DOE Next Generation Internet
Applications, Network Technology, and Network Testbed Partnerships Program

Technologies and Tools for High-Performance Distributed Computing

Final Report

Nicholas T. Karonis

Department of Computer Science
Northern Illinois University
DeKalb, IL 60115

1 Introduction

In this project we studied the practical use of the MPI message-passing interface in advanced distributed computing environments. We built on the existing software infrastructure provided by the Globus Toolkit^{TM}, the MPICH portable implementation of MPI, and the MPICH-G integration of MPICH with Globus.

As a result of this project we have replaced MPICH-G with its successor MPICH-G2, which is also an integration of MPICH with Globus. MPICH-G2 delivers significant improvements in message passing performance when compared to its predecessor MPICH-G and was based on superior software design principles resulting in a software base that was much easier to make the functional extensions and improvements we did.

Using Globus services we replaced the default implementation of MPI's collective operations in MPICH-G2 with more efficient *multilevel topology-aware* collective operations which, in turn, led to the development of a new timing methodology for broadcasts [8]. MPICH-G2 was extended to inlcude client/server functionality from the MPI-2 standard [23] to facilitate remote visualization applications and, through the use of MPI idioms, MPICH-G2 provided application-level control of quality-of-service parameters as well as application-level discovery of underlying Grid-topology information. Finally, MPICH-G2 was successfully used in a number of applications including an award-winning record-setting computation in numerical relativity.

In the sections that follow we describe in detail the accomplishments of this project, we present experimental results quantifying the performance improvements, and conclude with a discussion of our applications experiences.

2 Details of Accomplishments

In this section we will discuss, in detail, each of the accomplishments that resulted from this project. We start with a general description of the Globus and MPICH-G2 startup mechanisms. While the work described there existed prior to this project, and therefore, is not part of our the work completed in this project, a brief discussion of these issues provides an important context in which we present our achievements. We continue with a description of our work in heterogeneous communication and application-level management of heterogeneity. We conclude this section with a description of our multilevel topology-aware collective operations.

2.1 Hiding Heterogeneity during Startup and Management

As illustrated in Figure 1 and discussed here, MPICH-G2 uses a range of Globus Toolkit services to address the various complex issues that arise in heterogeneous, multisite Grid environments, such as cross-site authentication, the need to deal with multiple schedulers with different characteristics, coordinated process creation, heterogeneous communication structures, executable staging, and collation of standard output. In fact, MPICH-G2 serves as an exemplary case study of how Globus Toolkit mechanisms can be used to create a Grid-enabled programming tool, as we now explain.

Prior to startup of an MPICH-G2 application, the user employs the *Grid Security Infrastructure* (GSI) [12] to obtain a (public key) proxy credential that is used to authenticate the user to each remote sites. This step provides a single sign on capability.

The user may also use the Monitoring and Discovery Service (MDS) [9] to select computers on the basis of, for example, configuration, availability, and network connectivity.

Once authenticated, the user uses the standard `mpirun` command to request the creation of an MPI computation. The MPICH-G2 implementation of this command uses the *Resource Specifi-*

DISCLAIMER

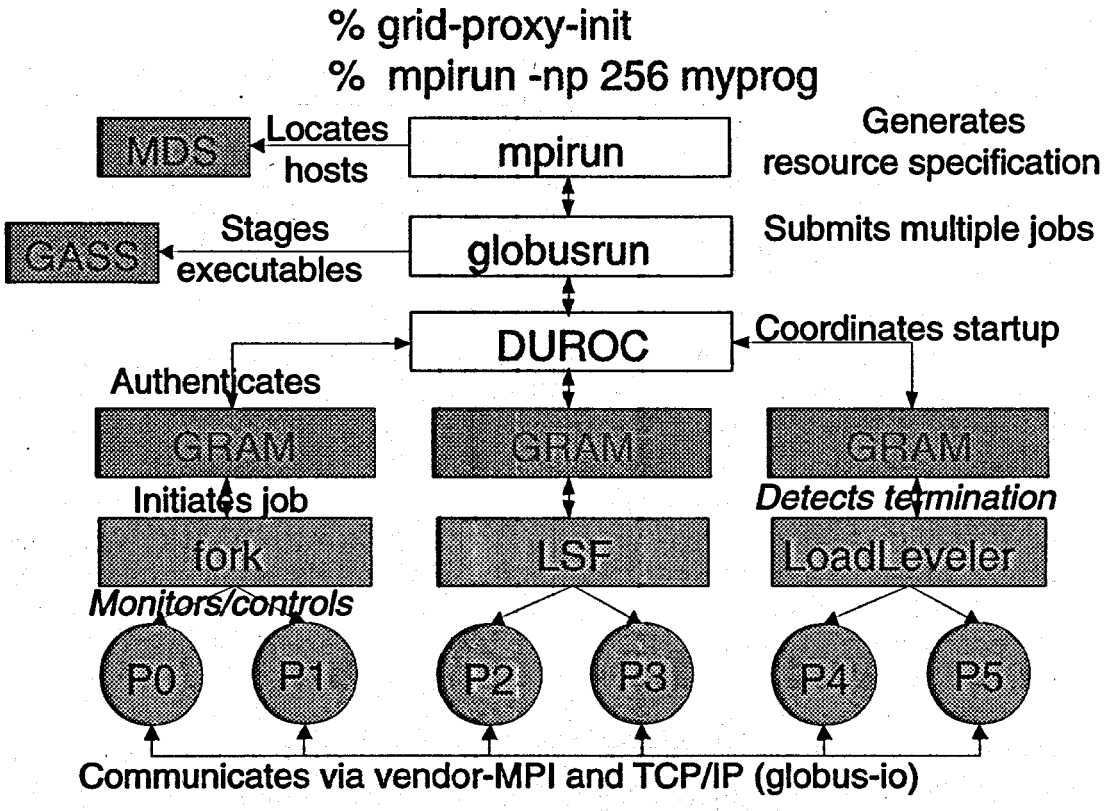

Figure 1: Schematic of the MPICH-G2 startup, showing the various Globus Toolkit components used to hide and manage heterogeneity. "Fork," "LSF," and "LoadLeveler" are different local schedulers.

