A Grid-Enabled MPI:
Message Passing in Heterogeneous Distributed Computing Systems

Ian Foster

Mathematics and Computer Science
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Nicholas T. Karonis

High–Performance Computing Lab
Department of Computer Science
Northern Illinois University
DeKalb, IL 60115

Abstract

Application development for high-performance distributed computing systems, or computational grids as they are sometimes called, requires "grid-enabled" tools that hide mundane aspects of the heterogeneous grid environment without compromising performance. As part of an investigation of these issues, we have developed MPICH-G, a grid-enabled implementation of the Message Passing Interface (MPI) that allows a user to run MPI programs across multiple computers at different sites using the same commands that would be used on a parallel computer. This library extends the Argonne MPICH implementation of MPI to use services provided by the Globus grid toolkit. In this paper, we describe the MPICH-G implementation and present preliminary performance results.

1 Introduction

High-performance "computational grids" [11] involve heterogeneous collections of computers that may reside in different administrative domains, run different software, be subject to different access control policies, and be connected by networks with widely varying performance characteristics. We believe that application development in these environments requires specialized "grid-enabled" tools that hide mundane aspects of the heterogeneous grid environment without compromising performance. These tools may implement familiar programming models, such as message passing, data parallelism, or object parallelism (perhaps with extensions), or may implement completely new programming models. In either case, research is required to understand the utility of different approaches and the techniques that may be used to implement these approaches in different environments.

As part of an investigation of these issues, we have developed MPICH-G, a grid-enabled implementation of the Message Passing Interface (MPI) that allows the user to run MPI programs across multiple computers at different sites using the same commands that would be used on a parallel computer. This library extends the Argonne MPICH implementation of MPI [15] to use services provided by the Globus grid toolkit [10], as follows:

1. The Globus information service is used to determine how to obtain access to the computers in question.
2. The Globus security service is used to handle authentication and authorization at each site.
3. The Globus executable management service is used to stage executables.
4. The Globus resource management service is used to start processes on each computer, interfacing with local schedulers where necessary.
5. The Globus communication service is used to manage the different communication methods that may apply in a heterogeneous environment, such as vendor-supplied protocols or TCP/IP.
6. The Globus file access service is used to direct standard output and error (stdout and stderr) streams to the user's terminal and to provide access to files regardless of location.
7. Globus process management facilities allow the programmer to monitor the progress of an application and terminate it if desired.

MPICH-G is a complete implementation of the MPI-1 standard and passes the MPICH test suite. Early experiences suggest that it achieves our goal of reducing barriers to the use of distributed computing by allowing the use of MPI as a portable, high-performance programming model for heterogeneous clusters and for wide-area computing systems. Several groups (e.g., at Lawrence Livermore National Laboratory (LLNL) and NASA Ames Research Center) are using it to run conventional MPI programs across multiple massively parallel processors (MPPs) within the same machine room. In this case, MPICH-G is used primarily to manage startup and to achieve efficient communication via use of different low-level communication methods. Other groups are using MPICH-G for metacomputing experiments, in which applications are distributed across MPPs located at different sites: Larsson for studies of distributed execution of a large computational electromagnetics code [17], and Chen and Taylor in studies of automatic partitioning techniques as applied to finite element codes [4]. MPICH-G can also be used to implement distributed visualization pipelines and similar applications in which components are located at different sites. In these latter examples, MPICH-G is used to manage heterogeneous authentication and startup mechanisms.

In the rest of this article, we describe the problems that we faced in developing MPICH-G, the techniques used to overcome these problems, and preliminary experimental results that indicate the costs associated with the MPICH-G implementation.

2 The Need for Grid-Enabled Tools

An extensive body of experience shows that the coupling of geographically distributed computers, databases, scientific instruments, and people can enable interesting new applications. Distributed supercomputing [19], knowledge synthesis [20], online instrument control [16], and teleimmersion [6] are just four examples. However, experience also shows that the barriers to the construction of such applications are considerable. Few programmers take the time to master the intricacies of such grid environments, and even then often produce applications that are fragile, nonportable, and perform poorly.

