Optimization of MPI Collective Communication on BlueGene/L Systems

George Almási
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
gheorghe@us.ibm.com

Charles J. Archer
IBM Systems and Technology Group
Rochester, MN 55901
archerc@us.ibm.com

C. Chris Erway
Dept. of Comp. Sci.
Brown University
Providence, RI 02912
cce@cs.brown.edu

Philip Heidelberger
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
philip@us.ibm.com

Xavier Martorell
Dept. of Comp. Arch.
Universitad Politechnica de Catalunia
08071 Barcelona (SPAIN)
xavim@ac.upc.es

José E. Moreira
IBM Systems and Technology Group
Rochester, MN 55901
jmoreira@us.ibm.com

B. Steinmacher-Burow
IBM Germany
Boeblingen 71032
(GERMANY)
steinmac@de.ibm.com

Yili Zheng
School of Elec. & Comp. Engr.
Purdue University
West Lafayette, IN 47907
yzheng@purdue.edu

ABSTRACT

BlueGene/L is currently the world's fastest supercomputer. It consists of a large number of low power dual-processor compute nodes interconnected by high speed torus and collective networks. Because compute nodes do not have shared memory, MPI is the the natural programming model for this machine. The BlueGene/L MPI library is a port of MPICH2.

In this paper we discuss the implementation of MPI collectives on BlueGene/L. The MPICH2 implementation of MPI collectives is based on point-to-point communication primitives. This turns out to be suboptimal for a number of reasons. Machine-optimized MPI collectives are necessary to harness the performance of BlueGene/L. We discuss these optimized MPI collectives, describing the algorithms and presenting performance results measured with targeted micro-benchmarks on real BlueGene/L hardware with up to 4096 compute nodes.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.1.3 [**Concurrent Programming**]: Parallel Programming

ICS'05, June 20-22, 2005, Boston, MA, USA

General Terms

Algorithms, Design, Measurement, Performance

Keywords

BlueGene, Collective Communication, MPI, Optimization, Performance

1. INTRODUCTION

BlueGene/L is a new massively parallel computer architecture developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). BlueGene/L systems use system-on-a-chip integration [6] and a highly scalable architecture [2] to assemble an army of low power dual-processor nodes with high speed interconnects. When operating at the target frequency of 700 MHz, LLNL's flagship 64K-node BlueGene/L system will deliver up to 360 Teraflops of peak computing power.

Each BlueGene/L compute node can address only its local memory, making message passing the natural programming model for the system. The BlueGene/L MPI implementation is an optimized port of Argonne National Laboratory's MPICH2 library [1]. The challenges of implementing high performance point-to-point communication in the MPI library has been described in our previous work [3, 4].

In this paper we describe improvements to BlueGene/L MPI collective communication. The speed of MPI collectives is, needless to say, often the critical factor determining the ultimate performance of parallel scientific applications. It is typical of MPI implementations (such as MPICH2) to implement collective communication in terms of point-to-point messages. However, the MPICH2 collective implementations [16] suffer from low performance on BlueGene/L sys-

tems. Our initial analysis concluded that there are at least three reasons for this:

- **Network topology awareness.** The MPICH2 collectives are written without specific network hardware in mind; they tend to perform well on crossbar-type networks. However, the most important BlueGene/L network hardware is a 3D torus, and the MPICH2 collective algorithms tend to map poorly onto this network, ending up using the limited cross-section bandwidth of the torus network very inefficiently and creating network hot spots that spoil performance. The collectives tend to scale poorly with network size.
- **Special purpose network hardware.** BlueGene/L features special hardware designed to speed up certain collective operations. One such feature is the *deposit bit* which lets torus packets deposit a copy on every node they touch on the way to their final destination. There is also special purpose hardware just to speed up reductions and barrier operations. The default MPICH2 collective algorithms do not take advantage of these features.
- **Other hardware properties.** Like any computer, the design process of BlueGene/L has resulted in a number of architectural compromises that require software to deal with. These compromises affect performance to a large degree and caused many surprises during the implementation process. As examples we can cite the very high cost of memory copies on the machine and the lack of cache coherence between processors in a node.

We implemented of a number of optimized collective operations. Because MPICH2 was designed from the ground up to be extensible, we were able to add them to MPICH2 as plug-in modules. We did not try to provide better general purpose algorithms; instead we concentrated on the cases when optimization was possible and wrote special purpose algorithms that only get triggered when the conditions are right. For example, our MPI broadcast implementation is only triggered when the communicator it is invoked on is a 1, 2 or 3-dimensional rectangle on the physical network. In our first round of algorithm design we tackled only the most commonly used collectives such as `MPI_Alltoall`, `MPI_Allreduce`, `MPI_Barrier` and `MPI_Bcast`. We provide performance and scaling data for each of our algorithms.