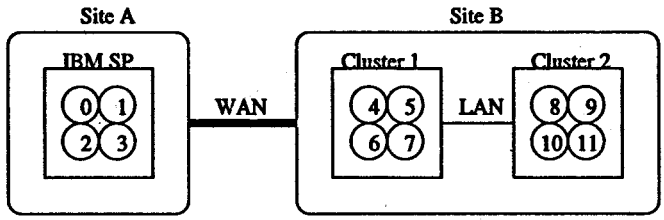

Figure 2: An example of an MPICH-G2 application running on a computational grid involving 4 processes on an IBM SP at Site A and 8 processes distributed evenly across two Linux clusters at Site B

cation Language (RSL) [6] to describe the job. In brief, users write *RSL scripts*, which identify resources (e.g., computers) and specify requirements (e.g., number of CPUs, memory, execution time, etc.) and parameters (e.g., location of executables, command line arguments, environment variables, etc.) for each. Based on the information found in an RSL script, MPICH-G2 calls a *co-allocation library* distributed with the Globus Toolkit, the Dynamically-Updated Request Online Coallocator (DUROC) [7], to schedule and start the application across the various computers specified by the user.

The DUROC library itself uses the *Grid Resource Allocation and Management* (GRAM) [6] API and protocol to start and subsequently manage a set of subcomputations, one for each computer. For each subcomputation, DUROC generates a GRAM request to a remote GRAM server, which authenticates the user, performs local authorization, and then interacts with the local scheduler to initiate the computation. DUROC and associated MPICH-G2 libraries tie the various subcomputations together into a single MPI computation.

GRAM will, if directed, use *Global Access to Secondary Storage* (GASS) [3] to stage executable(s) from remote locations (indicated by URLs). GASS is also used, once an application has started, to direct standard output and error (stdout and stderr) streams to the user's terminal, and to provide access to files regardless of location, thus masking essentially all aspects of geographical distribution except those associated with performance.

Once the application has started, MPICH-G2 selects the most efficient communication method possible between any two processes, using vendor-supplied MPI (*v*MPI) if available, or *Globus communication* (Globus IO) with *Globus Data Conversion* (Globus DC) for TCP, otherwise.

DUROC and GRAM also interact to monitor and manage the execution of the application. Each GRAM server monitors the life cycle of its subcomputation as it passes from pending to running and then to terminating, communicating each state transition back to DUROC. Each subcomputation is held at a DUROC-controlled barrier and is released from that barrier only after all subcomputations have started executing. Also, a request to terminate the computation ("control C") may be initiated by the user at which time DUROC and the GRAM servers, communicating via GRAM process control messages, terminate all processes.

After the processes have started, MPICH-G2 uses information specified in the RSL script to create *multilevel clustering* of the processes based on the underlying network topology. Figure 2 depicts an MPI application involving 12 processes distributed across three machines located at two sites. We depict 4 processes (`MPI_COMM_WORLD` ranks 0-3) on the IBM SP at Site A and 4 processes on each of two Linux clusters (`MPI_COMM_WORLD` ranks 4-7 and 8-11, respectively) at Site B. Each process in `MPI_COMM_WORLD` is assigned a *topology depth*. Processes that communicate

Rank		0	1	2	3	4	5	6	7	8	9	10	11
Depth		4	4	4	4	3	3	3	3	3	3	3	3
Colors	wide area	0	0	0	0	0	0	0	0	0	0	0	0
	local area	0	0	0	0	1	1	1	1	1	1	1	1
	system area	0	0	0	0	1	1	1	1	2	2	2	2
	*v*MPI	0	0	0	0								

Figure 3: An example of *depths* and *colors* used by MPICH-G2 to represent network topology in a computational grid.

using only TCP are assigned topology depths of 3 (to distinguish between wide area, local area, and intramachine TCP messaging), and processes that can also communicate using a *v*MPI have a topology depth of 4. Using these topology depths MPICH-G2 groups processes at a particular level through the assignment of *colors*. Two processes are assigned the same color at a particular level if they can communicate with each other at the network level.

Figure 3 depicts the *topology depths* and *colors* for the processes depicted in Figure 2. Those processes capable of communicating over *v*MPI, (i.e., those executing on the IBM SP), have a depth of 4, while the other processes, (i.e., those executing on a Linux cluster), have a depth of 3. Since all processes are on the same wide-area network, they all have the same *color* (0) at the wide-area level. Similarly, at the local-area level, all the processes at Site A are assigned one color (0), while all the processes at Site B are assigned another (1). This structure continues through the system-area level, where processes are assigned the same color if and only if they are on the same machine. Finally, processes that can communicate over a *v*MPI are assigned the same color at the *v*MPI level if and only if they can communicate directly with each other over the *v*MPI.

Topology depths and colors are used in the multilevel topology-aware collective operations and topology-discovery mechanism described in Sections 2.2 and 2.3, respectively.