The specific problems encountered by the developers of such grid applications vary widely according to the grid environment and application type in question. We use Figure 1 to illustrate some of the problems that we have been concerned with in the development of MPICH-G. This figure shows three massively parallel processing (MPP) systems, each constructed from symmetric multiprocessor (SMP) nodes. Two of the MPPs are located within the same institution and hence are connected by some form of (hopefully high-speed) local area network (LAN), while the third is located at a remote site and hence is reached by a wide area network (WAN). The following is a partial list of the problems that we may encounter in such an environment.

1. The two sites will likely operate different authentication and authorization mechanisms and impose different access control policies. A user is unlikely to have the same user id at the two sites.

2. The two sites are unlikely to share a file system. Hence, specialized techniques are required to transfer executables and program files between sites.

3. The different MPPs may be controlled by different schedulers with different scheduling policies.

4. We need to allocate resources concurrently at multiple sites and establish a single computational environment (in MPI terms, a single `MPI_COMM_WORLD`) that spans those resources. (We refer to this as the "co-allocation" problem.)

5. Efficient communication requires that different communication methods be used in different situations. Within an SMP, shared-memory communication should be used, whether by using explicit shared-memory operations or by using shared memory operations to provide fast implementations of other abstractions such as message passing. Between SMPs within the same MPP, a vendor-supplied message-passing library should be used. Only between MPPs should the universally available but slow TCP/IP be used. (An exception to this rule is shown in the upper MPP in Figure 1. In some cases, a limitation on the number of nodes that can communicate using the vendor-supplied library may require the use of TCP/IP even within an MPP.)

6. The topology of the overall computational system needs to be taken into account when implementing communication algorithms. Taking into account the different TCP/IP performance (in terms of both absolute speeds and bisection bandwidths) within an MPP, over a LAN, and over a WAN, the example system features five different communication speed regimes.

We believe that the solution to these types of problem is to develop grid-enabled tools that provide efficient implementations of familiar (or unfamiliar) programming models for use by application developers. In

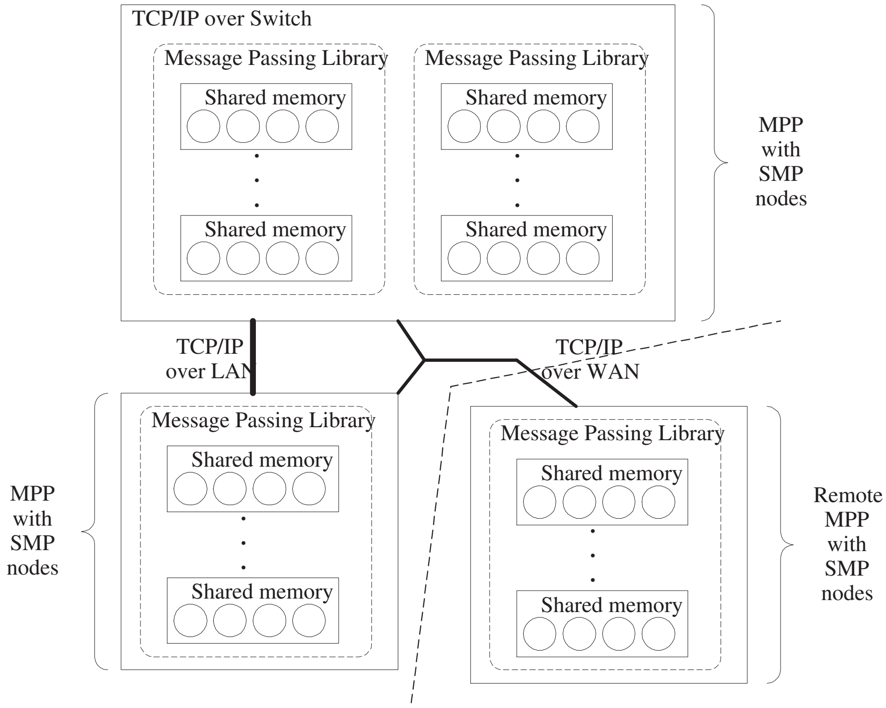

Figure 1. The structure of a prototypical "computational grid" computing environment, of the type supported by MPICH-G. See text for details.