The rest of this paper is organized as follows. Section 2 provides an overview of the BlueGene/L system and the optimized implementation of MPI collectives. Section 3 presents the algorithms and implementation of MPI collectives for torus network. Section 4 describes the collective operations for BlueGene/L collective and global interrupt networks. Section 5 compares the performance and scaling of our implementations. We conclude in section 6.

2. OVERVIEW

2.1 Brief overview of BlueGene/L systems

The BlueGene/L hardware and system software have been extensively described in other publications [2, 5]. Here we present a short summary of the BlueGene/L architecture to serve as the background to the following sections.

Processing core: Each BlueGene/L node ASIC include two standard PowerPC 440 processing cores. The standard PowerPC 440 processors are not designed to support multiprocessor architectures. Hence the L1 caches are not cache coherent. To overcome this limitation, 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. The L2 and L3 caches are coherent between the two processors.

Communication networks: 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 with 154 MBytes/s payload bandwidth in each direction. The 64 racks in the full BlueGene/L system form a 64 32 32 three-dimensional torus. The network hardware guarantees reliable, deadlock free delivery of variable length packets. Torus packets are routed on an individual basis, using either the *deterministic* routing algorithm or the *adaptive* routing algorithm. Deterministic routing assures in-order packet arrival, whereas adaptive routing permits better link utilization.

The *collective* network is a configurable network for high performance broadcast and reduction operations, with a latency of 2.5 μs for a 65,536-node system. It has reliability guarantee identical to the torus network and provides point-to-point capabilities as well. The collective network packet length is fixed at 256 bytes, all of which can be used for payload. The payload bandwidth of the collective network is about 337 MBytes/s. The *global interrupt* (GI) network provides configurable OR wires to perform full-system hardware barriers in 1.5 μs.

Operating modes: To deal with the non-coherence of the processors in a node, software allows multiple modes of operation. The simplest of these is *heater mode*, in which one of the two processors is in an idle loop and does no useful computation. In *coprocessor mode* one of the processors runs the main thread of the user's program, while the other processor helps out with communication and/or computation tasks. In this case cache coherence has to be managed by software. In *virtual node mode* the two processors of a compute node act as different processes: each has its own MPI rank, and all hardware resources are equally shared.

2.2 Software architecture of BlueGene/L MPI

We built BlueGene/L MPI by porting MPICH2 [1], an MPI library designed with scalability and portability in mind. MPICH2 provides the implementation of point-to-point messages, intrinsic and user defined datatypes, communicators, and collective operations, and interfaces with the lower layers of the implementation through the Abstract Device Interface version 3 (ADI3) layer [9].

The ADI Layer is described in terms of MPI requests (messages) and functions to send, receive, and manipulate these requests. 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.

The BlueGene/L Message Layer is an active message system [8, 12, 18, 19] that implements the transport of arbitrary-sized messages between compute nodes using the torus network. It consists of four main components: basic functional support, point-to-point communication primi-

tives (or protocols), collective communication primitives and development utilities. The basic functional component acts as a support infrastructure for the implementation of all the communication protocols. 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 packets are re-assembled into a message.

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/collective network hardware, as well as to poll the state of the network.

2.3 Software design decisions

The performance of MPI collectives tends to be highly dependent on the circumstances of their invocation. This is especially true for BlueGene/L because of the peculiarities of the network hardware. Our mission statement was to enable high performance for the subset of invocation scenarios where hardware or software can help. For all other scenarios we allow MPICH2 default collectives to take over.

Plug-ins: We added a testing phase to every communicator creation and every collective invocation in MPICH2 (as mentioned before, the library is designed to allow this). During communicator creation we test for global properties of the communicator. The two most interesting tests are (a) whether the communicator is `MPI_COMM_WORLD` and (b) whether the communicator has a contiguous rectangular shape on the torus network.

During invocation, we eliminate complex situations involving non-contiguous buffers and intercommunicators (we allow the MPICH2 default implementations in these cases). Furthermore we discriminate based on message size, since for certain collectives we have multiple algorithms optimizing latency (for short messages) or bandwidth (for long ones).

Global algorithm decisions: The selection of the actual algorithm to perform a collective operation is done when the collective has been invoked. This can lead to undesirable situations if the decision is made locally, because MPI programming errors (such as invoking `MPI_Bcast` with different size arguments across the participating nodes), and even certain legitimate MPI calls, can lead to individual nodes choosing *different* algorithms to implement the same operation. This usually results in deadlocks.

