Design and implementation of message-passing services for the Blue Gene/L supercomputer

G. Almási
C. Archer
J. G. Castaños
J. A. Gunnels
C. C. Erway
P. Heidelberger
X. Martorell
J. E. Moreira
K. Pinnow
J. Ratterman
B. D. Steinmacher-Burow
W. Gropp
B. Toonen

The Blue Gene®/L (BG/L) supercomputer, with 65,536 dual-processor compute nodes, was designed from the ground up to support efficient execution of massively parallel message-passing programs. Part of this support is an optimized implementation of the Message Passing Interface (MPI), which leverages the hardware features of BG/L. MPI for BG/L is implemented on top of a more basic message-passing infrastructure called the ***message layer****. This message layer can be used both to implement other higher-level libraries and directly by applications. MPI and the message layer are used in the two BG/L modes of operation: the coprocessor mode and the virtual node mode. Performance measurements show that our message-passing services deliver performance close to the hardware limits of the machine. They also show that dedicating one of the processors of a node to communication functions (coprocessor mode) greatly improves the message-passing bandwidth, whereas running two processes per compute node (virtual node mode) can have a positive impact on application performance.*

Introduction

The Blue Gene*/L (BG/L) supercomputer is a new, massively parallel system developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). BG/L uses system-on-a-chip (SoC) integration [1] and a highly scalable architecture [2] to assemble a machine with 65,536 dual-processor compute nodes. Operating at a clock frequency of 700 MHz, BG/L will deliver 180 or 360 teraflops of peak computing power, depending on its mode of operation.

Each BG/L compute node can address only its local memory, making message passing the natural programming model for the machine. This paper describes how we designed and implemented application-level message-passing services for BG/L. The services include both an implementation of the Message Passing Interface (MPI) [3] and a more basic message-passing infrastructure called the *message layer*.

Our starting point for MPI on BG/L [4] is the MPICH2 library [5] from Argonne National Laboratory. MPICH2 is designed with a portability layer called the Abstract Device Interface, Version 3 (ADI3), which simplifies the job of porting it to different architectures. With this design, we could focus on optimizing the constructs that were of importance to BG/L.

BG/L is a feature-rich machine. A good implementation of message-passing services in BG/L must leverage those features to deliver high-performance communication services to applications. Its compute nodes are interconnected by two high-speed networks: a three-dimensional (3D) torus network that supports direct point-to-point communication [6] and a collective network to support broadcast and reduction operations. Those networks are mapped to the address space of user processes and can be used directly by a message-passing library. We show how we designed our message-passing implementation to take advantage of both types of memory-mapped networks.

Another important architectural feature of BG/L is its dual-processor compute nodes. A compute node can operate in one of two modes. In *coprocessor* mode, a single process, spanning the entire memory of the node, can use both processors by running one thread on each processor. In *virtual node mode*, two single-threaded

0018-8646/05/$5.00 © 2005 IBM

processes, each using half of the node memory, run on one compute node, with each process bound to one processor. This creates the need for two modes in our message-passing services, each with a different performance impact.

We validated our MPI implementation on BG/L by analyzing the performance of various benchmarks on 32-node and 512-node prototypes. We used microbenchmarks to assess how well MPI performed compared with the limits of the hardware and how different modes of operation within MPI compared with one another. We used the NASA Advanced Supercomputing (NAS) Parallel Benchmarks [7] to demonstrate the benefits of virtual node mode when executing computation-intensive benchmarks.

The rest of this paper is organized as follows. Next is an overview of the hardware and software architectures of BG/L, followed by a discussion of those details of BG/L hardware and software that were particularly influential to our MPI implementation. The architecture of our MPI implementation is then presented. A discussion of the basic architecture of the BG/L message layer is broken down into discussions of point-to-point and collective operations in the message layer. We then describe and discuss the experimental results on the prototype machines that validate our approach, and draw our conclusions.

BG/L supercomputer overview

The BG/L hardware [2, 8] and system software [9, 10] have been extensively described elsewhere. In this section, we present a short summary of the BG/L architecture to serve as a background to the sections following.

Hardware architecture

BG/L incorporates 65,536 compute nodes and 1,024 input/output (I/O) nodes based on a custom SoC design that integrates embedded low-power processors, high-performance network interfaces, and embedded memory. The basic unit of this architecture is a chip incorporating two standard 32-bit embedded IBM PowerPC* 440 (PPC440) processors with private L1 instruction and data caches, a small (2-KB) L2 cache and prefetch buffer, and 4 MB of embedded dynamic random access memory (DRAM). The PPC440 cores are not designed to support multiprocessor architectures. Therefore, their L1 caches are not coherent, and the processors do not implement atomic memory operations. To make up for this, the BG/L chip provides a custom synchronization device, the *lockbox* (a limited number of memory locations for fast atomic test-and-sets and barriers), and a fast software memory coherency mechanism (the *blind device*).

Each processor in the chip is augmented with a dual floating-point unit (FPU) consisting of two 64-bit FPUs operating in parallel. The dual FPU contains two 32×64-bit register files and is capable of dispatching two fused multiply–adds in every cycle, i.e., 2.8 Gflops per node at the 700-MHz target frequency. When both processors are used, the peak performance is doubled to 5.6 Gflops.

BG/L chips are also equipped with 1 GB of double-data-rate (DDR) memory. The chips are built into racks of 1,024 each (since their low-power design permits very dense packaging), adding up to a total of 1 TB of memory and 5.6 teraflops of floating-point performance.

The BG/L machine features five different networks (not all of which are described here). For the purposes of our MPI implementation, the most important network is the *torus*. Each of the 65,536 compute nodes is connected to its six neighbors through bidirectional links. The 64 racks in the full BG/L system form a $64 \times 32 \times 32$ 3D torus. The network hardware guarantees reliable, deadlock-free delivery of variable-length packets.

The fully equipped BG/L also contains 1,024 I/O nodes (one I/O node to 64 compute nodes) that connect the computational core with the external world. We call the collection formed by one I/O node and its associated compute nodes a processing set, or *pset*. Compute and I/O nodes are built using the same BG/L chip, but I/O nodes have the Ethernet network enabled.

The *collective network* provides high-performance broadcast and reduce operations spanning all compute and I/O nodes. It provides a latency of 2.5 μs for a 65,536-node system. Its name notwithstanding, the collective network also provides point-to-point messaging capabilities which are used to implement communication between I/O nodes and compute nodes. The collective is wired in a way that allows psets to be physically disjoint from one another with the I/O node acting as the root of the pset.

The *global interrupt* network provides configurable `OR` wires to perform full-system hardware barriers in 1.5 μs.

The torus, collective, and global interrupt network interfaces are mapped to the memory of the PPC440 cores. Network packets can be sent and received by reading and writing these addresses.

All of the torus, collective, and global interrupt links between midplanes (a 512-compute-node unit of allocation) are wired through a custom link chip that performs redirection of signals. The link chips provide isolation between independent partitions while maintaining fully connected networks within a partition.

Software architecture

User application processes run exclusively on compute nodes under the supervision of a custom compute node kernel (CNK). The CNK is a simple, minimalist runtime system written in approximately 5,000 lines of C++ that supports a single application running by a single user in

each BG/L node, reminiscent of PUMA [11]. It provides exactly two threads: one on each PPC440 processor. The CNK does not require or provide scheduling and context switching. Physical memory is statically mapped, protecting a few kernel regions from user applications. Porting scientific applications to run on this new kernel has been a straightforward process because we provide a standard `glibc` runtime system with most of the POSIX system calls.