developing these implementations, the tool developer must be concerned not only with translating the programming model to the grid environment, but also with revealing to the programmer those aspects of the grid environment that impact performance. For example, a grid-enabled MPI might handle automatically issues of authorization, startup, and process management, hence addressing the first four points listed above. It might also incorporate specialized techniques for point-to-point and collective communication in highly heterogeneous environments, hence addressing points 5 and 6. Finally, it might also extend the MPI model to provide programmers with access to resource location services, information about grid topology, group communication protocols, and quality-of-service management services, so as to enable new programming techniques appropriate for grid environments.

In principle, such grid-enabled tools could be constructed from scratch. However, the task is greatly simplified if the programmer has access to appropriate low-level services. As we explain below, we use the Globus toolkit as a source of such services in our work.

The state of the art with respect to such tools is not very advanced. Systems such as Condor [18], NEOS [5], and NetSolve [3] all implement grid-based programming models of various sorts. Various implementations of message passing libraries provide some support for heterogeneous execution (e.g., p4 [2] and PVM [14]), but these systems do not support the flexible use of alternative low-level communication protocols, interfaces to different MPP schedulers, or the MPI standard. PVMPI [7] exploits a renaming capability provided by MPI's profiling interface to use PVM mechanisms to couple vendor-supplied MPIs on different MPPs. The resulting system supports heterogeneous execution of MPI programs but cannot deal with heterogeneous startup mechanisms or dynamic selection of communication methods.

3 Building Blocks

Our grid-enabled MPI implementation is constructed from two existing software systems: MPICH and Globus. We describe these briefly here.

3.1 MPICH

MPICH [15] is the most widely used implementation of the MPI standard. Its architecture features a layered design, in which higher-level MPI communication constructs such as collective operations, communicators, and topologies are implemented in terms of basic communication operations provided by an "abstract device." Various such devices have been designed, enabling high-performance implementations of MPICH on a variety of platforms. We exploit this device architecture in our work, defining a "Globus communication device" that supports the use of multiple low-level communication methods in heterogeneous wide area environments.

MPICH also defines a uniform startup mechanism for MPI programs. For example, the command

```
mpirun -np 64 myprog
```

starts the MPI program `myprog` as 64 processes, whether on a shared-memory multiprocessor (via `fork`), a set of workstations on a local area network (e.g., via `rsh`), or on an MPP (e.g., via `POE` commands on an IBM SP). Our MPICH-G implementation allows the same command syntax to be used even when starting programs across multiple MPPs of different architectures. We believe that it is a significant achievement that we can provide a similarly simple and uniform interface in much more complex grid environments.

3.2 Globus

Globus is a widely used toolkit for building wide area applications. The toolkit comprises a set of interrelated components, each providing services and associated APIs that address a distinct aspect of wide area computing [10]. Components developed to date are

1. the Nexus communication library, providing support for multimethod communication;
2. resource management services, providing uniform interfaces to local schedulers and support for brokering and co-allocation (see below);
3. security services, providing support for single sign-on, multiway security contexts, and interfaces to local security services;
4. file access services, providing staging services and uniform interfaces to files, regardless of location;
5. an Lightweight Directory Access Protocol (LDAP)-based information service, the Metacomputing Directory Service (MDS), providing uniform access to up-to-date information about Globus resource structure and state;
6. a fault detection service, providing a notification service for faulty processes; and
7. executable management services that support staging of executables to remote computers.

Globus has distinct local service, global service, and client components. At Globus sites, a small set of servers provide (deliberately simple) *local services* such as authentication, resource allocation, and status monitoring. In particular, a Globus Resource Allocation Manager (GRAM) implements a uniform interface to local resources (computers, networks, etc.) for authentication and allocation. Additional *global services*, defined in terms of these local services, provide more sophisticated functionality, such as resource brokering, co-allocation of resources, and fault detection. Finally, *client libraries* allow application programs and tools to invoke local and global services.

