Implementing MPI on the BlueGene/L Supercomputer

George Almási[1], Charles Archer[2], José G. Castaños[1], C. Chris Erway[1], Philip Heidelberger[1], Xavier Martorell[1], José E. Moreira[1], Kurt Pinnow[2], Joe Ratterman[2], Nils Smeds[1], Burkhard Steinmacher-burrow[1], William Gropp[3], and Brian Toonen[3]

[1] IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
{gheorghe,castanos,cerway,philip,xavim,jmoreira,nsmeds,steinmac}@us.ibm.com
[2] IBM Systems Group
Rochester, MN 55901
{archerc,kwp,jratt}@us.ibm.com
[3] Mathematics and Computer Science Division, Argonne National Laboratory
Argonne, IL 60439
{gropp,toonen}@mcs.anl.gov

Abstract. The BlueGene/L supercomputer will consist of 65,536 dual-processor compute nodes interconnected by two high-speed networks: a three-dimensional torus network and a tree topology network. Each compute node can only address its own local memory, making message passing the natural programming model for BlueGene/L. In this paper we present our implementation of MPI for BlueGene/L. In particular, we discuss how we leveraged the architectural features of BlueGene/L to arrive at an efficient implementation of MPI in this machine. We validate our approach by comparing MPI performance against the hardware limits and also the relative performance of the different modes of operation of BlueGene/L. We show that dedicating one of the processors of a node to communication functions greatly improves the bandwidth achieved by MPI operation, whereas running two MPI tasks per compute node can have a positive impact on application performance.

1 Introduction

The BlueGene/L supercomputer is a new massively parallel system being developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). BlueGene/L uses system-on-a-chip integration [6] and a highly scalable architecture [3] to assemble 65,536 dual-processor compute nodes. When operating at its target frequency of 700 MHz, BlueGene/L will deliver 180 or 360 Teraflops of peak computing power, depending on its mode of operation. BlueGene/L is targeted to become operational in early 2005.

Each BlueGene/L compute node can address only its local memory, making message passing the natural programming model for the machine. This paper describes how we implemented MPI [11] on BlueGene/L.

Our starting point for MPI on BlueGene/L [4] is the MPICH2 library [1], from Argonne National Laboratory. MPICH2 is architected 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 BlueGene/L.

BlueGene/L is a feature-rich machine and a good implementation of MPI needs to leverage those features to deliver high-performance communication services to applications. The BlueGene/L compute nodes are interconnected by two high-speed networks: a three-dimensional torus network that supports direct point-to-point communication and a tree network with support for broadcast and reduction operations. Those networks are mapped to the address space of user processes and can directly be used by a message passing library. We will show how we architected our MPI implementation to take advantage of both memory mapped networks.

Another important architectural feature of BlueGene/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 processes, each using half of the memory of the node, run on one compute node, with each process bound to one processor. This creates the need for two modes in our MPI library, with different performance impacts. In coprocessor mode, we dedicate one processor (and corresponding user-level thread) to perform communication functions. In virtual node mode, we run two MPI tasks per (physical) compute node.

We validate our MPI implementation on BlueGene/L by analyzing the performance of various benchmarks on our 512-node prototype. This prototype was built using first-generation BlueGene/L chips and operates at 500 MHz. We use microbenchmarks to assess how well MPI performs compared to the limits of the hardware and how different modes of operation within MPI compare to each other. We use the NAS Parallel Benchmarks to demonstrate the benefits of virtual node mode when executing computation-intensive benchmarks.

The rest of this this paper is organized as follows. Section 2 presents an overview of the hardware and software architectures of BlueGene/L. Section 3 discusses those details of BlueGene/L hardware and software that were particularly influential to our MPI implementation. Section 4 presents the architecture of our MPI implementation. Section 5 describes and discusses the experimental results on our 512 node prototype that validate our approach. Finally, Section 6 contains our conclusions.

2 An overview of the the BlueGene/L supercomputer

The BlueGene/L hardware [3] and system software [5] have been extensively described previously. In this section we present a short summary of the BlueGene/L architecture to serve as background to the following sections.

2.1 The BlueGene/L hardware architecture

The 65,536 compute nodes of BlueGene/L are based on a custom system-on-a-chip design that integrates embedded low power processors, high performance network interfaces, and embedded memory. The low power characteristics of this architecture permit a very dense packaging. One air-cooled BlueGene/L rack contains 1024 compute nodes (2048 processors) with a peak performance of 5.7 Teraflops.

The BlueGene/L chip incorporates two standard 32-bit embedded PowerPC 440 processors with private L1 instruction and data caches. A small 2 kB L2 cache for each processor acts as a prefetch buffer for the L1 caches. The chip also includes 4 MB of embedded DRAM, which can be partitioned into a shared L3 cache and a directly addressable scratchpad. The chip also contains a controlled for external DDR memory. The current packaging uses nine 512Mb chips for a total of 512MB per node. The standard PowerPC 440 cores are not designed to support multiprocessor architectures: the L1 caches are not coherent and the processor does not implement atomic memory operations. To overcome these limitations BlueGene/L provides a variety of custom synchronization devices in the chip such as the lockbox (a limited number of memory locations for fast atomic test-and-sets and barriers) and 16 KB of shared SRAM.