2.2 Heterogeneous Communications

MPICH-G2 achieves major performance improvements relative to the earlier MPICH-G [10] by replacing Nexus [13], the multimethod, single-sided communication library used for all communication in MPICH-G, with specialized MPICH-specific communication code. While Nexus has attractive features (e.g., multiprotocol support with highly tuned TCP support and automatic data conversion), other attributes have proved less attractive from a performance perspective. MPICH-G2 now handles all communication directly by reimplementing the good things about Nexus and improving the others. The result, as we show in Section 3, is that we achieve performance virtually identical to vendor MPI and MPICH configured with the default TCP (ch_p4) device. We provide here a detailed description of the improvements and additions to MPICH-G used to achieve this impressive performance.

Increased bandwidth. In MPICH-G, each communication involved the copying of data to and from Nexus buffers in sending and receiving processes. MPICH-G2 eliminates these two extra copies in the case of intramachine messages where a vendor MPI exists. In this situation, sends and receives now flow directly from and to application buffers, respectively. In addition, for TCP messaging involving basic MPI datatypes (e.g., `MPI_INT`, `MPI_FLOAT`) the sending process also transmits directly from the application buffer.

Reduced latency for intramachine vendor MPI messaging. Multiprotocol support is achieved in Nexus by polling each protocol (TCP, vendor MPI, etc.) for incoming messages in a roundrobin fashion [11]. However, this strategy is inefficient in many situations: it is relatively expensive to poll a TCP socket and in practice it is often the case that many processes in a MPICH-G2 computation use only vendor MPI (for communicating with other processes on the same machine).

While this inefficiency can be reduced by adaptive polling [11] or by introducing distinct proxy processes [14, 20], MPICH-G2 takes a more direct approach, exploiting the knowledge about message source that is provided by TCP receive commands to eliminate TCP polling altogether in many situations. MPICH-G2 polls TCP *only* when the application is expecting data from a source that dictates, or might dictate (e.g., `MPI_Recv` specifies source=`MPI_ANY_SOURCE`), TCP messaging.

This avoidance of unnecessary polling when coupled with the need to guarantee progress on both the vendor MPI and TCP protocols leads to implementation decisions that can affect an application's point-to-point communication performance. Specifically, for processes executing on machines where a vendor MPI is available, the context in which the application calls `MPI_Recv` affects the manner in which MPICH-G2 implements that function, as follows:

- **Specified.** The source rank specified in the call to `MPI_Recv` explicitly identifies a process on the same machine (in the same vendor MPI job). Furthermore, no asynchronous requests are outstanding (e.g., incomplete `MPI_Irecv` and/or `MPI_Isend`). If these two conditions are met, MPICH-G2 implements `MPI_Recv` by directly calling the `MPI_Recv` of the underlying vendor MPI. This is the most favorable circumstances under which an `MPI_Recv` can be performed.
- **Specified-pending.** This category is similar to the *specified* category in that the `MPI_Recv` specifies an explicit source rank on the same machine. This time, however, one or more unsatisfied receive requests are present, and each such request specifies a source on the same machine. This situation forces MPICH-G2 to continuously poll (`MPI_Iprobe`) the vendor MPI for incoming messages. This scenario results in less efficient MPICH-G2 performance since the induced polling loop increases latency.
- **Multimethod.** Here the source rank for the `MPI_Recv` is `MPI_ANY_SOURCE` or `MPI_Recv` is called in the presence of unsatisfied asynchronous requests that require, or might require, TCP messaging. In this situation, MPICH-G2 must poll both TCP and the vendor MPI continuously. This is the least efficient MPICH-G2 scenario, since the relatively large cost of TCP polling results in even greater latency.

In Section 3, we present a quantitative analysis of the performance differences that result from these different structures.

More efficient use of sockets. The Nexus single-sided communication paradigm results in MPICH-G2 opening *two pairs of sockets* between communicating processes and using each pair as a simplex channel (i.e., data always flowing in one direction over each socket pair). MPICH-G2 opens a *single pair of sockets* between two processes and sends data in both directions. This approach reduces the use of system resources; moreover, by using sockets in the bidirectional manner in which they were intended, it also improves TCP efficiency.

Multilevel topology-aware collective operations. Early implementations of MPI's collective operations sought to construct communication structures that were optimal under the assumption that all processes were equidistant from one another [2, 5]. Since this assumption is unlikely to be valid in Grid environments, however, it is desirable that a Grid-enabled MPI incorporate

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int me, flag;
  int *depths;
  int **colors;
  MPI_Comm LANcomm, VcommA, VcommB;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Attr_get(MPI_COMM_WORLD, MPICHX_TOPOLOGY_DEPTHS, &depths, &flag);
  MPI_Attr_get(MPI_COMM_WORLD, MPICHX_TOPOLOGY_COLORS, &colors, &flag);

  MPI_Comm_split(MPI_COMM_WORLD, colors[me][1], 0, &LANcomm);
  MPI_Comm_split(MPI_COMM_WORLD, (depths[me] == 4 ? colors[me][3] : -1),
                 0, &VcommA);
  MPI_Comm_split(MPI_COMM_WORLD,
                 (depths[me] == 4 ? colors[me][3] : MPI_UNDEFINED),
                 0, &VcommB);