Globus toolkit components are designed to support the incremental development of grid-enabled tools and applications. In principle, the user should be able to take either an existing or new program and gradually make it more "grid-aware" by introducing additional services. Preliminary application experiences suggest that this incremental development methodology works well [10]. Various groups are using a similar methodology to apply Globus components in other tool projects (e.g., [1, 13]); however, MPICH-G is the most sophisticated such system constructed to date.

4 The MPICH-G Library

We briefly describe the techniques used to implement some of the MPICH-G capabilities listed in the introduction.

4.1 Startup: `mpirun` and the `machines` File

MPICH provides a standard command for starting MPI programs, namely, `mpirun`. This command specifies the number of processes that are to be created and can also provide flags relating to debugging and so forth.

On a parallel computer such as the IBM SP, the MPICH implementation of `mpirun` simply generates an appropriate job submission command to whatever scheduler is used to obtain access to the MPP. On the other hand, in a network of workstations environment, a `machines` file is accessed to determine which machines the MPI program should be started on. For example, the following file indicates that one process should be started on each of `donner` and `dalek`, and two processes on `pitcairn`.

```
donner
dalek
pitcairn 2
```

Our only change to the MPICH startup model is that we generalize the contents of the `machines` file to include resource manager (GRAM) names. For example, the following file names three such resource managers, at three different sites:

```
donner.mcs.anl.gov-fork 8
bonny.isi.edu-fork 8
moti4.ncsa.uiuc.edu-lsf 64
```

The MPICH-G implementation then uses the Globus information service, MDS, to perform a simple form of resource location, accessing MDS to determine detailed contact information (e.g., port numbers) for the specified resource managers. Hence, the user need not be concerned with low-level details regarding the physical location and interfaces of resources.

The user can build on this simple capability to implement more sophisticated resource location schemes. For example, rather than specifying node counts in the `machines` file, the user can perform an MDS search to determine how many nodes are available on each machine, and can rewrite the `machines` file appropriately. Or, the user can perform an MDS search to locate resource managers with particular properties (e.g., idle nodes and specified network bandwidth) and then place the names of those systems in the `machines` file.

4.2 Job Submission and Execution

Once the `machines` file has been read and resource manager contacts determined, the MPICH-G `mpirun` implementation calls a Globus-provided function called `globusrun` to manage the task of job submission and execution. This function uses a variety of Globus services and libraries, as follows:

Co-allocation. As noted above, the creation of a computation that spans multiple MPPs is a difficult problem. We must allocate resources on the selected computers, start processes, and link these processes into a computation. Different computers differ widely in the mechanisms used for resource allocation and process creation, so a first requirement is to negotiate the appropriate mechanisms at each site. A second concern is that startup can be a timeconsuming and error-prone activity; hence, we require techniques for detecting failure (e.g., via timeout) and synchronizing once startup completes. These two concerns are addressed via the use of the GRAM interface (discussed above) and an appropriate *co-allocator* library, respectively. MPICH-G uses the Dynamically-Updated Request Online Co-allocator (DUROC). DUROC submits requests, verifies correct startup, and provides functions that can then be used to coordinate the various subjobs so as to create (in our current case) a single `MPI_COMM_WORLD` spanning all processes. The need to reserve resources at multiple sites simultaneously remains as a problem, which we are investigating in current work.

Authentication and authorization. A significant obstacle to the use of multiple distributed resources is that the user will typically have a distinct "trust relationship" (e.g., account), or even no prior trust relationship at all, at different sites. Hence, starting a program can be a frustrating process involving multiple logins. MPICH-G avoids this because the Globus Security Infrastructure supports single sign-on and automatic mapping (under site control) to appropriate local accounts. Public key technology is used to avoid the transfer of plaintext passwords.

Executable staging. Manual staging of executables is another painful activity. MPICH-G overcomes this obstacle by using the Globus "Global Access Secondary Storage" (GASS) service to stage executables to remote machines. Currently, this technique works only if the programmer has supplied an appropriate executable for each remote computer. In future work, the Globus group plans to investigate automated techniques for identifying and generating appropriate executables, for example by using compile servers.