Many of the CNK system calls are not directly executed in the compute node, but are *function-shipped*, i.e., forwarded through the collective to an I/O node where they are then executed. For example, when a user application performs a `write` system call, the CNK sends collective packets to the I/O node managing the pset. The packets are received on the I/O node by the console I/O daemon (CIOD). This daemon buffers the incoming packets, performs a GNU/Linux** write system call against a mounted file system, and returns the status information to the CNK through the collective network. The daemon also handles job start and termination on the compute nodes.

I/O nodes run a customized version of the PowerPC port of the GNU/Linux kernel (Version 2.4), similar to the one found in the MontaVista distribution [12]. They also implement I/O and process control services for the user processes running on the compute nodes. We mount a small ramdisk with system utilities to provide a root file system.

The system is complemented by a control system implemented as a collection of processes running in an external computer. All of the visible state of the BG/L machine is maintained in a commercial database. We have modified the middleware (such as IBM LoadLeveler* and mpirun) to operate through the CIOD-based control system, rather than launching individual daemons on all of the nodes.

Hardware and system software impact on MPI implementation

In this section we present a detailed discussion of the Blue Gene/L features that have a significant impact on the MPI implementation.

Torus network

The torus network guarantees deadlock-free delivery of packets. Packets are routed on an individual basis, using one of two routing strategies: a *deterministic* routing algorithm, in which all packets follow the same path along the x, y, z dimensions (in this order), and a minimal *adaptive* routing algorithm, which allows individual packets to make decisions about routing, resulting in potential out-of-order delivery of packets.

MPI ordering semantics enforce the order in which incoming messages are matched against the queue of posted messages. Adaptively routed packets may arrive out of order, forcing the MPI library to reorder them in software. Packet reordering is expensive because it involves memory copies and requires packets to carry additional sequence and offset information, reducing payload. On the other hand, deterministic routing leads to more network congestion and increased message latency, even on lightly used networks.

Each link in the torus network delivers two bits of raw data per central processing unit (CPU) cycle (0.25 bytes per cycle per link). Torus packets are multiples of 32 bytes, up to 256 bytes. The first 16 bytes of every packet contain destination, routing, and software header information. Thus, at most, 240 bytes of each packet can be used to transmit useful data. In addition, for every 256 bytes injected into the torus by the processors, 14 additional bytes traverse the wire with cyclic redundancy checks (CRCs), etc. Thus, the efficiency of the torus network is $\eta = 240/270 = 89\%$, or $\eta \times 0.25 = 0.22$ bytes per cycle per link. This translates to 154 MB/s/link at the 700-MHz chip frequency.

Adding up the raw bandwidth of the six incoming and six outgoing links on each node, we obtain $12 \times 0.25 = 3$ bytes per cycle per node. The corresponding bidirectional payload bandwidth is 2.64 bytes per cycle per node, or 1.8 GB/s/node.

The network guarantees reliable packet delivery. In any given link, it resends packets with errors, as detected by the CRC. Irreversible packet losses are considered to be fatal errors and result in the machine being taken offline for repairs. The communication library considers the machine to be completely reliable.

Collective network

The collective network serves a dual purpose. It is designed to perform MPI collective operations efficiently, but it is also the main mechanism for communication between I/O and compute nodes. The collective network supports point-to-point messages of fixed length (256 bytes), delivering four bits of raw data per CPU cycle (350 MB/s). It has reliability guarantees identical to the torus. The collective packet length is fixed at 256 bytes, all of which can be used for payload. Ten additional bytes are added to each packet for operation control and link reliability. Thus, the efficiency of the collective network is $\eta = 256/266 = 96\%$.

An arithmetic logic unit (ALU) in the collective network hardware can combine incoming and local packets using bitwise and integer operations, and forward the resulting packet along the network. This mechanism can be used to implement MPI collective operations. Collective operations, such as broadcast and reductions

[3], are done by having each node simply inject its contribution into the network and extract the result. Because the collective network ALU performs only integer operations, floating-point MPI reductions have to be performed in two phases, one to calculate the exponents and another for the normalized mantissas.

Packet routing on the collective network is based on packet classes. Collective network configuration is a global operation that requires the configuration of all nodes in a job partition. At the moment, we support only MPI collective operations that involve all of the compute nodes in a partition (i.e., operations on the `MPI_COMM_WORLD` communicator in MPI).

CPU/network interface

As described before, the torus, collective, and barrier networks are mapped into the processor memory. Torus and collective packets are read and written by the processor with special 16-byte single-instruction multiple-data (SIMD) load and store instructions of the custom FPUs. The SIMD load and store instructions used to read and write network packets require that memory accesses be aligned to a 16-byte boundary. However, the MPI library does not have control over the alignment of user buffers; moreover, a packet aligned correctly at the sender may not be aligned correctly at the receiver because of a different alignment of the receiving buffer. Consequently, the MPI library is forced to realign packets through memory-to-memory copies.

Network access overhead

Torus and collective packet reads into aligned memory take approximately 204 CPU cycles. Packet writes can take between 50 and 100 cycles, depending on whether the packet is being sent from cache or main memory.

CPU streaming memory bandwidth

For MPI purposes, we are primarily interested in the bandwidth for accessing large contiguous memory buffers. These accesses are typically handled by prefetch buffers in the L2 cache, resulting in a bandwidth of about 4.3 bytes per cycle.

We note that the available bandwidth of main memory and the torus and collective networks are of the same order of magnitude. Performing memory copies on this machine to get data into and from the torus results in reduced performance. It is imperative that network communication be zero-copy wherever possible.

Operating modes and cache coherency

As mentioned before, the two PPC440 processors in a chip are not cache-coherent. Software must take great care to ensure that coherency is correctly handled at the granularity of the CPU L1 cache lines: 32 bytes. Data structures shared by the CPUs should be aligned at 32-byte boundaries to avoid coherency problems.

This restriction has resulted in multiple operating modes for the machine. *Heater mode*, a term inherited from PUMA, is how we initially developed the system software and MPI library. In this node, one of the processors does not contribute to the computation, but merely sits in an idle loop. In *virtual node mode*, the processors become two MPI tasks, partitioning the memory and sharing the network resources. This has the obvious drawback of reducing a precious commodity—memory—even further; however, it completely solves the coherency problem because the processors do not share any memory. *Coprocessor mode* attempts to deal with the problem head-on by allowing the second processor to share memory and providing coherency in software.

Architecture of BG/L MPI

The Blue Gene/L MPI is an optimized port of the MPICH2 [5] library, an MPI library designed with scalability and portability in mind. **Figure 1** shows two components of the MPICH2 architecture: message passing and process management. MPI process management in BG/L is implemented using system software services. We do not discuss this aspect of MPICH2 further in this paper.

BG/L MPI is MPI-1.1-compliant and supports a subset of the MPI-2 standard. There are parts of MPI-2 (such as dynamic process management) that we do not plan to support on BG/L; other parts of MPI-2, such as one-sided communication, are not yet implemented, but we have plans to support them; at the time of this writing, MPI I/O development is still underway.

Figure 1 shows how BG/L MPI follows the overall architecture of MPICH2. The upper layers of BG/L MPI functionality are implemented entirely by MPICH2 code, which also provides the implementation of point-to-point messages, intrinsic and user-defined datatypes, communicators, and collective operations. MPICH2 interfaces with the lower layers of the implementation through the ADI3 layer [13], which consists of a set of data structures and functions that must be provided by the BG/L specific implementation.