Each processor is augmented with a dual floating-point unit consisting of two 64-bit floating-point units operating in parallel. The dual floating-point unit contains two 32×64-bit register files, and is capable of dispatching two fused multiply-adds in every cycle. In one operating mode, only one of the CPUs is used for computation, while the other CPU is used for communications, giving a peak performance of 2.8 GFlop/s per node at a target speed of 700 Mhz. In computationally intensive codes, or in programs that have disjoint computation and communication phases, both CPUs can be used for computation, with a peak performance of 5.6 GFlop/s.

In addition to the 65,536 compute nodes, BlueGene/L contains a variable number of I/O nodes (1 I/O node to 64 compute nodes in the current configuration) that connect the computational core with the external world. Compute and I/O nodes are built using the same BlueGene/L chip, although the packaging is slightly different. In particular, I/O nodes have the Ethernet network enabled.

The main network used for point-to-point messages is the torus. Each compute node is connected to its 6 neighbors through bi-directional links. The 64 racks in the full BlueGene/L system form a $64 \times 32 \times 32$ three-dimensional torus. The network hardware in the chip guarantees reliable, deadlock free delivery of variable length packets. The torus network also provides simple broadcast capabilities by (optionally) leaving a copy of the payload in every node along its path.

On of the major problems of scaling to very large number of processors is performing global operations efficiently. BlueGene/L's solutions to this problem are the tree and global interrupt networks that connect all compute and I/O nodes. The tree is a configurable network for high performance broadcast and reduction operations, with a latency of 2.5 microseconds for a 65,536-node system. The tree also provides point-to-point capabilities and is the main mechanism to communication between I/O and compute nodes. The global interrupt network provides configurable OR wires to perform multinode hardware barriers in 1.5 microseconds in the full system.

All the torus, tree 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.

2.2 The BlueGene/L system software architecture

User application processes run exclusively on compute nodes under the supervision of a custom Compute Node Kernel (CNK). I/O nodes run the Linux operating system and implement I/O and process control services for the user processes running on the compute nodes. Control, management and configuration services running in external servers complete the system software design that we describe in this section.

The CNK is a simple, minimalist runtime system written in approximately 5000 lines of C++ that supports a single application running by a single user in each BG/L node. It also provides exactly two threads that are bound to each PPC440 processor. Therefore, the CNK does not require any scheduling or expensive context switch mechanism. The CNK statically maps all the available physical memory, protecting a few kernel regions from user applications. Porting scientific applications to run into this new kernel has been a straightforward process because we provide a standard Glibc runtime system with most of the Posix system calls.

The Linux kernel running in the I/O nodes (currently version 2.4.19) is a standard PPC kernel with minor changes in the booting code, context switch to save and restore the extended FPU registers, processor interrupts, memory layout and additional control registers. We mount a small ramdisk with system utilities to provide a root filesystem.

Many of the system calls are not directly executed in the compute node, but are function shipped through the tree to the I/O node. For example, when a user application performs a write system call, the CNK sends tree packets to the I/O node managing the processing set. The packets are received by a custom daemon running in the I/O nodes called ciod. This daemon buffers the incoming packets, performs a Linux write system call against a mounted filesystem (which verifies the privileges of the current user using standard Unix protocols), and returns the error code to the CNK through the tree. The ciod daemons (one in each I/O node) also receive job launch and termination commands from the control system, loading the application images, and broadcasting images and signals through the tree to the compute nodes.

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

3 Hardware and system software impact on MPI implementation

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

3.1 The torus network

The torus network, described in Section 2.1, is the main communication network for point-to-point messages. The network hardware guarantees deadlock-free delivery of packets. Packets are routed on an individual basis, using one of two routing strategies: a *deterministic* routing algorithm, all packets between two nodes follow the same path along the x, y, z dimensions (in this order); and a minimal *adaptive* routing algorithm, which permits better link utilization. In this case consecutive packets can follow different routes, and might reach their destination out of order.

Efficiency: The torus packet length is between 32 and 256 bytes in multiples of 32. The first 16 bytes of every packet contain destination, routing and software header information. Therefore, only 240 bytes of each packet can be used as payload. For every 256 bytes injected into the torus, 14 additional bytes traverse the wire with CRCs, sequence numbers, and retransmision counts. Thus the efficiency of the torus network is $\eta = \frac{240}{270} = 89\%$.

Link bandwidth: Because the network and the CPU run on synchronized clocks, the bandwidth of the torus links is best expressed in terms of CPU cycles. Each link delivers two bits of raw data per CPU cycle (0.25 bytes/cycle), or $\eta \times 0.25 = 0.22$ bytes/cycle of payload data. This translates into 110 MBytes/s/link at the current CPU frequency of 500 MHz (154 MBytes/s/link at the target 700 Mhz frequency).

Per-node bandwidth: Adding up the raw bandwidth of the 6 incoming + 6 outgoing links on each node, we obtain $12 \times 0.25 = 3$ bytes/cycle per node. The corresponding bidirectional payload bandwidth is 2.64 bytes/cycle/node.