Communication. As described in an earlier paper [8] which focused specifically on multimethod communication in MPICH-G, the Nexus communication library is used to provide access to multiple communication methods [9]: e.g., TCP/IP in the wide area, vendor-specific protocols within a computer, and shared memory within a cluster.

Monitoring, control, stdout. The `globusrun` utility used by `mpirun` also provides a number of other useful capabilities. Callbacks provided by GRAMs allow it to detect and report termination. Control functions provided by the GRAM API allow it to terminate a computation in the event of a user signal (control-C) or if a component fails. Finally, GASS mechanisms are used to collect standard output and error streams and route these back to the originating terminal.

5 Performance Studies

An empirical evaluation of a library such as MPICH-G should, ideally, address at least the following issues:

1. Startup costs: What is the cost of the authentication, authorization, resource location/allocation, and other management mechanisms? Are these mechanisms scalable?

2. Communication costs: What is the impact of the multimethod communication support on point-to-point and collective communication performance, for both simple benchmark programs and real applications, and in both homogeneous and heterogeneous environments?

3. Reliability: Are the management and communication mechanisms provided able to operate reliably in wide area environments?

We present here preliminary results for point-to-point communication performance in homogeneous systems; optimization in this configuration, and other measurements, are ongoing. We use the "ping-pong" benchmark programs provided with MPICH [15] to evaluate the performance of MPICH-G. We study performance on an IBM SP2 system at Lawrence Livermore National Laboratory (LLNL). This system runs AIX 4.3.1 and is configured with four-way SMP nodes with 332 MHz PowerPC 604e processors. This configuration provides 1.2 GB/s bandwidth to memory and 150 MB/s switch bandwidth. All communication measurements are between processors on different nodes.

We measured performance for five different communication libraries:

1. **IBM-MPI**, the nonthreaded IBM implementation of MPI.

2. **IBM-MPL**, the IBM implementation of MPL, the original communication library provided on the IBM SP.

3. **MPICH-mpl**, MPICH operating over the IBM MPL library.

4. **Nexus**, the Globus communication library (also operating over the IBM MPL library in this situation).

5. **MPICH-G**, MPICH-G operating over the Globus communication library (which in turn uses the IBM MPL library).

In addition, for each of these libraries we measured performance when operating over two different bindings for the IBM and IBM MPL library: one that uses the more efficient user space communication and one based on TCP/IP. Also, for Nexus and MPICH-G we evaluated the impact of two different values for the "skip-poll" parameter, as discussed below. The results are presented in Tables 1 and 2.

In brief, we find that when using user space communication, MPICH-G incurs an overhead of 48 μsecfor a zero-length message (when skip poll=10K) and achieves 35 percent of the peak bandwidth achieved by IBM's MPI. These are certainly not good results, but nor are they dreadful, and on the basis of previous studies [12, 8], we believe that we understand the source of these overheads and know how to eliminate a significant part of them, by eliminating extra copies, improving memory management, and streamlining certain interfaces. Overall, we believe that we can achieve performance close to that of MPICH-mpl in most situations.

The user space results for Nexus and MPICH-mpl provide some insights into the nature of the overheads. The zero-byte latency for Nexus is 42 μsec, while that for MPICH-mpl is only 32 μsec; this difference reflects certain known overheads associated with the Nexus communication model and implementation [12]. But the bulk of the overhead (31 μsec) is clearly associated with the layering of MPICH-G on Nexus, something that we have not optimized carefully. The bandwidth numbers for Nexus and MPICH-G are identical, indicating that the overheads here lie in Nexus. The source of this overhead is additional copies performed in the Nexus system on send and receive. These can be corrected, but the necessary optimizations have not yet been performed.

When using TCP/IP for communication, MPICH-G incurs a similar overhead for zero-length messages (69 μsec) but now attains 61 percent of the bandwidth achieved by IBM's MPI. The overheads associated with the layering of MPICH-G over Nexus and the bandwidth behaviors seen for Nexus and MPICH-G are comparable to those seen in the user space case.