The ADI3 interface deals with MPI requests (messages) and functions to send, receive, and manipulate these requests. Figure 1 depicts the ADI3 implementation on BG/L in a separate box. The ADI3 implementation relies on the *message layer*, an active message system [14–19], to transport arbitrary-sized messages between compute nodes using the torus network. The message layer is described in more detail in the following section. It interfaces with MPICH2 both by providing an application program interface (API) that ADI3 can call,

and through a set of event callback functions that the ADI3 implementation registers with the message layer.

The MPICH2 collective function implementation provides another hook-up point for BG/L-specific functions. We provide several implementations of optimized collectives using specific features of the hardware. Some of these implementations rely on the message layer, while others go to the *packet layer* to manipulate network hardware directly.

The packet layer is a very thin stateless layer of software that simplifies access to the BG/L network hardware. It provides functions to read and write the torus and collective networks, and to poll the state of the network. To help the message layer implement zero-copy messaging protocols, the packet layer provides convenience functions that allow software to "peek" at the header of an incoming packet without incurring the expense of unloading the whole packet from the network.

Message layer architecture

The message layer implements a functionality much like that of the low-level application program interface (LAPI): sending and dispatching of arbitrary-sized messages. It is divided into three main categories: basic functional support, point-to-point communication primitives (or protocols), and collective communication primitives. The base layer acts as a support infrastructure for the implementation of all of the communication protocols. We start with the most basic functional aspects of the message layer—the ability to send and receive packets and messages.

Initialization

The message layer takes full control of a number of hardware resources in the BG/L system—all of the torus hardware FIFOs (queues in which access becomes available according to the first in, first out rule). The message layer does not share these resources.

Initialization resets all state machines, primes the rank-mapping subsystem, and selects the operating mode (virtual node mode, heater mode, or coprocessor mode), on the basis of input parameters.

Advance loop

The advance loop is called whenever the message layer has to make progress sending and/or receiving packets. While the basic operating mode of the message layer is polling-based, the advance loop is capable of executing from an interrupt handler. The torus (and collective) hardware support interrupt-driven operation, although handling a hardware interrupt costs the processor about 10^3 cycles of context-switching overhead.

The message layer maintains a queue of messages being sent. Messages generate packets to be sent to the network.

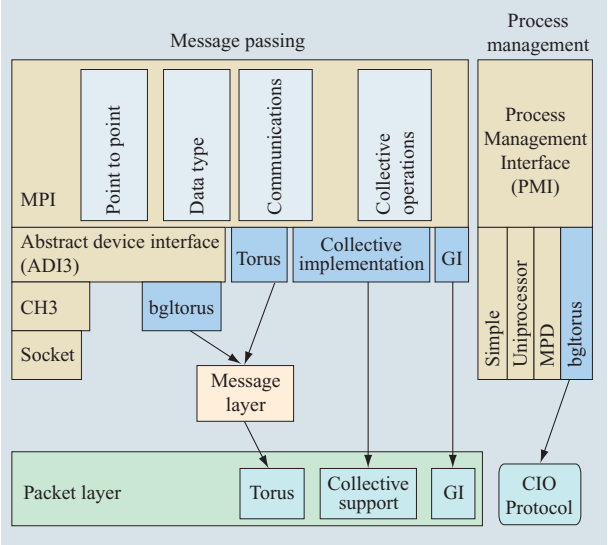

Figure 1

Blue Gene/L MPI software architecture. (GI = global interrupt; CIO = Control and I/O Protocol. CH3 is defined as the primary device distributed with MPICH2 for communication; MPD = multipurpose daemon.)

When a message is done sending packets, it is dequeued from the send queue. Incoming packets are received and dispatched on the basis of their types.

In coprocessor and virtual node mode, the network hardware allows simultaneous access to two disjoint sets of hardware FIFOs without compromising performance. In virtual node mode, an additional (virtual) queue is used to send messages *between* nodes.

Posting messages

Posting a message involves insertion into a send queue. The message layer supports several alternative ways to send a message.

For most packets sent through the torus network, e.g., the datastream of an MPI rendezvous message, packet ordering is irrelevant. These types of messages can be sent with fewer restrictions and adaptive network routing.

Eager messages and MPI control messages are posted to be sent with ordering guarantees. The message layer ensures non-overtaking packet semantics by two techniques: *FIFO pinning* assigns a single hardware FIFO to a particular message, ensuring that packets enter the network in the same order in which they are generated. The second technique, *deterministic routing* (mentioned previously in the section on the hardware and system software impact on MPI implementation), ensures that packets arrive at the destination in the same order in which they entered the network.

Process mapping

On a machine such as BG/L, the correct mapping of MPI applications to the torus network is a critical factor in maintaining application performance and scaling. **Figure 2** compares the scaling characteristics of the NAS Parallel Benchmark [7, 20, 21] Block Tridiagonal Solver (BT) on BG/L when mapped onto a mesh naively and optimally.

The message layer, like MPI, has a notion of process ranks, ranging between 0 and $N - 1$, where N is the number of processes participating. Message-layer ranks are the same as the global ranks in MPI. The message layer allows *arbitrary* mapping of torus coordinates to ranks. This mapping can be specified via an input file listing the torus coordinates of each process in increasing rank order. An example mapping file describing a possible mapping of eight ranks onto a $2 \times 2 \times 2$ mesh would look like this:

```
0 0 0 0
1 0 0 0
0 1 0 0
1 1 0 0
0 0 1 0
1 0 1 0
0 1 1 0
1 1 1 0
```

The default rank to torus coordinate mapping is called `XYZT` and corresponds to the lexical ordering of (x, y, z) triplets (in coprocessor or heater mode) or (x, y, z, t) quadruplets (in virtual node mode, with t representing the processor ID in each processor of a compute node).

Coprocessor mode support

To support the concurrent operation of the two non-cache-coherent processors in a compute node (described previously in the section on the hardware and software impact on MPI implementation), the message layer allows the use of the second processor both as a communication coprocessor and as a computation coprocessor. The message layer provides a non-L1-cached—and hence coherent—area of the memory to coordinate the two processors. This memory is called the *scratchpad*. The main processor supplies a pool of work units to be executed by the coprocessor. Work units can be functions that are *permanent*, executed whenever the coprocessor is idle, or *transient*, executed once and then removed from the pool. An example of a permanent function would be the one that uses the coprocessor to help with the *rendezvous* protocol. To start a transient function, one invokes the `costart` function provided by the message layer. The main processor waits for the completion of the work unit by invoking the `cojoin` function.

The coprocessor can also help with communication tasks. One of the permanent work units is a communication thread that runs constantly. Administrative data for messages received by the coprocessor is held in the scratchpad. Messages processed by the coprocessor are always aligned at cache-line boundaries, and, at the end of the reception, the two processors cooperatively enforce coherency in software.

Virtual node mode support

In virtual node mode, the kernel runs two separate processes (and hence, MPI tasks): one in each processor of a compute node. Some resources, such as the DDR memory and the torus network, are evenly split between the processors; others, such as the L3 cache, are shared.

The two MPI tasks running in the two CPUs of a compute node share the network but also have to communicate with each other. To solve this problem, we have implemented a virtual torus device, serviced by a virtual packet layer, in the scratchpad memory. Virtual FIFOs make portions of the scratchpad look like a send FIFO to one of the processors and a receive FIFO to the other. Access to the virtual FIFOs is mediated with help from the hardware lockboxes.

Virtual node mode doubles the number of tasks in the message layer; it also introduces a refinement in the addressing of tasks. As shown in the mapping file above, instead of being addressed with a triplet (x, y, z) denoting the torus coordinates, tasks are addressed with quadruplets (x, y, z, t) where t is the processor ID (0 or 1) in the current node. In coprocessor mode, t is always 0.