Reliability: 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 catastrophic and stop the machine. As far as communication libraries are concerned, the network can be considered to be completely reliable.

Network ordering semantics: 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 before delivery. Packet re-ordering is expensive because it involves memory copies and requires packets to carry additional sequence and offset information. On the other hand, deterministic routing leads to more network congestion and increased message latency even on lightly used networks.

Packet priority: The torus network routing mechanism forwards packets incoming into the node with a higher priority than packets that are generated by the node. Through traffic over a particular link has the effect of throttling local traffic.

3.2 The tree network

The tree 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 tree supports point-to-point messages of fixed length (256 bytes), delivering 4 bits of raw data per CPU cycle (350 Mbytes/s at the target CPU frequency of 700 MHz).

Efficiency: The tree packet length is fixed at 256 bytes, all which can be used for payload. An additional 10 bytes are used with each packet for operation control and link reliability. Thus, the efficiency of the tree network is $\eta = \frac{256}{266} = 96\%$.

Collective operations: An ALU in the tree network hardware can combine incoming and local packets using bitwise and integer operations, and forward the resulting packet along the tree. Floating-point reductions can be performed in two phases, one to calculate the maximum exponent and another to add the normalized mantissae.

Packet routing on the tree network is based on packet classes. Tree network configuration is a global operation that requires the configuration of nodes not participating on the global operation, and for that reason, we only support operations on MPI_COMM_WORLD.

Reliability: The tree network has reliability guarantees identical to the torus.

3.3 CPU/network interface

The torus, tree and barrier networks are partially mapped into user-space memory. Torus and tree packets are read and written with special the 16-byte SIMD load and store instructions of the custom FPUs described in Section 2.1.

Alignment: The SIMD load and store instructions used to read and write network packets require that memory accesses be aligned to a 16 byte boundary. The MPI library does not have control over the alignment of user buffers. In addition, the sending and receiving buffer areas can be aligned at different boundaries. One simple, but inefficient solution in this case is to re-align packets by performing memory copies.

Network access overhead: Torus/tree packet reads into aligned memory take about 204 CPU cycles. Packet writes can take between 50 and 100 cycles, depending on the whether the packet is being sent from cache or main memory.

A full-size torus packet takes 270 bytes on the network. At the maximum network data rate of 3 bytes/cycle/node, a node has to handle every packet within $\frac{270}{3} = 90$ CPU cycles. This is plainly impossible with a single CPU, and only marginally possible with both CPUs in the node reading and writing packets. We conclude that in some modes of operation the bottleneck will not be network bandwidth but the ability of CPUs to process packets fast enough.

3.4 CPU streaming memory bandwidth

Another constraint of the machine is the available memory bandwidth. For MPI purposes we are interested mostly in the bandwidth for accessing large contiguous memory buffers. These accesses typically result in L1 cache misses, but are handled by prefetch buffers in the L2 cache, resulting in a bandwidth of about 4.3 bytes/cycle.

Comparing the memory bandwidth with the torus and tree network bandwidths, we note that they are in the same order of magnitude. We conclude that performing memory copies on this machine to get data into/from the torus results in reduced performance. It is imperative that network communication be zero-copy wherever possible.

3.5 Inter-core cache coherency

The two processors in a node are not cache coherent. Software must take great care to insure that coherency is correctly handled in software. Coherency handled at the granularity of the CPUs' L1 cache lines: 32 bytes. This means that structures not delimited by 32 byte boundaries cannot be shared by the CPUs.

4 Architecture of BlueGene/L MPI

The BlueGene/L MPI is an optimized port of the MPICH2 [1] library. MPICH2 was designed with scalability and portability in mind, and therefore was a good candidate for porting to BlueGene/L. Figure 1 shows the architecture of MPICH2 on BlueGene/L. We note that there are two major components of MPI. One is the message passing functionality offered to application programs, as represented by the functions in the MPI library. The other is the process management functionality, used to create, initialize, and terminate parallel jobs [2]. MPI process management in BlueGene/L is implemented using system software services. We do not discuss this aspect of MPICH2 further in this paper.

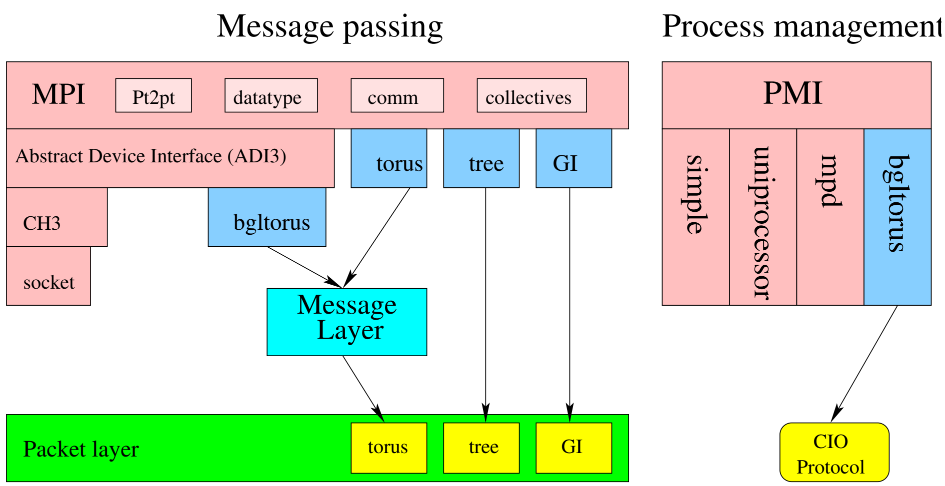

Fig. 1. BlueGene/L MPI software architecture.