  MPI_Finalize();
}
```

Figure 4: An example MPICH-G2 application that uses *topology depths* and *colors* to create communicators that group processes into various topology-aware clusters.

collective operation implementations that take into account the actual topology. MPICH-G2 does this, and we have demonstrated substantial performance improvements for our *multilevel topology-aware* approach [18] relative both to topology-*un*aware binomial trees and earlier topology-aware approaches that distinguish only between "intracluster" and "intercluster" communications [17, 19].

As we explain in the next subsection, MPICH-G2's topology-aware collective operations are constructed in terms of topology discovery mechanisms that can also be used by topology-aware applications.

2.3 Application-Level Management of Heterogeneity

We have experimented within MPICH-G2 with a variety of mechanisms for application-level management of heterogeneity in the underlying platform. We mention two here.

Topology discovery. Once an MPI program starts, all processes can be viewed as equivalent, distinguished only by their rank. This level of abstraction is desirable from a programming viewpoint but makes it difficult to write programs that exploit aspects of the underlying physical topology, for example, to minimize expensive intercluster communications.

MPICH-G2 addresses this issue *within the standard MPI framework* by using the MPI communicator construct to deliver topology information to an application. It associates *attributes* with each MPI communicator to communicate this topology information, which is expressed within each process in terms of *topology depths* and *colors*, as described in Section 2.1.

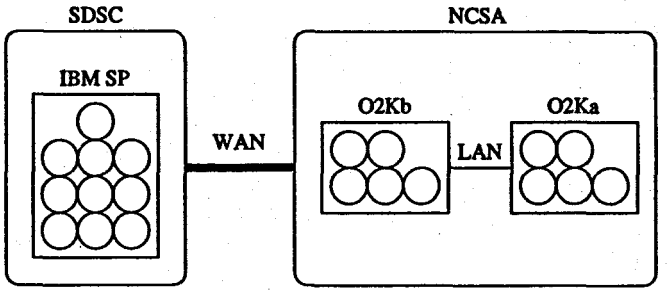

Figure 5: An example of a Grid computation involving 10 processes on one IBM SP at SDSC and another 10 processes distributed evenly across two SGI Origin2000s ($O2K_a$ and $O2K_b$) at NCSA.

MPICH-G2 applications can then query communicators to retrieve attribute values and structure themselves appropriately. For example, it is straightforward to create new communicators that reflect the underlying network topology. Figure 4 depicts an MPICH-G2 application that first queries the MPICH-G2-defined communicator attributes `MPICHX_TOPOLOGY_DEPTHS` and `MPICHX_TOPOLOGY_COLORS` to discover topology depths and colors, respectively, and then uses those values to create three communicators: `LANcomm`, which groups processes based on site boundaries, `VcommA`, which groups processes based on their ability to communicate with each other over *v*MPI, while placing all processes that cannot communicate over *v*MPI into a separate communicator, and `VcommB`, which groups the processes in much the same way as `VcommA`, but this time does not place processes that cannot communicate over *v*MPI in a communicator (i.e., `VcommB` is set to `MPI_COMM_NULL` for those processes).

Quality-of-service management. We have experimented with similar techniques for purposes of quality of service management [25]. When running over a shared network, an MPI application may wish to negotiate with an external resource management system to obtain dedicated access to (part of) the network. We show that communicator attributes can be used to set and initiate quality-of-service parameters between selected processes.

2.4 Multilevel Topology-Aware Collective Operations

Figure 5 depicts an MPI application involving 20 processes distributed over three machines located at the San Diego Supercomputer Center (SDSC) and the National Center for Supercomputing Applications (NCSA). We depict 10 processes on the IBM SP at SDSC and 5 processes on each of two Origin2000s, $O2K_a$ and $O2K_b$, at NCSA. The slowest communication is between sites, which uses TCP over a wide-area network, with faster communication between the O2Ks at NCSA, which uses TCP over their local-area network, and the fastest communication, of course, within each machine.

In the remainder of this section we describe a broadcast using first the topology-unaware implementation currently distributed with MPICH, then a 2-level topology-aware approach, and finally our multilevel topology-aware broadcast.

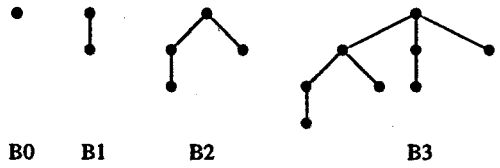

Figure 6: The binomial trees B_0 through B_3.

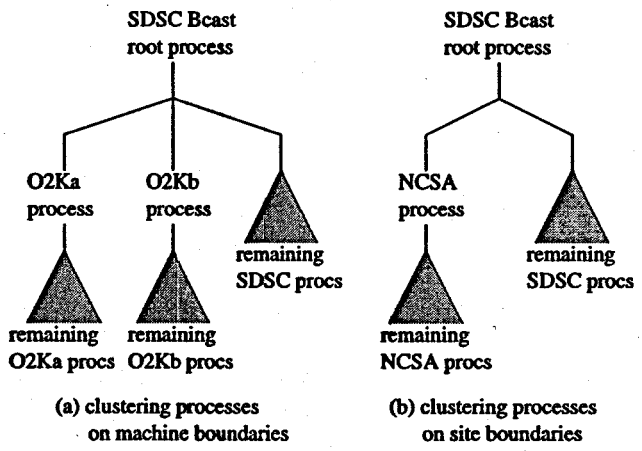

Figure 7: An example of two 2-level topology-aware broadcast trees rooted at SDSC spanning 2 Origin2000s ($O2K_a$ and $O2K_b$) at NCSA and an IBM SP at SDSC: (a) clustering processes on *machine boundaries* and (b) clustering on *site boundaries*.

2.4.1 A Topology-*Un*aware Broadcast

Topology-unaware implementations of broadcast, including the one distributed with MPICH, often make the simplifying assumption that the communication times between all process pairs in the computation are equal. Under this assumption the broadcast is often implemented by using a *binomial tree*.