Message-layer support for MPI point-to-point communication

To facilitate the implementation of MPI, the message layer provides a number of protocols to support point-to-point message transmission. The protocols are each suited for particular bandwidth and latency requirements. The BG/L MPI implementation uses all of these protocols, depending on communication requirements.

The point-to-point messaging protocols are designed for easy interaction with MPICH2. To match the MPI relaxed completion semantics and allow for good performance, all message-layer primitives are nonblocking; the results of user actions are announced through callbacks registered by the user.

To send a message, the user of the message layer must be given the send buffer. In addition, at least one message object must be allocated. This is typically done by colocating message objects with MPI `Request` objects.

The message is initialized by the MPI call that sends the message, and it is then posted to the message layer. The message carries the address of the `done()` callback function that will be called when the send is complete; this callback will then update the MPI `Request` object accordingly.

At the receiving end, an incoming message triggers an upcall to MPI, which retrieves a `Request` object by consulting MPICH2 data structures. Further incoming packets will be directed to this `Request` object and update its status until the received message is complete.

Every messaging protocol knows how to *packetize* and *unpacketize* messages, i.e., how to break down a send buffer into torus and collective packets and how to restore the message from the packets at the receiving end.

Eager protocol

The eager protocol is one of the simplest in terms of both the programmer interface and implementation. The sender assumes that the receiver will be able to handle the message and simply starts sending packets.

To ensure correct MPI semantics, eager protocol packets are sent with ordering guarantees, i.e., by enforcing both FIFO pinning and packet ordering on the network. Unpacking eager protocol packets is simple: An offset counter keeps track of the current position in the message and unpacks incoming packets into the appropriate addresses.

Because of the deterministic routing, the eager protocol is unsuitable for longer messages, since it tends to create hot spots in the network, causing slowdowns.

"*One packet*" *protocol*

The "one packet" protocol is a simplified version of the eager protocol for cases in which the send buffer fits into a single packet. It is the ideal protocol for very short messages because it has a very low overhead.

Rendezvous protocol

The rendezvous protocol corrects deficiencies that are suffered by both the one-packet and eager protocols. In the rendezvous protocol, data packets are adaptively routed, avoiding the creation of hot spots in the network. Also, in coprocessor mode, rendezvous protocol packets can be received by the coprocessor, freeing up the main processor and allowing for better simultaneous use of all links.

In the rendezvous protocol, the sender first sends a scout packet to the receiver, asking for permission to send the rest of the data. To satisfy MPI ordering semantics, it is enough to ensure the order of the scout packet with respect to the rest of the traffic between the sender and receiver; rendezvous data packets can be routed dynamically. This is true because, while MPI requires messages to be *matched* at the receiver in the same order in which they were sent, no requirements exist with respect to the order in which messages are actually sent.

In our current implementation of the protocol, reception is executed by the coprocessor, subject to availability, cache coherency, and alignment constraints.

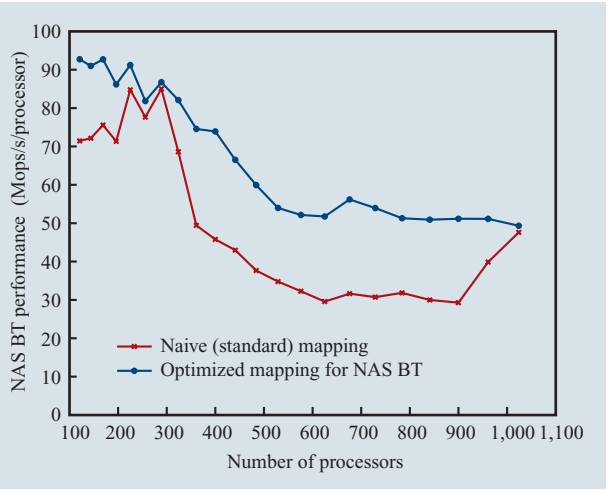

Figure 2

Comparison of NAS BT benchmark (class B) scaling characteristics when mapped onto the Blue Gene/L torus naively (red curve) and optimally (blue curve).

In particular, the receiver coprocessor is able to handle only packets carrying contiguous data aligned at cache-line boundaries (to avoid coherency problems).

Adaptive eager protocol

The adaptive eager protocol is a version of the eager protocol that uses no deterministically routed packets. To ensure MPI matching semantics, the receiver sends a confirmation packet every time a new message is matched. The sender has to wait for the confirmation before sending a new message.

The obvious drawback of this solution is that there is a lag time of at least one network round trip between subsequent message sends. This limits performance if the sender is sending many short messages to the same receiver. This situation, however, is unlikely to appear in well-tuned MPI programs, because it can be fixed very easily by sending one larger message instead of many small ones. We believe that the adaptive eager protocol will become more important as the BG/L machine is scaled beyond 20,000 processors, when the negative characteristics of the (deterministically routed) eager protocol will start to appear.

Virtual node mode protocol

The virtual node mode protocol helps communication between the two processors on the same node when the system operates in virtual node mode. Since each of the two processors can see the memory area of the other, this protocol involves sending a packet from the sender to the receiver indicating the address of the send buffer. The

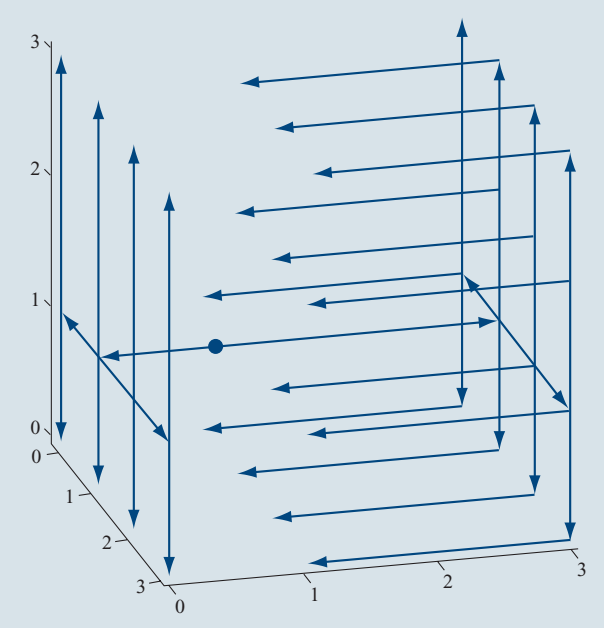

Figure 3

One of three independent deposit-bit-based broadcast routes in a three-dimensional mesh.

receiver then copies the buffer into local memory and notifies the sender (through a response packet) that the send buffer can be overwritten. This mechanism allows for much faster communication than through the scratchpad, because the message data is not cycled through uncached memory.

Message-layer support for MPI collective operations

Most MPI implementations, including MPICH2, typically implement collective communication in terms of point-to-point messages. On the BG/L platform, the default collective implementations of MPICH2 suffer from low performance for at least three reasons:

- The MPICH2 collectives are written for a crossbar-type network, not for special network topologies such as the BG/L torus network. Thus, the default implementation more often than not suffers from poor mapping (see the previous section on message-layer architecture).
- Point-to-point messaging in BG/L MPI has a high overhead because of the relative slowness of the CPU compared with the network speed. Thus, implementing, for example, MPI broadcast in terms of a series of point-to-point messages results in poor behavior at short message sizes, where overhead dominates the execution time of the collective.
- Some of the network hardware performance-enhancing properties are unused when only standard point-to-point messaging is employed. A good example of such a feature is the torus *deposit bit*, which lets packets be "deposited" on every node they touch on the way to the destination. Another such feature is the collective network, which is not used at all by point-to-point messaging.