Regarding the message passing functionality, the top layer of Figure 1 is implemented by MPICH2 code. MPICH2 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 Abstract Device Interface version 3 (ADI3) layer [9]. The ADI3 layer consists of a set of data structures and functions that need to be provided by the implementation. In BlueGene/L, the ADI3 layer is implemented using the BlueGene/L Message Layer, which in turn uses the BlueGene/L Packet Layer.

Certain BlueGene/L MPI operations bypass the ADI3 layer in order to more directly interface with the hardware, as represented by the torus, tree, and global interrupts. In the remaining of this section we focus on the main implementation path represented by ADI3, Message Layer, and Packet Layer. We describe how MPI works in each of the two operating modes of the machine, coprocessor and virtual node mode. We also discuss the different messaging protocols used in our implementation.

4.1 The ADI3 layer

The ADI layer is described in terms of MPI requests (messages) and functions to send, receive, and manipulate these requests. The BlueGene/L implementation of ADI3 is called `bgltorus`. It implements MPI requests in terms of Message Layer messages, assigning one message to every MPI request. Message Layer messages operate through callbacks. Messages corresponding to send requests are posted in a send queue. When a message transmission is finished, a callback is used to inform the sender. Correspondingly, there are callbacks on the receive side to signal the arrival of new messages. Those callbacks perform matching of incoming Message Layer messages to the list of MPI posted and unexpected requests.

4.2 Message Layer

The BlueGene/L Message Layer is an active message system [8, 10, 13, 14] that implements the transport of arbitrary-sized messages between compute nodes using the torus network. It can also broadcast data, using the broadcast capability of the torus as described in Section 2.1. The Message Layer breaks messages into fixed-size packets and uses the Packet Layer to send and receive the individual packets. At the destination, the Message Layer is responsible for reassembling the packets, which may arrive out of order, back into a message.

The Message Layer addresses nodes using the equivalent of `MPI_COMM_WORLD` ranks. Internally, it translates these ranks into physical torus x, y, z coordinates, that are used by the Packet Layer. The mapping of `MPI_COMM_WORLD` ranks to torus coordinates is programmable by the user. It can be used to optimize application performance by choosing a mapping that better supports the logical communication topology of the application.

Message transmission in the Message Layer is implemented using one of multiple available communication protocols, roughly corresponding to the protocols present in more conventional MPI implementations, such as the eager and rendezvous protocols. We describe those protocols further in Section 4.6.

The Message Layer is able to handle arbitrary collections of data, including non-contiguous data descriptors described by MPICH2 *data loops*. The Message Layer incorporates a number of complex data packetizers and unpacketizers that satisfy the multiple requirements of 16-byte aligned access to the torus, arbitrary data layouts, and zero-copy operations.

4.3 The Packet Layer

The Packet Layer is a very thin stateless layer of software that simplifies access to the BlueGene/L network hardware. It provides functions to read and write the torus/tree hardware, as well as to poll the state of the network. Torus packets typically consist of 240 bytes of payload and 16 bytes of header information. Tree packets consist of 256 bytes of data and a separate 32-bit header. 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.

4.4 Support for coprocessor mode

To support the concurrent operation of the two non-cache-coherent processors in a compute node, we have developed a *dual core library* that allows the use of the second processor both as a communication coprocessor and as a computation coprocessor.

The dual core library uses a portion of the (non-cached, and hence coherent) embedded DRAM scratchpad to coordinate the two processors. Through this area of memory, the main processor supplies a pool of work units to be executed by the coprocessor. Work units can be *permanent*, executed whenever the coprocessor is idle otherwise, or *single-shot* functions, 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 (Section 4.6). To start a single-shot function, one invokes the `co_start` function. The main processor waits for the completion of the work unit by invoking the `co_join` function.

4.5 Support for virtual node mode

The CNK in the compute nodes also supports a virtual node mode of operation for the machine. In that mode, the kernel runs two separate processes in each compute nodes. Node resources (primarily the memory and the torus network) are evenly split between both processes. In virtual node mode, an application can use both processors in a node simply by doubling its number of MPI tasks, without having to explicitly handle cache coherence issues. The now distinct MPI tasks running in the two CPUs of a compute node have to communicate to each other. We have solved this problem by implementing a virtual torus device, serviced by a virtual packet layer, in the scratchpad memory.

4.6 MPI messaging protocols

The one-packet protocol: optimizing latency We deployed the *one-packet* protocol in the Message Layer to handle short messages fitting into a single packet. That is, messages with length less than 240 bytes. Short message packets are always sent with deterministic routing, in order to avoid the issue of out-of-order arrival.