We comment finally on the significance of the skip poll parameter. As discussed elsewhere [9], the performance of multimethod systems that depend on polling to detect incoming communications can be sensitive to the frequency with which different interfaces are polled. In the current case, a user space poll is cheap (less than one μsec), while an IP poll can cost 10s of microseconds. Hence, a simple round-robin strategy that polls the two interfaces in sequence will often delay the processing of incoming user space communications. We allow the user to control the polling strategy used by providing a parameter "skip-poll" that specifies how many "fast" polls are performed before a slow poll is performed. Hence, a very large skip-poll value such as 10,000 is a close approximation to the case when the slow protocol is not used at all, while skip-poll=0

Table 1. Preliminary performance results for MPICH-G: One-way message times on the LLNL IBM SP2

Communication Library	Skip poll	Latency (μsec)	Time (μsec) vs. Msg Size (bytes) 10	100	1K	10K	100K	1M
User space communication:								
IBM-MPI		25	27	32	64	284	1745	12714
IBM-MPL		24	26	30	63	235	1673	12681
MPICH-mpl		32	33	44	75	233	1630	12888
Nexus	10K	42	44	48	88	356	3944	35252
MPICH-G	10K	73	76	80	121	363	3249	35813
Nexus	0	161	162	167	224	701	6424	59886
MPICH-G	0	360	362	368	443	958	6458	57016
TCP/IP-based communication:								
IBM-MPI		131	134	143	251	976	4850	35272
IBM-MPL		129	133	141	251	718	4542	35061
MPICH-mpl		184	184	290	393	966	5800	35348
Nexus	10K	160	163	173	293	899	6993	57557
MPICH-G	10K	200	206	218	340	989	7058	58092
Nexus	0	287	289	294	430	1109	7856	62826
MPICH-G	0	530	544	558	693	1429	8141	62443

Table 2. Preliminary performance results for MPICH-G: Bandwidths on the LLNL IBM SP2

Communication Library	Skip poll	Latency (μsec)	Bandwidth (KB/sec) vs. Msg Size (bytes) 10	100	1K	10K	100K	1M
User space communication:								
IBM-MPI		25	349	3034	15142	34381	55935	76809
IBM-MPL		24	370	3219	15396	41401	58358	77005
MPICH-mpl		32	292	2211	12975	41868	59882	75769
Nexus	10K	42	221	1995	10975	27366	24757	27701
MPICH-G	10K	73	128	1217	8067	26896	30051	27268
Nexus	0	161	60	583	4355	13918	15200	16312
MPICH-G	0	360	26	265	2201	10184	15121	17127
TCP/IP-based communication:								
IBM-MPI		131	72	681	3884	10003	20132	27686
IBM-MPL		129	73	688	3882	13594	21498	27853
MPICH-mpl		184	52	336	2481	10099	16834	27626
Nexus	10K	160	59	563	3331	10854	13964	16966
MPICH-G	10K	200	47	446	2864	9869	13835	16810
Nexus	0	287	33	331	2271	8801	12430	15543
MPICH-G	0	530	17	174	1407	6833	11994	15639

corresponds to round-robin polling. We see from Tables 1 and 2 that the round-robin strategy performs significantly worse than skip-poll=0. Fortunately, experience shows that even quite small skip-poll values can provide acceptable overheads while providing easonable responsiveness for the different methods.

6 Future Work

We are working with colleagues to extend the MPICH-G implementation in a number of areas.

Shared-memory support. To date, we have explored the use of just two communication methods: user-space communications within an MPP and TCP/IP between MPPs. On computers such as the IBM SP, we can also exploit more efficient shared memory communications within SMP clusters, hence providing a total of three different communication methods. We are working with colleagues at USC/ISI to implement and evaluate this strategy.

Topology-aware communication operations. In heterogeneous grid environments, collective operations such as `MPI_REDUCE` can execute significantly faster if their implementation takes advantage of knowledge of the underlying system topology. For example, an `MPI_REDUCE` operation in the environment of Figure 1 might well first reduce within each SMP node, then within each MPP, and finally across MPPs. In order to implement such optimizations, the MPICH implementation requires information about the topology of the underlying machine. We are working with colleagues at LLNL to identify the required information and will extend the Globus device with additional functions that provide this information.