Our work on collective communication in the message layer is far from complete. In this paper, we present the concepts behind optimized `MPI_Bcast` and `MPI_Alltoall[v]` algorithms.

The standard MPICH2 implementation of `MPI_Bcast` is a binomial tree algorithm; for large messages it is implemented as a binomial scatter followed by an allgather. This algorithm, mapped naively into the torus, causes several links to be used by multiple streams at once, resulting in low aggregate bandwidth.

We have implemented a version of MPI broadcast based on van de Geijn's optimized mesh algorithm [22], which uses the multicast capability of the torus hardware to broadcast packets in a rectangular region of the mesh/torus. The algorithm is pipelined at a packet level. **Figure 3** shows the route of a single packet in a $4 \times 4 \times 4$ mesh. The route is fairly complicated, but allows up to three such routes to be set up in the same 3D mesh without any two routes sharing a torus link. The algorithm can then cut the message into three equal parts, allowing broadcast to happen up to three times faster on this network.

The expected performance of the optimized torus broadcast algorithm depends on the number of simultaneous edge-disjoint spanning trees that can be found in a rectangular region. In addition, performance is limited by synchronization problems (as discussed by Watts and van de Geijn [22]), and by the load on the processors, which have to keep up with considerable incoming and outgoing traffic on the links.

We have also written a version of `MPI_Alltoall[v]` that uses the message layer directly and optimizes the injection of packets to achieve high network efficiency. While our algorithm is conceptually equivalent to the greedy "post everything/wait" algorithm also employed by MPICH2, it yields better performance because, at the message-layer level, we are able to optimize packet injection to even out network traffic and we are able to use the memory system prefetch engines to obtain faster access to read buffers.

Our short-term future plans include implementations of `MPI_Barrier`, `MPI_Allgatherv`, and `MPI_Allreduce`.

Our primary focus is on these primitives because they are in demand by the people doing application tuning on BG/L today. In particular, broadcast, allgather, and barrier are heavily used by the High-Performance Linpack (HPL) benchmark [23], which determines the Top500 [24] ranking of BG/L.

Performance analysis

In this section, we discuss the performance characteristics of the MPI library. We present microbenchmark results that analyze different aspects of our current MPI implementation. We compare different message-passing protocols. We compare performance between coprocessor mode and virtual node mode. Finally, we measure the performance of BG/L-specific optimized implementations of some MPI collectives.

To measure performance, we used various microbenchmarks, some written directly on top of the message layer, others on top of MPI. These are some of the same benchmarks we actually used in the development of the message layer and MPI.

All performance figures presented in this paper were measured on second-generation BG/L systems running at 700 MHz. Most of our microbenchmark runs were made on 32-node systems. Scalability studies were performed on systems consisting of up to 512 nodes.

Table 1 shows the half-round-trip latency of one-byte messages sent with all of the four point-to-point protocols. Latencies were measured between two nearest neighbors in the torus, with both message-layer and MPI versions of Turner's pong program [25]. The one-packet protocol has the lowest overhead, about 1,600 cycles. The highest overhead by far belongs to the rendezvous protocol, with the two eager variants in the middle of the range. When measured from within MPI, the latency numbers increase drastically because of the additional software overhead. All measurements are shown both in cycles and in microseconds.

MPI adds approximately 750 cycles of overhead in the case of the one-packet protocol and more than 1,300 cycles in the case of the eager protocol. In the case of the adaptive eager protocol, MPI overhead also measures the time required to obtain the next token from the receiver; hence, MPI time more than doubles compared with the message-layer timing.

Figure 4(a) shows half-round-trip latency as a function of the Manhattan distance[1] between the sender and the receiver in the torus. The figure clearly shows a linear dependency, starting with 3.35 μs for nearest-neighbor nodes and increasing by 90 ns with every hop.

[1] The "Manhattan distance" between two points is the sum of the (absolute) differences of their coordinates. Informally, it is like having to walk on a grid—something like city blocks in New York—to get to your destination.

Table 1 Round-trip latency comparison of all protocols. (© Springer Verlag LNCS 3149,3241, 2004. Reprinted with permission.)

Protocol name	*Message layer*		*MPI*	
	Cycles	μs	Cycles	μs
One-packet	1,600	2.29	2,350	3.35
Eager	2,700	3.86	4,000	5.71
Adaptive eager	3,300	4.71	11,000	15.71
Rendezvous	12,000	17.14	17,500	25.0

Point-to-point message bandwidth

Figure 4(b) shows MPI bandwidth measured on a single bidirectional link of the machine (sending and receiving at the same time). The figure shows the raw bandwidth limit of the machine running at 700 MHz (2 links $\times$ 175 = 350 MB/s), the net bandwidth limit ($\eta \times 2 \times 175$ = 310 MB/s), and the measured bandwidth as a function of message size. The figure shows good performance for relatively short message sizes: half bandwidth is reached for messages of 500 bytes.

Figures 4(c), **4(d)**, **4(e)**, and **4(f)** compare the multilink performance of the eager, adaptive eager, and rendezvous protocols, the latter with and without the help of the coprocessor. We can observe the number of simultaneous active connections that a node can handle. This is determined by the amount of time spent by the processor handling each individual packet belonging to a message; when the processor cannot handle the incoming and outgoing traffic, the network backs up.

In the case of the eager and rendezvous protocols, without the help of the coprocessor, the main processor is able to handle two bidirectional links simultaneously. The adaptive eager protocol, which is the least optimized at the moment, cannot handle even two links. In any case, when network traffic increases, the processor becomes a bottleneck, as shown by Figures 4(c), 4(d), and 4(e).

Figure 4(f) shows the effect of the coprocessor helping out in the rendezvous protocol: MPI is able to handle the simultaneous traffic of more than three bidirectional links in this case.

Coprocessor and virtual node modes compared

Figure 5(a) shows a comparison of per-task performance in coprocessor and virtual node modes. We ran a subset of the out-of-the-box version of Class B NAS Parallel Benchmarks [21] on a 32-compute-node subsystem of the 512-node BG/L prototype. The numbers of MPI tasks in coprocessor mode that we used are as follows: 25 for BT and SP and 32 for the other benchmarks. In virtual node mode, we used 64 MPI tasks for all benchmarks.