A binomial tree B_k is an ordered tree (i.e., children of each node are ordered) of order $k \geq 0$ defined recursively. As shown in Figure 6, the binomial tree B_0 consists of a single node. The binomial tree B_k ($k > 0$) has a root with k children where the i^{th} child ($0 < i \leq k$) is the root of the binomial tree B_{k-i}. Figure 6 depicts the binomial trees B_0 through B_3.

When communication times between all process pairs in the computation are equal and have relatively low latency, Bar-Noy and Kipnis show that implementing a broadcast with a binomial tree has the desirable property that all processes will complete the broadcast at approximately the same time thus, achieving proper load balancing [2].

2.4.2 A 2-Level Topology-Aware Broadcast

Existing 2-level topology-aware approaches [17, 19] cluster processes into groups. The two natural choices for the machines depicted in Figure 5 are to cluster the processes based either on *machine boundaries*, creating three groups – the IBM SP, $O2K_a$, and $O2K_b$, or *site boundaries* creating two

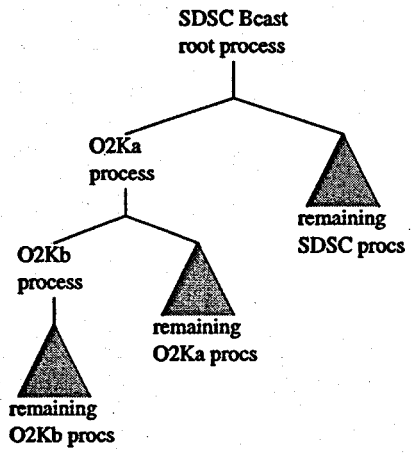

Figure 8: An example of a multilevel topology-aware broadcast tree rooted at SDSC spanning 2 Origin 2000s ($O2K_a$ and $O2K_b$) at NCSA and an IBM SP at SDSC.

groups – SDSC and NCSA. While both are reasonable choices and would improve performance when compared with the topology-*un*aware binomial tree distributed with MPICH, both choices ignore the disparity in network performance between the local- and wide-area networks. Consider, for example, a broadcast rooted at one of the processes at SDSC. Figure 7a depicts the broadcast tree of the 2-level approach when the processes are clustered on machine boundaries. The broadcast starts with the SDSC root process sending messages to designated processes on each of the O2Ks at NCSA, resulting in two messages traveling across the wide-area network, and concludes with broadcasts within each machine. By contrast, Figure 7b depicts the broadcast tree when the processes are clustered on site boundaries. In this case the root at SDSC sends a single message across the wide-area network to a process on one of the two O2Ks at NCSA and concludes with a broadcast within the IBM SP with another simultaneous broadcast across all the processes at NCSA, which would typically require multiple messages to travel across NCSA's local network.

2.4.3 A Multilevel Topology-Aware Broadcast

The multilevel topology-aware approach we present minimizes messaging across the slowest links *at each level* by clustering the processes at the wide-area level into site groups, and then within each site group, clustering processes at the local-area level into machine groups. Using the same broadcast example from Section 2.4.2, we depict in Figure 8 the broadcast tree used by a multilevel approach. Here the broadcast starts with the SDSC root process sending a single message across the wide-area network to one of the processes at NCSA, in Figure 8 we depict a process on $O2K_a$. The broadcast continues with the receiving process on $O2K_a$ sending a single message across NCSA's local network to a process on $O2K_b$ and the entire broadcast concludes with broadcasts within each machine. This multilevel clustering minimizes messaging over the slower wide- and local-area networks.

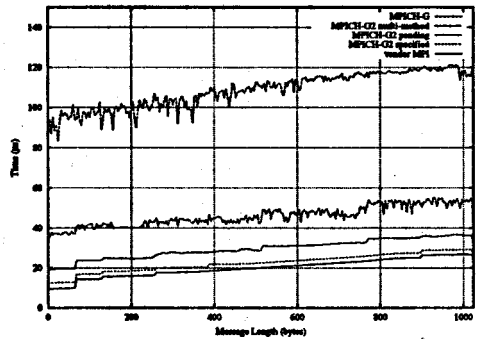

Figure 9: vMPI experiments – small message latency.

3 Performance Experiments

We present the results of detailed performance experiments that characterize the performance of MPICH-G2 and demonstrate the major improvements achieved relative to its predecessor, MPICH-G. We begin by looking at the performance of *intramachine* communication over a vendor MPI. Then, we examine performance when TCP is the only choice for communicating between a pair of processes. In all cases, mpptest [16], the performance tool included in the MPICH distribution, is used to obtain all results.

Following that we examine the benefits of our multilevel topology-aware strategy for MPI's collective operations by using MPI_Bcast to compare our multilevel approach to MPICH's default topology-*un*aware binomial tree and MagPIe, a 2-level topology-aware approach.

3.1 Point-to-point Operations

In this section we examine the performance improvements in point-to-point messaging resulting from removing Nexus in MPICH-G2.

3.1.1 Vendor MPI

Evaluating the performance of MPICH-G2 when using a vendor MPI as an underlying communication mechanism is not as simple as running a single set of ping-pong tests. As discussed earlier, the performance achieved by MPICH-G2 can be affected by outstanding requests and by the use of `MPI_ANY_SOURCE`. Therefore, we have divided the experiments into the three categories described in Section 2.2.

Our vendor MPI experiments were run on an SGI Origin2000 at Argonne National Laboratory. Both MPICH-G2 and MPICH-G were built using a nonthreaded, no-debug flavor of Globus 1.1.4 and performed intramachine communication via SGI's implementation of MPI.