The eager protocol: maximizing single-link bandwidth The eager protocol is designed to deliver messages between 200 bytes and 10 kbytes in size. The receiver of an eager message has to accept and process each incoming packet. Since the network is reliable, no provisions for packet retransmission exist in the Message Layer.

The processing of eager protocol packets is much simpler when the network guarantees in-order delivery. When packets arrive out of order, software on the receive side spends processor cycles finding the destination message buffer and the offset in that buffer based on information in the packet.

The rendezvous protocol: optimizing processor usage and bandwidth The eager protocol is able to maximize single link bandwidth, but the per-packet processor overhead is too large to support the full bandwidth of the network. Reading a packet from the network requires 204 CPU cycles. Sending a packet takes between 50 and 100 cycles. When the network is at maximum capacity, data can flow at the rate of 3 Bytes/cycle on every node. At 270 raw bytes per packet, a processor has 90 cycles to handle each packet. Clearly, that is not possible with a single processor, and only marginally possible with two.

The goal of the rendezvous protocol is to minimize the amount of processing each packet needs. For example, packets that carry the destination address with it have a lower processing overhead at the reception side. This requires an initial dialog between the sender and the receiver to establish the destination address. For best performance we need to use the MPI rendezvous protocol.

The initial handshake of the rendezvous protocol costs 1500 cycles of processor time each on the sender and the receiver. This initial cost is amortized at the rate of about 150 cycles for every packet, making the rendezvous protocol viable beyond message lengths of 20 packets, or (at the rate of 240 bytes of payload per packet) about 5 kbytes.

Self-contained packets are also more suitable to be handled by the co-processor. The packets carry their destination addresses and processing them is a local operation suitable for the non-cache-coherent coprocessor. The main processor is notified at the end of the message through a small data structure in non-cached memory.

The software coherency protocol necessary for the hand-over of received data from the coprocessor to the main processor costs about 5000 CPU cycles. This makes coprocessor mode the preferred solution for message lengths of 10 kbytes or more.

Another advantage of self-contained packets is that they are insensitive to the arrival order. Thus, the bulk of rendezvous messages can be transmitted with adaptively routed packets, allowing for better network utilization.

A summary of protocols Table 1 shows a summary of the messaging protocols deployed today in the BlueGene/L MPI library. Each protocol is categorized by routing behavior, nearest neighbor bandwidth, latency, and coprocessor support. The short and eager protocols do not have coprocessor support today.

Protocol	Routing	Nearest Neighbor Bandwidth	Latency	Coprocessor	Range
Short	Deterministic	Low	Very good	No	0-240Bytes
Eager	Deterministic	High	Good	No	0.2-10KBytes
Rendezvous	Adaptive	Very high	Poor	Yes	3KB - inf

Table 1. A summary of the BlueGene/L MPI protocols.

5 Experimental results

In this section we present preliminary performance results using the first BlueGene/L 512-node prototype, which became operational in October 2003. We first present microbenchmark results that analyze different aspects of our current MPI implementation. We then compare different message passing protocols. Next, we analyze BlueGene/L-specific implementations of common collectives. Finally, we present results for the NAS parallel benchmarks in both coprocessor and virtual node modes. None of the results presented here use link chips – we restrict our studies to three-dimensional meshes rather than torii.

5.1 Hardware capabilities

In this section we measure the basic performance characteristics of the MPI libraries: roundtrip latency and link bandwidth.

Roundtrip latency analysis: We measured short message latency between two neighbor nodes on BlueGene/L using Dave Turner's `mpipong` program [12]. Current $\frac{1}{2}$-roundtrip latency stands at approximately 3000 cycles (6 microseconds at 500 MHz CPU frequency), consisting of multiple components shown in Figure 2 (a).

The overhead of the MPI library, 800 cycles (26% of total), is incurred by the MPICH2 code. This overhead can be ameliorated by deploying higher compiler optimization levels and/or using better compilers.

The ADI3 glue layer (`bgltorus`) and the Message Layer together contribute about 400 cycles (13%) of overhead, mostly testing the data types involved in the message, translating from communicator coordinates to message layer coordinates, and creating the Message Layer message that holds the data. At the receiving end, the overhead is spent in matching the incoming message against the queue of posted MPI requests.

Sending and receiving the packet itself is expensive for single-packet messages, because the packetizing techniques we use to avoid copying data typically do not apply to the first packet. Packet copies cost almost 300 cycles. Thus, the lowest layer of software spends 1000 cycles (29%) sending a packet and receiving the response.

The final component of MPI latency is the latency of actual hardware, currently at 32%. We estimate that this time could be reduced to 5-6% by employing shorter packets (currently all packets are 256 byte long), resulting in 20-25% overall savings in latency. We expect today's 3000-cycle latency numbers to improve as the MPI implementation matures.