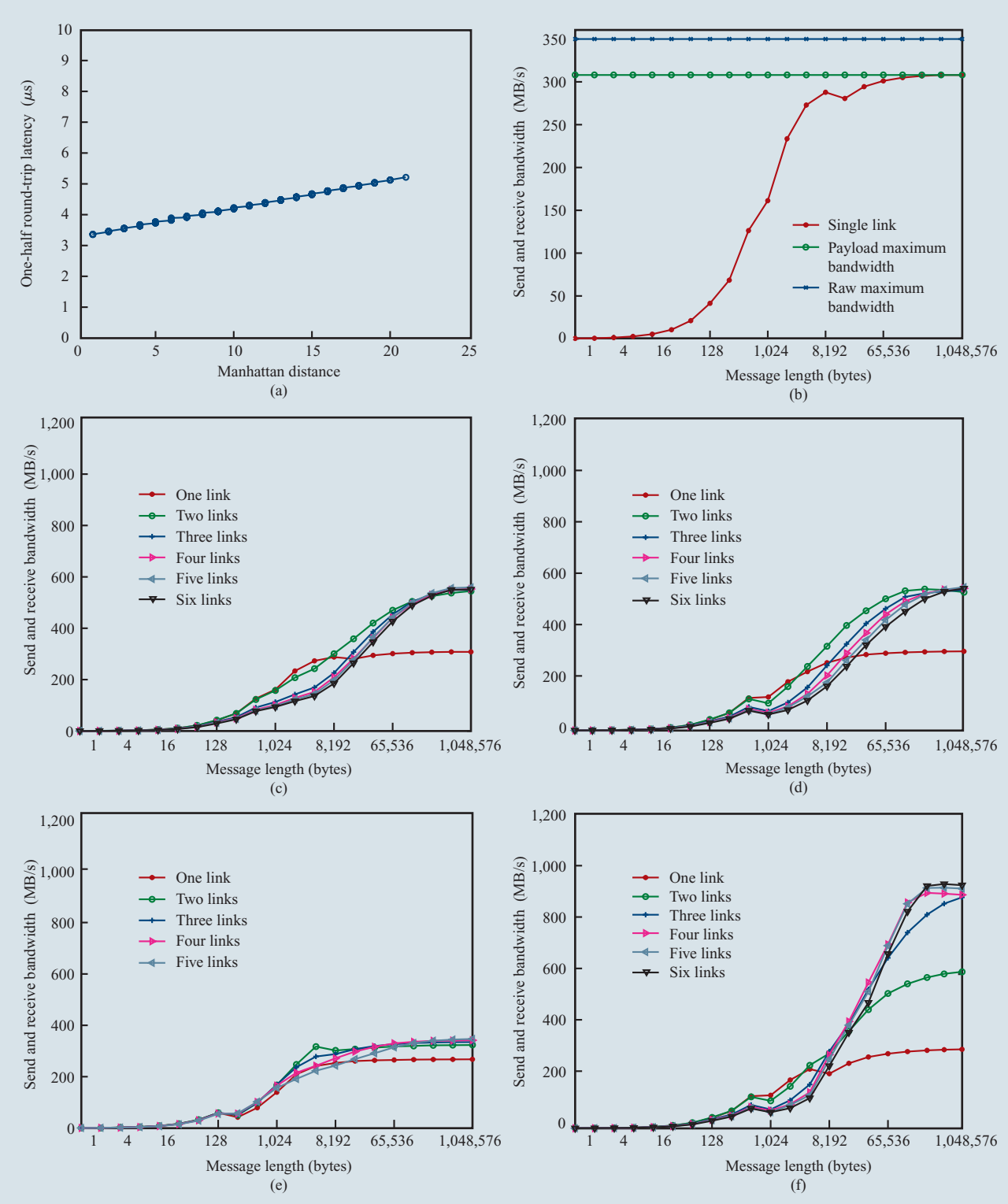

Figure 4

(a) Round-trip latency as a function of Manhattan distance. Comparison of multilink bandwidth performance of MPI protocols: (b) single link measured MPI bandwidth, (c) eager protocol, (d) rendezvous protocol, (e) adaptive eager protocol, and (f) rendezvous protocol with coprocessor support. (©2004, Springer Verlag LNCS 3149,3241. Reprinted with permission.)

Ideally, per-task performance in virtual node mode would be equal to that in coprocessor mode, resulting in a net doubling of total performance (because of the doubling of tasks executing). However, because of sharing node resources—including the L3 cache, memory bandwidth, and communication networks—individual processor efficiency degrades between 2% and 20%, resulting in less than ideal performance results. Nevertheless, the improvement warrants the use of virtual node mode for these classes of computation-intensive codes.

Optimized MPI broadcast on the torus and collective

Figure 5(b) compares the performance of three implementations of `MPI_Bcast`. The baseline is the default implementation of `MPI_Bcast` in MPICH2 using point-to-point messages. We compare this with a mesh-aware implementation of broadcast using point-to-point MPI messages. Finally, we have a mesh-aware implementation of broadcast directly in the message layer, this one using the deposit-bit feature of the torus network hardware. All three algorithms ran on the same $4 \times 4 \times 2$ mesh.

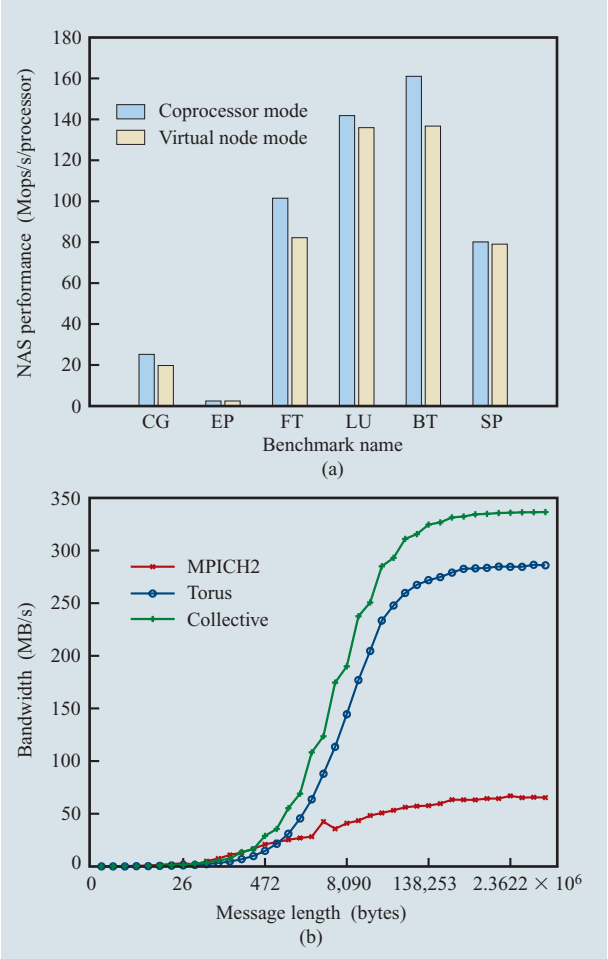

Figure 5

(a) Comparison of per-task performance in coprocessor and virtual node mode. (b) Performance comparison of broadcast implementations.

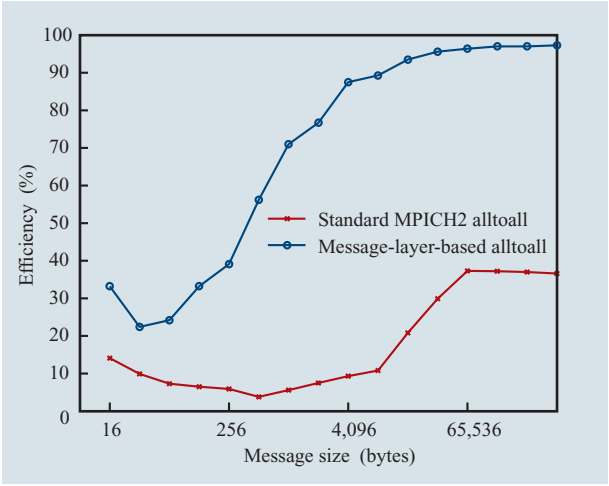

Figure 6

Comparison of `MPI_Alltoall` implementations.

The standard MPI broadcast outperforms both of the optimized algorithms for small message lengths but tops out at about 60 MB/s, less than half the bandwidth of a single link. The collective-based implementation reaches 100% efficiency (about 350 MB/s bandwidth); the torus-based implementation tops out at a bit less than two links' worth of bandwidth, or about 60% efficiency compared with its theoretical peak. The root cause for the relatively low efficiency is that the processor is unable to keep up with the torus traffic. We are planning to correct this, at least in coprocessor mode, by allowing the coprocessor to help with the broadcast.