One MPICH-G2 design goal was to minimize latency overhead for intramachine communication relative to an underlying vendor MPI. As can been seen in Figure 9, MPICH-G2 does an outstanding job in this regard: only a few extra microseconds of latency are introduced by MPICH-G2 when the source of the message is specified and no other requests are outstanding. In contrast, MPICH-G added approximately 80 microseconds of latency to each message, because the multiple steps required to implement the Nexus single-sided communication model.

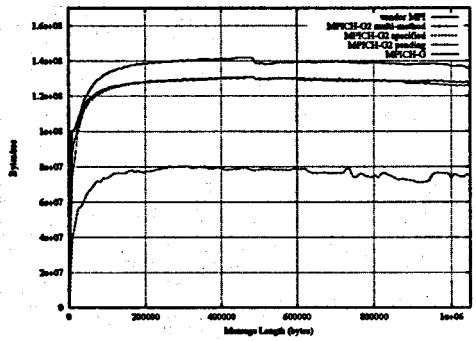

Figure 10: vMPI experiments – realized bandwidth.

The introduction of pending receive requests has a modest impact on MPICH-G2 message latencies. Messages falling into the *specified-pending* category incur slightly more overhead, as the MPICH-G2 progress engine must continuously poll (probe) the vendor MPI rather than blocking in a receive. Overall, MPICH-G2 latencies increase by several microseconds relative to the first case but are still far less than those of MPICH-G.

The use of `MPI_ANY_SOURCE` has the largest impact on MPICH-G2 performance. The additional cost is associated with having to poll TCP as well as the vendor MPI. Polling TCP increases the latency of messages by nearly 20 microseconds over those in the *specified-pending* category. While the increase is significant, however, these latencies are still considerably less than for MPICH-G.

While MPICH-G2 message latencies are affected by the use of `MPI_ANY_SOURCE` and pending receive requests, the realized bandwidths are largely unaffected. Figure 10 shows the bandwidths obtained for messages up to one megabyte. We see that the bandwidths for MPICH-G2 are nearly identical for all but small messages. While the large message bandwidths for MPICH-G2 are approximately 7% less than those for the the vendor MPI (for reasons we do not yet understand), they represent an improvement of more than 60% over MPICH-G.

3.1.2 TCP/IP

Performance optimization work on MPICH-G2 performed to date has focused on intramachine messaging when a vendor MPI is used as the underlying communication mechanism. The MPICH-G2 TCP/IP communication code has not been optimized. However, its performance is quite reasonable when compared with MPICH-G and to MPICH configured with the default TCP (ch_p4) device.

All TCP/IP performance measurements were taken using a pair of SUN workstations in Argonne's Mathematics and Computer Science Division. These two machines were connected to a local-area network via gigabit Ethernet. Both MPICH-G and MPICH-G2 were built using a non-threaded, no-debug flavor of Globus 1.1.4.

Figure 11 shows the small message latencies exhibited by all three systems. We see that for most message sizes, MPICH-G2 is 20% to 30% slower than MPICH/ch_p4, although the difference is much smaller for very small messages. We also see that MPICH-G2 latencies, in most cases, are somewhat less than those of MPICH-G.

The most notable data point is barely visible on the graph but emphasizes a clear optimization that is missing in MPICH-G2. The latency for zero-byte messages is 140 microseconds, while the latency for an eight-byte message is 224 microseconds. The reason for this large difference is that

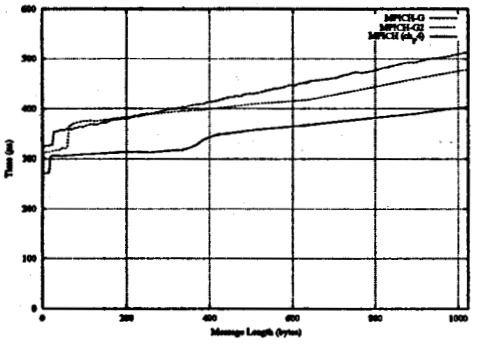

Figure 11: TCP/IP experiments – small message latency.

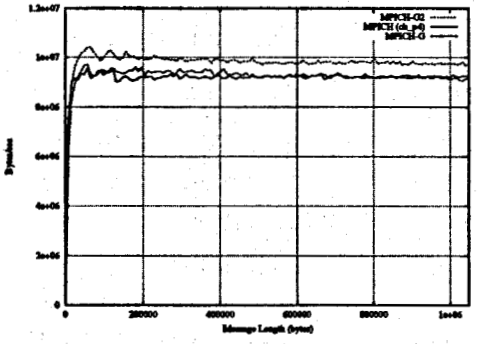

Figure 12: TCP/IP experiments – realized bandwidth.

MPICH-G2 currently uses separate system calls to send the message header and the message data. This data point suggests that by combining these two writes into a single vector write, we could reduce the latency of small messages significantly. While this difference might seem unimportant for machines separated by a wide-area network, it can be significant when MPICH-G2 is used to combine multiple machines with the same machine room or even at the same site.

Figure 12 shows the bandwidths obtained by all three systems for message sizes up to one megabyte. For large messages, we see that MPICH-G2 performs approximately 5% better than the other two systems. This improvement is a result of the message data being sent directly from the user buffer rather than being copied into a separate buffer before `write` is called. For preposted receives with contiguous data, further improvement is possible. Data for these receives can be read directly into the user buffer, avoiding a buffer copy that, at present, always takes place at the receiver.

3.2 Collective Operations

To demonstrate the advantages of our multilevel approach, we examine its effects on `MPI_Bcast`. The MPICH implementation of `MPI_Bcast` is based on binomial trees; hence, in a distributed heterogeneous environment like a computational Grid its performance is acutely sensitive to the distribution of the processes and the root of the broadcast. For example, in an application using