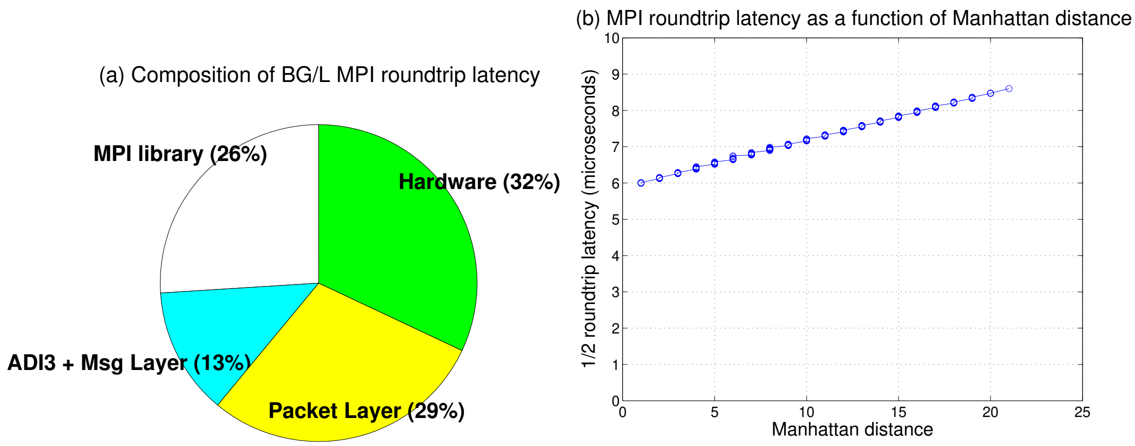

Fig. 2. Latency measurements in BlueGene/L.

Latency as a function of Manhattan distance: Figure 2 (b) shows $\frac{1}{2}$-roundtrip latency as a function of the Manhattan distance between the sender and the receiver in the torus. The figure shows a clear linear dependency, with 120 ns of additional latency added for every hop. We expect the per-hop latency to diminish as CPU frequency increases.

Single-link bandwidth: Figure 3(a) shows the available bandwidth measured with MPI on a single bidirectional link of the machine (both sending and receiving). The figure shows both the raw bandwidth limit (250 MBytes/s) and the payload bandwidth limit (220 MBytes/s) in addition to the measured bandwidth as a function of message size. With the relatively low message processing overhead of the MPI eager protocol, high bandwidth is reached even for relatively short messages: $\frac{1}{2}$ bandwidth is reached for messages about 1 kbyte long.

5.2 A comparison of point-to-point messaging protocols

Figure 3 (b), (c) and (d) compare the multi-link performance of the eager and rendezvous protocols, the latter with and without the help of the coprocessor. The eager and rendezvous protocols, without the coprocessor's help, are able to handle about two bidirectional links. When network traffic increases past two links, the processor becomes a bottleneck, as shown by Figures Figure 3 (b) and Figure 3 (c).

While multiple strategies may exist for harnessing the coprocessor – handling sends as well as receives, both for the eager protocol and the rendezvous protocol – currently only reception of rendezvous messages is implemented in BlueGene/L MPI. Figure 3 (d) shows the effect of the coprocessor helping out in the rendezvous protocol: MPI is able to handle the traffic of up to three bidirectional links.

5.3 Optimized Torus Broadcast

One of the challenges of implementing an efficient MPI library on the BlueGene/L machine is to find the efficient algorithms for the MPI collectives that are well suited to the machine's network. The MPICH2 implementation of the `MPI_Bcast` primitive has a limited efficiency on the BlueGene/L machine because it is designed for a machine with a crossbar type network.

The BlueGene/L MPI implementation includes an optimized broadcast algorithm based on MPI point-to-point communication that can be used with any communicator that maps onto a regular mesh in the physical torus.