Optimized MPI alltoall[v] on the torus

Figure 6 compares the performance of two implementations of `MPI_Alltoall`. The baseline is the unmodified MPICH2 implementation. The MPICH2 uses one of four strategies, depending on message size and communicator size, but all four algorithms are essentially designed for multiported systems [26].

We compare the baseline with a highly optimized message-layer-based implementation of alltoall which injects packets in an order that allows good memory use at the sender and evens traffic in the network.

We consider 100% efficiency to be represented by the theoretical cross-section bandwidth of the network (an $8 \times 8 \times 8$ mesh for this experiment). The optimized implementation achieves close to 100% efficiency, while the baseline goes up only to 30%.

Conclusions

The Blue Gene/L supercomputer represents a new level of scalability in massively parallel computers. Given the large number of nodes, each with its own private memory, we need an efficient implementation of message-passing services, particularly in the form of an MPI library, to effectively support application programmers. The BG/L architecture provides a variety of features that can be exploited in an MPI implementation, including the torus and collective networks and the two processors in a compute node.

This paper reports on the architecture of our MPI implementation and presents initial performance results. Starting with MPICH2 as a basis, we provided an implementation that efficiently uses the collective and torus networks and has two modes of operation for leveraging the two processors in a node. Key to our approach was the definition of a BG/L message layer that maps directly to the hardware features of the machine. The performance results show that different message protocols exhibit different performance behaviors, with each protocol being better for a different class of messages. The results also show that the coprocessor mode of operation provides the best communication bandwidth, whereas the virtual node mode can be very effective for the computation-intensive codes represented by the NAS Parallel Benchmarks.

Our MPI library is already being used by various application programmers at IBM and LLNL, and those applications are demonstrating very good performance and scalability in Blue Gene/L. Other application-level communication libraries, which would be implemented using the BG/L message layer, are also being considered for the machine. The lessons learned on this prototype will guide us as we move to larger and larger machine configurations.

Acknowledgments

The Blue Gene/L project has been supported and partially funded by the Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Lawrence Livermore National Laboratory Subcontract No. B517552.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds in the United States, other countries, or both.

References

1. G. S. Almási, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castaños, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, A. Deutsch, M. B. Dombrowa, W. Donath, M. Eleftheriou, B. Fitch, J. Gagliano, A. Gara, R. Germain, M. E. Giampapa, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, A. P. Lanzetta, D. Lieber, M. Lu, M. Mendell, L. Mok, J. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, R. Rand, R. Regan, R. Sahoo, A. Sanomiya, E. Schenfeld, S. Singh, P. Song, B. D. Steinmacher-Burow, K. Strauss, R. Swetz, T. Takken, P. Vranas, T. J. C. Ward, J. Brown, T. Liebsch, A. Schram, and G. Ulsh, "Cellular Supercomputing with System-on-a-Chip," *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, *Digest of Technical Papers*, Vol. 2, 2002, pp. 152–153.
2. N. R. Adiga et al., "An Overview of the Blue Gene/L Supercomputer," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2002, pp. 1–22; see *http://www.sc-conference.org/sc2002/*.
3. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference,* Second Edition, The MIT Press, Cambridge, MA, 2000.
4. G. Almási, C. Archer, J. G. Castaños, M. Gupta, X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen, "MPI on BlueGene/L: Designing an Efficient General Purpose Messaging Solution for a Large Cellular System," *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, 2003, pp. 252–261.
5. MPICH and MPICH2 homepage; see *http://www-unix.mcs.anl.gov/mpi/mpich*.
6. N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, "Blue Gene/L Torus Interconnection Network," *IBM J. Res. & Dev.* **49**, No. 2/3, 265–276 (2005, this issue).
7. NAS Parallel Benchmarks; see *http://www.nas.nasa.gov/Software/NPB*.
8. A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the Blue Gene/L System Architecture," *IBM J. Res. & Dev.* **49**, No. 2/3, 195–212 (2005, this issue).
9. G. Almási, R. Bellofatto, J. Brunheroto, C. Cascaval, J. G. Castaños, L. Ceze, P. Crumley, C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An Overview of the BlueGene/L System Software Organization," *Proceedings of the 9th International Euro-Par Conference*, 2003, pp. 543–555.
10. J. E. Moreira, G. Almási, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castaños, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart, "Blue Gene/L Programming and Operating Environment," *IBM J. Res. & Dev.* **49**, No. 2/3, 367–376 (2005, this issue).
11. L. Shuler, R. Riesen, C. Jong, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup, "The PUMA Operating System for Massively Parallel Computers," *Proceedings of the Intel Supercomputer Users' Group, Annual North America Users' Conference*, 1995.
12. Monta Vista Software, 1237 East Arques Avenue, Sunnyvale, CA 94085; see *http://www.mvista.com*.
13. W. Gropp, E. Lusk, D. Ashton, R. Ross, R. Thakur, and B. Toonen, MPICH Abstract Device Interface Version 3.4 Reference Manual, May 20, 2003 (draft); see *http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf*.

14. G. Chiola and G. Ciaccio, "Gamma: A Low-Cost Network of Workstations Based on Active Messages," *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, 1997.
15. S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of the International Conference on Supercomputing*, 1995, pp. 1–22.
16. T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995, pp. 40–53.
17. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp. 256–266.
18. M. Banikazemi, R. Govindaraju, R. Blackmore, and D. K. Panda, "MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems," *IEEE Trans. Parallel & Distr. Syst.* **12**, No. 10, 1081–1093 (October 2001).
19. R. Brightwell and L. Shuler, "Design and Implementation of MPI on Puma Portals," *Proceedings of the 2nd MPI Developer's Conference*, 1996, pp. 18–25.
20. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *Int. J. Supercomputer Appl.* **5**, No. 3, 63–73 (1991).
21. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA 94035, December 1995; see *http://www.nas.nasa.gov/Research/Reports/Techreports/1995/PDF/nas-95-020.pdf*.
22. J. Watts and R. van de Geijn, "A Pipelined Broadcast for Multidimensional Meshes," *Parallel Processing Lett.* **5**, No. 2, 281–292 (1995).
23. HPL Benchmark; see *http://www.netlib.org/benchmark/hpl/*.
24. J. Dongarra, H.-W. Meuer, and E. Strohmaier, TOP500 Supercomputer Sites; see *http://www.netlib.org/benchmark/top500.html*.
25. D. Turner, A. Oline, X. Chen, and T. Benjegerdes, "Integrating New Capabilities into NetPIPE," *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, 2003; see *http://www.scl.ameslab.gov/netpipe/np_euro.pdf*.
26. J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, "Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems," *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1994, pp. 298–309.

Received June 8, 2004; accepted for publication July 27, 2004; Internet publication April 12, 2005

George Almási *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gheorghe@us.ibm.com)*. Dr. Almási is a Research Staff Member at the IBM Thomas J. Watson Research Center. He received an M.S. degree in electrical engineering from the Technical University of Cluj-Napoca, Romania, in 1991 and an M.S. degree in computer science from West Virginia University in 1993. In 2001 he received a Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign; his thesis dealt with ways of optimizing and compiling MATLAB code. For the last three years, Dr. Almási has been working on various aspects of the Blue Gene system software environment, including the MPI communication libraries.

Charles Archer *IBM Systems and Technology Group, 3605 Highway 52 N., Rochester, Minnesota 55901 (archerc@us.ibm.com)*. Mr. Archer is a software engineer working on the Blue Gene/L project. He received a B.S. degree in chemistry and a B.A. degree in mathematics from Minnesota State University at Moorhead, and an M.S. degree in chemistry from Columbia University. He is currently a graduate student in computer science at the University of Minnesota. Mr. Archer has worked on the OS/400* PASE project and grid computing. His current role is development, optimization, and maintenance of the Blue Gene/L message-passing software stack.