```
For (each message size M)
  MPI_Barrier(MPI_COMM_WORLD)
  if (MPI_COMM_WORLD rank == 0)
    t0 = get_time()
  For (r = 0; r < Nprocs; r ++)
    MPI_Bcast(root=r to MPI_COMM_WORLD message size M)
    ack_barrier()
  if (MPI_COMM_WORLD rank == 0)
    t1 = get_time()
    report message size M, time t1-t0
```

Figure 13: The broadcast timing application.

$P = 2^k$ processes distributed evenly across $C = 2^i, 0 \leq i \leq k$ clusters, a broadcast implemented using a binomial tree propagates the message down its longest path using at least $log_2 C$ intercluster messages and $log_2 \frac{P}{C}$ intracluster messages. In contrast, under certain intercluster network performance conditions described by Bar-Noy and Kipnis in their postal model, our multilevel method could be used to send 1 intercluster message and $log_2 \frac{P}{C}$ intracluster messages. Assuming an intercluster latency l_s sec and bandwidth b_s Kb/sec; and an intracluster latency l_f sec and bandwidth b_f Kb/sec, broadcasting a message of N Kb using the binomial tree conservatively takes $O((logC)(l_s + \frac{N}{b_s}) + (log\frac{P}{C})(l_f + \frac{N}{b_f}))$, whereas broadcasting the same message using our multilevel method takes only $O((l_s + \frac{N}{b_s}) + (log\frac{P}{C})(l_f + \frac{N}{b_f}))$.

We wrote a small MPI application (depicted in Figure 13) that times the broadcasts of messages of increasing size. To represent a broadcast with an arbitrary root, we timed how long it would take to broadcast each message of size M as each process in `MPI_COMM_WORLD` took its turn as the root. Also, in order to eliminate any potential pipelining that might occur between consecutive broadcasts, we inserted a barrier (`ack_barrier()`) after each broadcast in which all processes other than rank 0 `MPI_Send` an ACK message to process 0 and then wait to `MPI_Recv` a GO message. Process 0, after `MPI_Recv`'ing the ACK message from all the other processes, `MPI_Send`'s a GO message to each of the other processes, one at a time. We chose to write our own barrier rather than calling `MPI_Barrier` because we have reimplemented `MPI_Barrier` to reflect multilevel topology and we wished these tests to reflect the differences only in the broadcast implementations.

We conducted experiments running the MPI application depicted in Figure 13 on three computers: the IBM SP at the San Diego Supercomputer Center (SDSC-SP) and the IBM SP (ANL-SP) and SGI Origin200 (ANL-O2K) at Argonne National Laboratory. We compare our multilevel topology approach to the binomial tree provided by MPICH and include comparisons to the 2-level approach provided by MagPIe. We ran the application four times, each time using 16 processes on each of the three computers. These results are depicted in Figure 14. The curves labeled "MagPIe-machine" and "MagPIe-site" represent two runs using MagPIe version 2.0.1, each time with a different cluster definition. In our first MagPIe run ("MagPIe-machine") we defined three clusters, one for each computer, of 16 processes each. In our second MagPIe run ("MagPIe-site") we defined two clusters: an ANL cluster comprising the two ANL machines having 32 processes and an SDSC cluster comprising the SDSC-SP having only 16 processes.

Figure 14 shows there are significant benefits to the multilevel approach when compared with a simple binomial tree and even when compared with a 2-level approach as implemented by MagPIe. A multilevel view of the network allows an application to avoid slower channels *at each level.* In

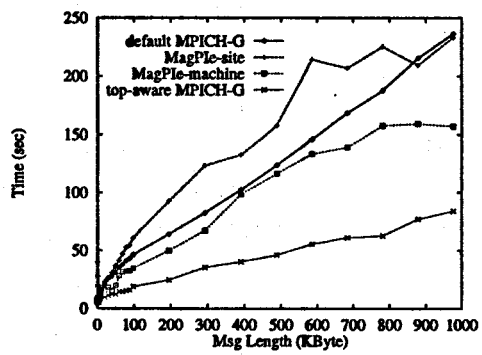

Figure 14: Original MPICH broadcast vs. topology-aware MPICH broadcast vs. MagPIe broadcast running 16 processes on the IBM SP at SDSC and 16 processes on each the IBM SP and SGI Origin2000 at ANL.

our experiments, the broadcast is optimized by sending one message across the wide-area network, then one message across the local-area network, and then many messages within each computer.

4 Application Experiences

MPICH-G2 has been used by many groups worldwide for a wide variety of purposes. Here we mention a few relevant experiences that highlight interesting features of the system.

One interesting use of MPICH-G2 is to run conventional MPI programs across multiple parallel computers within the same machine room. In this case, MPICH-G2 is used primarily to manage startup and to achieve efficient communication via use of different low-level communication methods. Other groups are using MPICH-G2 to distribute applications across computers located at different sites, for example, Taylor performing MM5 climate modeling on the NSF TeraGrid [26, 24], Mahinthakumar forming multivariate geographic clusters to produce maps of regions of ecological similarity [22], Larsson for studies of distributed execution of a large computational electromagnetics code [21], and Chen and Taylor in studies of automatic partitioning techniques, as applied to finite element codes [4].

MPICH-G2 has also been successfully used in demonstrations that promote MPI as an application-level interface to Grids for nontraditional distributed computing applications, for example, Roy et al. for studies in using MPI idioms for setting QoS parameters [25] and Papka and Binns for creating distributed visualization pipelines using MPICH-G2's client/server MPI-2 extensions [26, 24].

MPICH-G2 was awarded a 2001 Gordon Bell Award for its role in an astrophysics application used for solving problems in numerical relativity to study gravitational waves from colliding black holes [1]. The winning team used MPICH-G2 to run across four supercomputers in California and Illinois, achieving scaling of 88% (1,140 CPUs) and 63% (1,500 CPUs) computing a problem size five times larger than any other previous run.