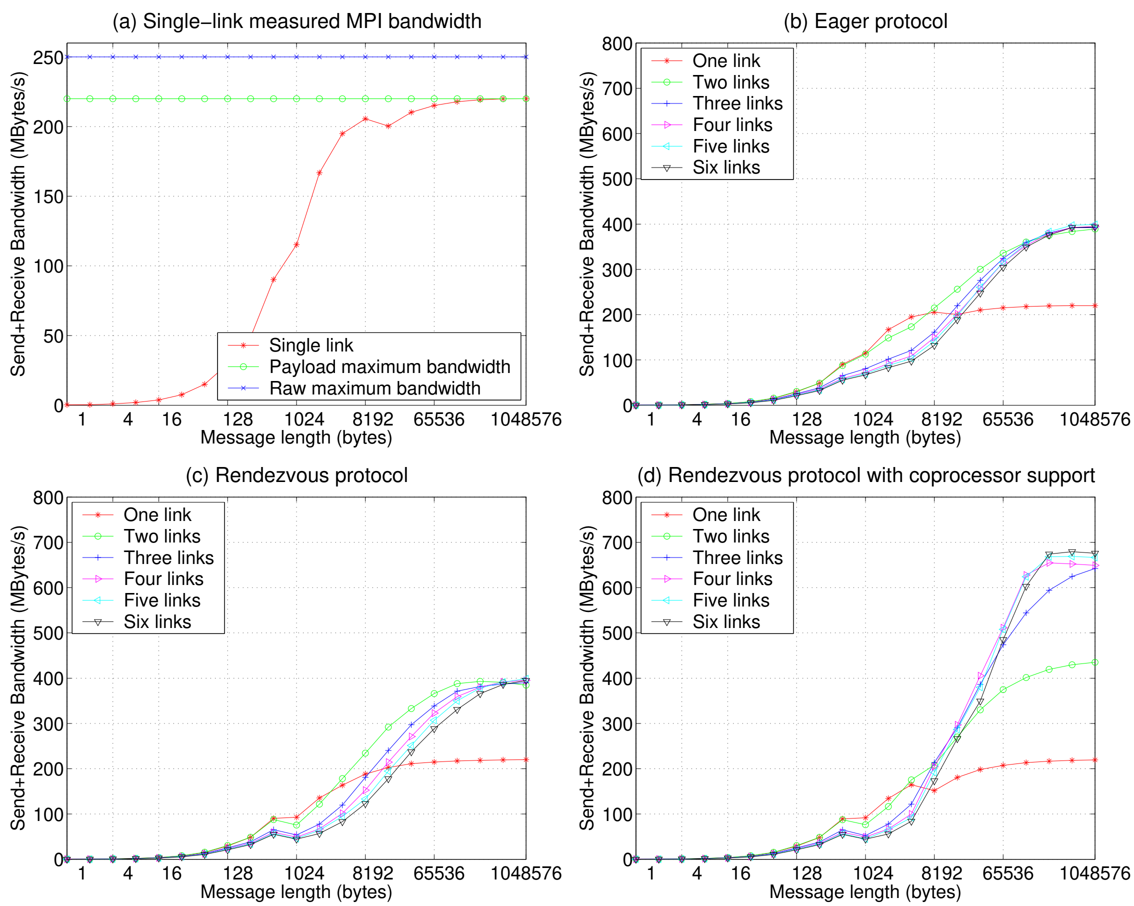

Fig. 3. Comparing multi-link bandwidth performance of MPI protocols.

For an n-dimensional mesh or torus, the algorithm consists of n concurrently executing stages (illustrated in Figure 4 (a) for the two-dimensional mesh case). The basic operation of the algorithm is a broadcast of a part of the message along a one-dimensional line in the n-dimensional topology. On an n-dimensional mesh the algorithm has the property that each process receives $\frac{1}{n}$ of the complete message from each of the x, y and z directions. On a torus topology each process receives $\frac{1}{2n}$ of the full message from each of the incoming links. Each block of the message is further subdivided to pipeline the broadcast process. The optimal subdivision size is a function of total message length, communicator topology and Manhattan distance in the network.

The BlueGene/L algorithm has proved superior to the standard MPICH2 broadcast algorithms, because those are oblivious to underlying network topology. The current implementation (Figure 4) is limited by the CPU processing capability of the node processors. For a mesh mapped communicator of size 8x4x4 an overall performance of 140MB/s is seen in single processor mode and 170MB/s in co-processor mode.

5.4 Tree bandwidth

As mentioned in Section 3, the tree supports collective operations, including broadcast and reduction operations. The MPI library currently uses the tree network to implement broadcast and integer reduce and allreduce operations on the `MPI_COMM_WORLD` communicator. Tree-based reduction of floating-point numbers is under development.

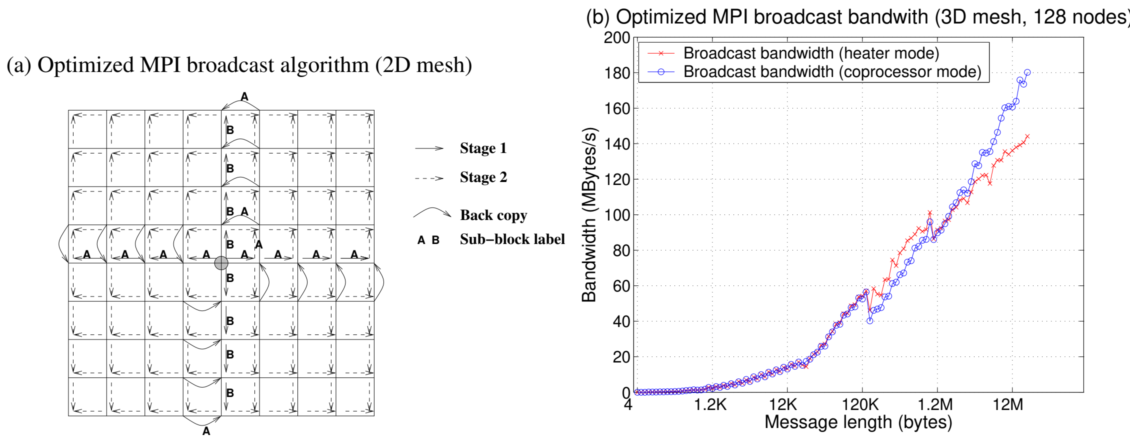

Fig. 4. Optimized mesh/torus broadcast algorithm.