The only way to insure correct behavior in such cases is to take the algorithm decision globally across the communicator (by invoking another collective). This leads to an increase in latency, and therefore we tend to do this only when we believe that the resulting gains in bandwidth are more important.

Unexpected messages: Another decision we made was not to deal with *unexpected* (or early) collective packets, i.e. packets that arrive to a node before that node has entered the collective implementation. Unexpected messages are normally dealt with by the point-to-point messaging subsystem. On BlueGene/L this is an expensive proposition because memory copies cause increased CPU loads and therefore performance loss. To keep our optimized algorithms simple and efficient we do not allocate memory buffers for early packets. We prevent early packets by prefixing collectives with barrier calls, taking advantage of BlueGene/L's dedicated barrier network where we can.

Non-blocking collectives: All collective primitives described in this paper are non-blocking, relying on termination callbacks to announce their completion. This was done to allow computation/communication overlap and to let the program service all networks simultaneously. It also allows us to use the collectives for purposes other than MPI if we have to.

Preconditions: In the remainder of this paper we will discuss a number of collective algorithms. For each of these we will specify the pre-conditions that must exist for the algorithm to be invoked, why we believe that the algorithm is better than the default, and of course we will document everything with numbers.

3. TORUS COLLECTIVES

In this section we deal with algorithms written for rectangular sections of the torus. We first need to clarify what a rectangular section is. We will denote a booted BlueGene/L partition is a collection of nodes

$$\Lambda = \{(x, y, z) \in \mathbf{Z}^3 |\ 0 \quad x < x_s \wedge 0 \quad y < y_s \wedge 0 \quad z < z_s\}$$

An MPI communicator is any set of nodes $\Gamma \subset \Lambda$. The communicator Γ is rectangular if and only if

$$\forall (x, y, z) \in \Gamma \quad (x_0, y_0, z_0) \quad (x, y, z) < (x_1, y_1, z_1) \wedge \|\Gamma\| = (x_1 \quad x_0 + 1) \quad (y_1 \quad y_0 + 1) \quad (z_1 \quad z_0 + 1)$$

A booted BlueGene/L partition is always rectangular, but `MPI_COMM_WORLD` need not be rectangular because an MPI job start on a subset of the booted partition. Rectangular communicators are important to us because they are regular and easy to reason about, but also because of the *deposit bit* capability of the torus: packets sent along a line and deposited on every node they touch. Next we will describe a number of algorithms that map into rectangular regions.

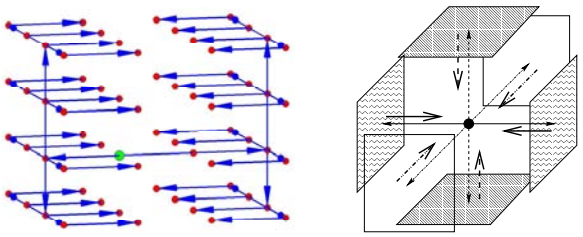
3.1 Long-message collectives

MPI_Bcast: The MPICH2 broadcast implementation uses two algorithms. For short messages it uses a binary tree to minimize processor load and latency; for long messages it performs a binary tree scatter followed by an allgather. On BlueGene/L the measured performance of these algorithms is very low, mostly for lack of topology awareness and high CPU overhead.

The BlueGene/L-optimized algorithm is suitable for long MPI broadcasts executed on rectangular subsets, meshes and tori. The implementation of MPI broadcast follows the general pattern proposed by Watts and van de Geijn [20]. The basic idea is to find a number of non-overlapping spanning trees in the rectangular mesh/torus. The broadcast message is split into components, and each component is pipelined separately (and simultaneously) along one of the spanning trees. Thus the theoretical achievable bandwidth of this algorithm is a multiple of single link bandwidth.

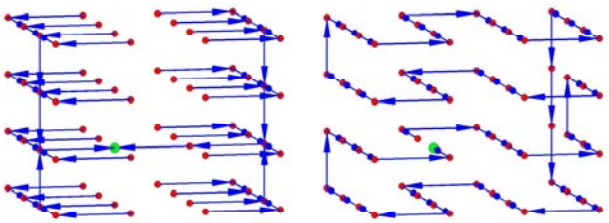
The multiplier cannot be more than the number of incoming links on any of the nodes in the communicator (each node has to get all pieces of the message). Thus, for a mesh the multiplier is equal to the dimension of the rectangular region (1, 2 or 3); if the rectangular region is wrapped back (toroidal), the multiplier doubles. The theoretical maximum bandwidth for a 3D torus is therefore 2 3 = 6 times the bandwidth of a single link.