5 Summary

We have described MPICH-G2, an implementation of the Message Passing Interface that uses Globus Toolkit mechanisms to support the execution of MPI programs in heterogeneous wide-area

environments. MPICH-G2 masks details of underlying networks, software systems, policies, and computer architectures so that diverse distributed resources can appear as a single `MPI_COMM_WORLD`. Arbitrary MPI applications 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 automatically via Globus Toolkit mechanisms. MPICH-G2 also enables the use of MPI features for user-level management of heterogeneity, for example, via the use of MPI's communicator construct to access system topology information. A wide range of successful application experiences have demonstrated MPICH-G2's utility in practical settings, both for traditional simulation applications and for less traditional applications such as distributed visualization pipelines.

While MPICH-G2 is already a sophisticated tool that is seeing widespread use, there are also several areas in which it can be extended and improved. Support for MPI-2 features, in particular dynamic process management, will be invaluable for Grid applications that adapt their resource usage to changing conditions and application requirements. This support will be provided as soon as it is incorporated into MPICH. More challenging is the design of techniques for effective fault management, a major topic for future research. Here we may be able to draw upon techniques developed within systems such as PVM [15].

Acknowledgments

We thank Olle Larsson and Warren Smith for early discussions and for prototyping the techniques that enable us to use vendor-supplied MPI. MPICH-G2 is, to a large extent, the result of our MPICH-G experiences. We therefore thank Jonathan Geisler, who originally designed and implemented MPICH-G while at Argonne, and George Thiruvathukal, who further developed MPICH-G also while at Argonne. We thank William Gropp, Ewing Lusk, David Ashton, Anthony Chan, Rob Ross, Debbie Swider, and Rajeev Thakur of the MPICH group at Argonne for their guidance, assistance, insight, and many discussions. We thank Sebastien Lacour for his efforts in conducting the performance evaluation and his many other contributions. His insight and ingenuity were invaluable to the implementation of the topology-aware components of MPICH-G2. We thank the San Diego Supercomputer Center and the National Center for Supercomputing Applications for providing access to their machines. Finally, we thank all the members of the Globus development team for their support, patience, and many ideas.

References

[1] G. Allen, T. Dramlitsch, I. Foster, M. Ripeanu N. T. Karonis, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with catus and globus. In *Proceedings of Supercomputing 2001*. IEEE Computer Society Press, 2001, winner Gordon Bell Award, Special Category.

[2] A. Bary-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 559–566, June 1992.

[3] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proc. IOPADS'99*. ACM Press, 1999.

[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] D.E. Culler, R. Karp, D.A. Patterson, A. Sahay. K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel compuation. In *Proceedings of the 4th SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[7] Karl Czajkowski, Ian Foster, and Carl Kesselman. Co-allocation services for computational grids. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing.* IEEE Computer Society Press, 1999.

[8] B. de Supinski and N. Karonis. Accurately measuring mpi broadcasts in a computational grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*, 2000.

[9] 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.

[10] 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*, 24(12):1735–1749, 1998.

[11] 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.

[12] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1998.

[13] 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.

[14] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogenous computing environment. In *Proc. EuroPVMMPI'98.* 1998.

[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] William Gropp and Ewing Lusk. Reproducible measurements of MPI performance characteristics. Technical Report ANL/MCS-P755-0699, Mathematics and Computer Science Division, Argonne National Laboratory, June 1999.

[17] P. Husbands and J.C. Hoe. MPI-StarT: Delivering network performance to numerical applications. In *Proceedings of Supercomputing '98*, November 1998.

[18] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, 2000.

[19] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang. MAGPIE: MPI's collective communcation operations for clustered wide area systems. In *Proceedings of Supercomputing '98*, November 1998.

[20] T. Kimura and H. Takemiya. Local area metacomputing for multidisciplinary problems: A case study for fluid/structure coupled simulation. In *Proc. Intl. Conf. on Supercomputing*, pages 145–156. 1998.

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

[22] G. Mahinthakumar, F. M. Hoffman, W. W. Hargrove, and N. Karonis. Multivariate geographic clustering in a metacomputing environment using globus. In *Proceedings of Supercomputing '99*. IEEE Computer Society Press, 1999.

[23] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

[24] Ncsa press release web page. `http://www.ncsa.edu/News/Access/Releases/011211.TeraGrid.html`.

[25] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-Service for message passing programs. In *Proceedings of Supercomputing 2000*. IEEE Computer Society Press, 2000.

[26] Teragrid web page. `http://www.teragrid.org`.