José Gabriel Castaños *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (castanos@us.ibm.com)*. Dr. Castaños joined the Blue Gene project in 2000 after receiving his Ph.D. degree in computer science at Brown University. His initial assignment as a Research Staff Member involved the development of applications for high-performance computing. He later became one of the technical leaders of the Blue Gene/L systems software and worked on many of its components: integrated software development environment, simulators, kernels, runtime libraries, and management infrastructure. Dr. Castaños received his undergraduate degrees in systems analysis (1988) and operations research (1989) at the Universidad Católica, Buenos Aires, Argentina.

John A. Gunnels *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gunnels@us.ibm.com)*. Dr. Gunnels received his Ph.D. degree in computer science from the University of Texas at Austin. He joined the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center in 2001. His research interests include high-performance mathematical routines, parallel algorithms, library construction, compiler construction, and graphics processors. Dr. Gunnels has coauthored several journal papers and conference papers on these topics.

C. Chris Erway *Computer Science Department, Brown University, P.O. Box 1910, Providence, Rhode Island 02912 (cce@cs.brown.edu)*. Mr. Erway received a B.A. degree in computer science and music from Cornell University in 2002. He is currently a Ph.D. student in computer science at Brown University. Mr. Erway was a student software engineer at the IBM Thomas J. Watson Research Center working on Blue Gene system software.

Philip Heidelberger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (philiph@us.ibm.com)*. Dr. Heidelberger received a B.A. degree in mathematics from Oberlin College in 1974 and a Ph.D. degree in operations research from Stanford University in

1978. He has been a Research Staff Member at the IBM Thomas J. Watson Research Center since 1978. His research interests include modeling and analysis of computer performance, probabilistic aspects of discrete event simulations, parallel simulation, and parallel computer architectures. He has authored more than 100 papers in these areas. Dr. Heidelberger has served as Editor-in-Chief of the *ACM Transactions on Modeling and Computer Simulation*. He was the general chairman of the ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS) Performance 2001 Conference, the program co-chairman of the ACM SIGMETRICS Performance 1992 Conference, and the program chairman of the 1989 Winter Simulation Conference. Dr. Heidelberger is currently the vice president of ACM SIGMETRICS; he is a Fellow of the ACM and the IEEE.

Xavier Martorell *Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (xavim@us.ibm.com)*. Dr. Martorell received M.E. and Ph.D. degrees in computer science from the Technical University of Catalonia (UPC) in 1991 and 1999, respectively. Since 1992 he has lectured on operating systems and parallel runtime systems. He has been an associate professor in the Computer Architecture Department at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, and compilers for high-performance multiprocessor systems. Dr. Martorell has participated in several long-term research projects with other universities and industries, primarily in the framework of the European Union ESPRIT and IST programs, and also in collaboration with the IBM Thomas J. Watson Research Center.

José E. Moreira *IBM Systems and Technology Group, 3605 Highway 52 N., Rochester, Minnesota 55901 (jmoreira@us.ibm.com)*. Dr. Moreira received B.S. degrees in physics and electrical engineering in 1987 and an M.S. degree in electrical engineering in 1990, all from the University of São Paulo, Brazil. He received his Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign in 1995. Since joining IBM in 1995, he has been involved in several high-performance computing projects, including the teraflop-scale ASCI Blue-Pacific, ASCI White, and Blue Gene/L. Dr. Moreira was a manager at the IBM Thomas J. Watson Research Center from 2001 to 2004; he is currently the Lead Software Systems Architect for the IBM eServer Blue Gene solution. Dr. Moreira is the author of more than 70 publications on high-performance computing. He has served on various thesis committees and has been the chair or vice-chair of several international conferences and workshops. Dr. Moreira interacts closely with software developers, hardware developers, system installers, and customers to guarantee that the delivered systems work effectively and accomplish their intended missions successfully.

Kurt Pinnow *IBM Engineering and Technology Services, 3605 Highway 52 N., Rochester, Minnesota 55901 (kwp@us.ibm.com)*. Mr. Pinnow is a Senior Technical Staff Member at IBM Rochester, Minnesota. He graduated from the University of Wisconsin in 1969 with a B.S. degree in applied mathematics and engineering physics, and since 1972 has worked at IBM in various performance-related capacities. His performance interests include communications performance, file systems performance, database performance, and general operating system architecture. Over the years he has been involved in both major revisions and minor software adjustments of the I-series infrastructure as it relates to performance. Mr. Pinnow has been working on the Blue Gene team for the last 18 months. His interests on this project include MPI and file systems performance areas, where he is involved in ensuring that Blue Gene/L meets its performance goals.

Joseph Ratterman *IBM Engineering and Technology Services, 3605 Highway 52 N., Rochester, Minnesota 55901 (jratt@us.ibm.com)*. Mr. Ratterman is a software engineer at IBM Rochester. Since receiving a B.S. degree in computer engineering from Iowa State University in 2003, he has been working on various performance-related aspects of Blue Gene/L, focusing primarily on MPI communication performance.

Burkhard D. Steinmacher-Burow *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (steinmac@us.ibm.com)*. Dr. Steinmacher-Burow is a Research Staff Member in the Exploratory Server Systems Department. He received a B.S. degree in physics from the University of Waterloo in 1988, and M.S. and Ph.D. degrees from the University of Toronto in 1990 and 1994, respectively. He subsequently joined the Universitaet Hamburg and then the Deutsches Elektronen-Synchrotron to work in experimental particle physics. In 2001, he joined the IBM Thomas J. Watson Research Center and has since worked in many hardware and software areas of the Blue Gene research program. Dr. Steinmacher-Burow is an author or coauthor of more than 80 technical papers.

William Gropp *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439 (gropp@mcs.anl.gov)*. Dr. Gropp received a B.S. degree in mathematics from Case Western Reserve University in 1977, an M.S. degree in physics from the University of Washington in 1978, and a Ph.D. degree in computer science from Stanford in 1982. He subsequently held the positions of assistant professor (1982–1988) and associate professor (1988–1990) in the Computer Science Department at Yale University. In 1990 he joined the Numerical Analysis Group at the Argonne National Laboratory, where he is a Senior Computer Scientist and Associate Director of the Mathematics and Computer Science Division, a Senior Scientist in the Department of Computer Science at the University of Chicago, and a Senior Fellow in the Argonne–Chicago Computation Institute. Dr. Gropp's research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He has played a major role in the development of the Message Passing Interface Standard. He is a coauthor of the most widely used implementation of MPI, MPICH, and he was involved in the MPI Forum as a chapter author for both MPI-1 and MPI-2. He has written many books and papers on MPI, including *Using MPI* and *Using MPI-2*. Dr. Gropp is one of the designers of the PETSc parallel numerical library; he has developed efficient and scalable parallel algorithms for the solution of linear and nonlinear equations.

Brian Toonen *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439 (toonen@mcs.anl.gov)*. Mr. Toonen received his B.S. degree in computer science from the University of Wisconsin at Oshkosh in 1993 and his M.S. degree in computer science from the University of Wisconsin at Madison in 1997. He is a Senior Scientific Programmer with the Mathematics and Computer Science Division at the Argonne National Laboratory. His research interests include parallel and distributed computing, operating systems, and networking. Mr. Toonen is currently working with the MPICH team to create a portable, high-performance implementation of the MPI-2 standard. Prior to joining the MPICH team, he was a Senior Developer for the Globus Project.