interface to a lookup service; Globus mechanisms can be used to implement `MPI_PUBLISH_NAME` and `MPI_LOOKUP_NAME`.

6 Related Work

Some message-passing libraries permit different communication methods to coexist. For example, the Intel Paragon implementations of p4 and PVM support heterogeneous computing by using the NX communication library for internal communication and TCP for external communication [1, 15]; p4 supports NX and TCP within a single process, while PVM uses a proxy process for TCP. In both systems, the choice of method is hard coded and requires modification to the implementations to add new communication methods (both p4 and PVM support a variety of methods, including IBM's MPL and shared memory, as well as TCP).

The deployment of optimized vendor implementations of MPI on parallel computers such as the IBM SP2 introduces new challenges for wide area implementations. The Nexus implementation of MPICH can use vendor-supplied MPIs as a low-level transport, but programs that use this library are then somewhat less efficient (especially for small messages) than programs that use vendor-specific MPIs within parallel computers. However, the latter approach requires that different vendor MPIs interoperate; this is not straightforward as different MPIs use different message formats. Two different approaches have been proposed to this problem. The PVMPI system [6] exploits MPI's profiling interface to support interoperability. This interface allows calls to MPI routines to be trapped and handled by user-supplied functions. In PVMPI, the "user-supplied" function determines whether to call regular MPI functions (within a computer) or PVMPI functions (between computers). PVMPI, however, does not address some of the more subtle requirements of an MPI implementation with respect to MPI's various send modes and collective communication. An effort called IMPI (for interoperable MPI), hosted by the National Institute of Standards and Technology and involving only the commercial MPI vendors (hence only homogeneous implementations), seeks to standardize an IP level interface to MPI, allowing different vendor implementations to connect to each other.

7 Summary

We have described an implementation of the Message Passing Interface designed to execute in wide area, heterogeneous environments. We developed this implementation by layering MPICH on the Nexus communication library and by using startup and information mechanisms provided by the I-WAY software environment (initially) and the Globus project (work in progress). This integration produces a system that can deal with heterogeneous communication mechanisms, authentication, resource management, and process management mechanisms. In particular, support for multimethod communication allows an MPI application to use different communication mechanisms depending on where it was communicating. This implementation has been used by numerous groups to develop wide area applications for wide area computing systems, initially as part of the I-WAY project and subsequently elsewhere.

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 over-

heads 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.

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. We also expect to develop support for MPI-2 functionality and to investigate ways in which information provided by the Globus information service can be used to optimize MPI collective operations.

Acknowledgments

Our work on Nexus and Globus is a joint effort with Carl Kesselman and his colleagues at the USC Information Sciences Institute. 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] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[2] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

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

[4] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997. Submitted.

[5] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2):123–130, 1996.

[6] G. Fagg, J. Dongarra, and A. Geist. PVMPI provides interoperability between MPI implementations. In *Proc. 8th SIAM Conf. on Parallel Processing*. SIAM, 1997.

[7] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[8] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[9] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY metacomputing experiment. *Concurrency: Practice & Experience*, 1998. to appear.

[10] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 10–17. IEEE Computer Society Press, 1996.

[11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[12] I. Foster and C. Kesselman, editors. *Computational Grids: The Future of High-Performance Distributed Computing*. Morgan Kaufmann Publishers, 1998.

[13] I. Foster and C. Kesselman. The Globus project: A progress report. In *Proceedings of the Heterogeneous Computing Workshop*, 1998. to appear.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[16] A. Grimshaw, J. Weissman, E. West, and E. Lyot, Jr. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.

[17] W. Gropp and E. Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

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

[19] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[22] T. Howes and M. Smith. The LDAP application program interface. RFC 1823, 08/09 1995.

[23] C. Lee, C. Kesselman, and S. Schwab. Near-realtime satellite image processing: Metacomputing in CC++. *Computer Graphics and Applications*, 16(4):79–84, 1996.

[24] B. Lowekamp and A. Beguelin. ECO: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society Press, 1997.

[25] A. Mainwaring. Active Message applications programming interface and communication subsystem organization. Technical report, Dept. of Computer Science, UC Berkeley, Berkeley, CA, 1996.

[26] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[27] J. Nieplocha and R. Harrison. Shared memory NUMA programming on the I-WAY. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 432–441. IEEE Computer Society Press, 1996.

[28] M. Norman, P. Beckman, G. Bryan, J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. Ostriker, J. Shalf, J. Welling, and S. Yang. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *International Journal of Supercomputer Applications*, 10(2):131–140, 1996.

[29] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*. IEEE Computer Society Press, 1996.

[30] Richard Wolski. Dynamically forecasting network performance using the network weather service. Technical Report TR-CS96-494, U.C. San Diego, October 1996.