User-level communication structures can also take advantage of topology information. In principle, MPI's topology operations provide a basis for providing this information to applications. We plan to study whether these operations are indeed appropriate, or whether MPI extensions are needed to allow programmers to implement efficient applications in wide area environments.

Looking further into the future, we are interested in exploring more sophisticated techniques suitable for true wide area operation, for example exploiting Nexus support for multicast [21] and using network performance information (e.g., [22]) to adapt a combining tree structure in response to changing network loads.

MPI-2 extensions. The MPI-2 revisions to the MPI standard introduce a number of new features, including single-sided operations, dynamic process creation and attachment, and parallel I/O. All three of these extensions can, in principle, be incorporated into MPICH-G easily: The Nexus communication library used in MPICH-G provides a single-sided communication operation as a primitive; Globus mechanisms support dynamic process creation and attachment; and a remote I/O binding for MPI-IO has already been developed. However, numerous details remain to be worked out in each of these areas, and the MPICH framework itself must be extended to support these new features.

7 Summary

We have described MPICH-G, an implementation of the Message Passing Interface that uses services provided by the Globus toolkit to allow the use of MPI in wide area environments. MPICH-G masks details of underlying networks and computer architectures so that diverse distributed resources can appear as a single "`MPI_COMM_WORLD`." Any arbitrary MPI application can be started on heterogeneous collections of machines simply by typing `mpirun`: authentication, authorization, executable staging, resource allocation, job creation, startup, and routing of stdout and stderr are all handled for free.

We believe that MPICH-G is interesting not only in its own right but also as a demonstration and test case for Globus services. MPICH-G was constructed by adapting MPICH, a widely used MPI implementation for workstations and MPPs. This adaptation involved the use of various Globus tools, for security, remote file access, synchronized startup, and multimethod communication. Relatively few changes to MPICH were required to support the use of these tools.

MPICH-G passes the MPICH test suite and is hence ready for broad distribution and use. Work is continuing on point-to-point performance optimization, application development, and research investigations relating to collective operation performance, network topology information, MPI-2 implementation, and other issues.

Acknowledgments

We gratefully acknowledge the contributions of Steven Tuecke, Brian Toonen, and Joe Bester to the design of the MPICH-G system; the meticulous work performed by Olle Larsson on MPICH-G performance evaluation; and the assistance of Bill Gropp and Rusty Lusk on MPICH implementation issues. We are also grateful to the members of the Globus project team at Argonne National Laboratory and the University of Southern California's Information Sciences Institute for their help.

This work was supported in part 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; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; and by the National Science Foundation.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proc. 4th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1995.

[2] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[3] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, November 1995.

[4] Jian Chen and Valerie Taylor. Mesh partitioning for distributed systems. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[5] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The Network-Enabled Optimization System (NEOS) Server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[6] Tom DeFanti and Rick Stevens. Teleimmersion. In *[11]*, pages 131–156.

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

[8] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. A wide-area implementation of the Message Passing Interface. *Parallel Computing*, 1998. to appear.

[9] 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.

[10] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998.

[11] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] 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.

[13] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter HPC++ and the HPC++Lib Toolkit. Springer Verlag, 1997.

[14] 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.

[15] 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.

[16] William Johnston. Realtime widely distributed instrumentation systems. In *[11]*, pages 75–103.

[17] Olle Larsson. Implementation and performance analysis of a high-order CEM algorithm in parallel and distributed environments. Master's thesis, University of Houston, 1998.

[18] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[19] Paul Messina. Distributed supercomputing applications. In *[11]*, pages 55–73.

[20] Reagan Moore, Chaitanya Baru, Richard Marciana, Arcot Rajasekar, and Michael Wan. Data-intensive computing. In *[11]*, pages 105–129.

[21] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), 1996.

[22] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, Portland, Oregon, 1997. IEEE Press.