Figure 5 shows the measured bandwidth of tree-based MPI broadcast and allreduce measured on the 512-node prototype. Broadcast bandwidth is essentially independent of message size, and hits the theoretical maximum of $0.96 \times 250 = 240$ Mbytes/s. Allreduce bandwidth is somewhat lower, encumbered by the software overhead of re-broadcasting the result.

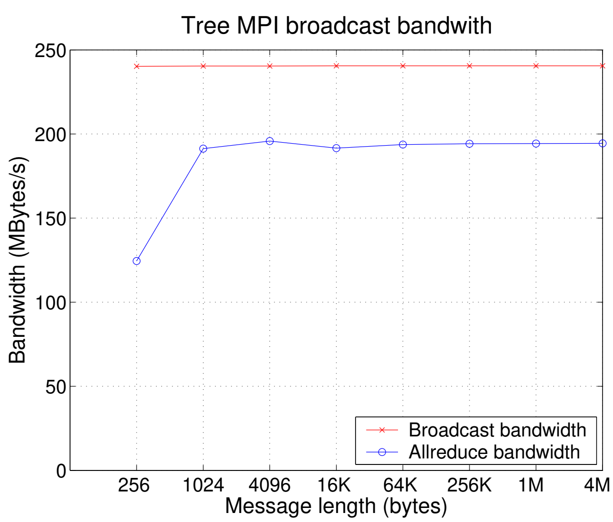

Fig. 5. Tree-based MPI broadcast and allreduce bandwidth.

5.5 Coprocessor mode vs. virtual node mode

Figure 6 shows a comparison of per-task performance in coprocessor and virtual node modes. We ran a subset of the class B NAS parallel benchmarks [7] on a 32-compute node subsystem of the 512-node prototype. We used 25 (for bt and sp) or 32 (for the other benchmarks) MPI tasks in coprocessor mode, and 64 (for all benchmarks) MPI tasks in virtual node mode.

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 the sharing node resources – including the L3 cache, memory bandwidth, and communication networks – individual processor efficiency degrades between 2-20%, resulting in less than ideal performance results. Nevertheless, the improvement warrants the use of virtual node mode for these classes of computation-intensive codes.

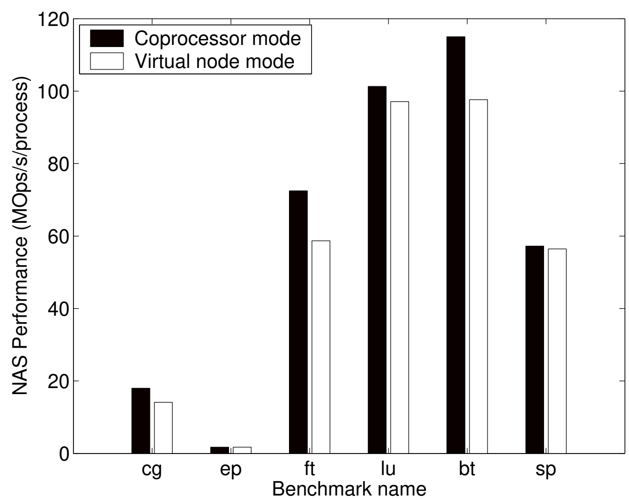

Fig. 6. Comparison of per-node performance in coprocessor and virtual node mode.

6 Conclusions

With its 65,536 compute nodes, the BlueGene/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 MPI to support application programmers effectively. The BlueGene/L architecture provides a variety of features that can be exploited in an MPI implementation, including the torus and tree networks and the two processors in a compute node.

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

BlueGene/L MPI has been deployed on our 512-node prototype, a small system compared to the complete BlueGene/L supercomputer, but powerful enough to rank at position 73 in the Top500 supercomputer list of November 2003. The prototype, with our MPI library, is already being used by various application programmers at IBM and LLNL. The lessons learned on this prototype will guide us as we move to larger and larger configurations.

References

1. The MPICH and MPICH2 homepage. `http://www-unix.mcs.anl.gov/mpi/mpich`.

2. Process Management in MPICH2. Personal communication from William Gropp.
3. N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.
4. G. Almási, C. Archer, J. G. C. nos, 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. Lecture Notes in Computer Science. Springer-Verlag, September 2003.
5. G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. C. nos, 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. In *Proceedings of Euro-Par 2003 Conference*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.
6. G. Almasi et al. Cellular supercomputing with system-on-a-chip. In *IEEE International Solid-state Circuits Conference ISSCC*, 2001.
7. 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, December 1995.
8. G. Chiola and G. Ciaccio. Gamma: a low cost network of workstations based on active messages. In *Proc. Euromicro PDP'97, London, UK, January 1997, IEEE Computer Society.*, 1997.
9. W. Gropp, E. Lusk, D. Ashton, R. Ross, R. Thakur, and B. Toonen. MPICH Abstract Device Interface Version 3.4 Reference Manual: Draft of May 20, 2003. `http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf`.
10. S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95, San Diego, CA, December 1995*, 1995.
11. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference, second edition*. The MIT Press, 2000.
12. D. Turner, A. Oline, X. Chen, and T. Benjegerdes. Integrating new capabilities into NetPIPE. Lecture Notes in Computer Science. Springer-Verlag, September 2003.
13. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 1995.
14. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.