Figure 1(a) shows the structure of one data stream in a 3-dimensional mesh. The other data streams are essentially rotated versions of the one depicted; Figure 1(b) shows how



(a) Single data stream (b) 3 streams

Figure 1: Optimized broadcast algorithm on a 3D mesh



(a) Reverse broadcast (b) Hamiltonian path

Figure 2: Optimized `Reduce` algorithm on a 3D mesh

three streams can be used at the same time without using any of the mesh links twice.

The algorithm attempts to exploit long straight lines in the data streams. Packets traveling along these lines can have their deposit bit turned on. The processors receiving these packets don't have to re-send them, thereby lowering CPU overhead and improving latency. The only nodes that have to re-inject packets onto the network are those that have to "turn" the message by a 90 degree angle. To further improve latency, packet re-injection is pipelined: each incoming packet is immediately sent forward along its data stream.

The performance of the broadcast algorithm is unfortunately not only limited by the network's limitations, but by the CPU load on the individual nodes. The busiest processors on the network determine performance. Unsurprisingly, these turn out to be the very nodes that have to re-inject packets on the network. These nodes limit practical performance to no more than two network links worth. A better algorithm, which uses both processors to ease CPU load, is in the works.

MPI_Reduce: `Reduce` can be viewed as broadcast in reverse; that is, the same stream used for a broadcast may be reversed and used for reduction to the same root, as in Figure 2(a). However, this overlooks an important difference. Here, each node must apply the specified reduce operation to combine its own data with each incoming packet, presenting a significant performance bottleneck (especially for CPU-intensive operations such as floating-point sums). Additionally, since each node's contribution changes the data it passes on, this prevents the use of the deposit bit.

Like broadcast, the busiest processors on the network determine long-message performance. For `Reduce`, the busiest are those that have to combine data from multiple neighbors (that is, those with indegree > 1 in the directed graph representing the stream). For these nodes, two or three incoming packets must be individually received and reduced before the resultant data may be sent along to the next node, slowing the overall operation.

Thus, to minimize the number of incoming data sources (indegree) per node, a Hamiltonian path chaining together all nodes in the communicator ensures each node receives data from only one neighbor. Figure 2(b) depicts such a path; results are collected along the stream in the direction of the root. As a result, the same per-packet overhead is experienced at every node in the stream, boosting bandwidth at a severe cost to latency.

For this reason, the Hamiltonian paths are preferred only for long vectors. Much work has been done to minimize the per-packet overhead of `Reduce`; for example, when possible, the PowerPC floating-point units, which retrieve packets from the network hardware, apply the reduce operation (e.g. sum, max) before saving data to memory.

Computation of the routes for a stream are performed locally. For rectangular meshes where one of the dimensions is even, a Hamiltonian path is constructed, producing two streams (using opposing directions along the same path) that may be be used simultaneously.

MPI_Allreduce: The long-message `Allreduce` on torus is essentially a pipeline connection of long-message `Reduce` and long-message broadcast primitives. Two non-overlapping streams are needed, the first leading to the root (for `Reduce`) and the second starting from the root (for broadcast). The second stream may be, most simply, the reverse of the first stream; recall that torus links are bi-directional, so the two streams do not interfere.

As in `Reduce`, Hamiltonian paths are desirable for long-message `Allreduce`. Reduction takes place along a single Hamiltonian path ending at the root, as in Figure 2(b); that stream is used in reverse to broadcast the results.

For long-message `Allreduce`, MPICH2 uses Rabenseifner's algorithm [14]. This algorithm implements `Allreduce` in two steps: first a `Reduce-Scatter`, followed by an `Allgather`. Pipelining packets and utilizing architectural features help our `Allreduce` achieve better bandwidth than the MPICH2 implementation for long messages.

MPI_Alltoall and MPI_Alltoallv: When designing BlueGene/L MPI, we did not anticipate having to provide an improved version of `Alltoall`. The operation is essentially limited by the cross-section bandwidth of the torus network. MPICH2 `Alltoall` is implemented by no less than four different algorithms. On BlueGene/L these algorithms suffer from a multiplicity of problems, such as high CPU overhead, creation of hot spots on the network and poor use of the compute nodes' memory subsystems.

The BlueGene/L-optimized `Alltoall`/`Alltoallv` algorithm works well on all communicators and all message sizes. Since `Alltoallv` subsumes `Alltoall`, the implementation of both is provided by a single function in the message layer. The algorithm keeps CPU overhead low by not using point-to-point messaging, and avoids network hotspots by randomizing torus packet injection. This randomization is done through a permuted list of destinations (MPI ranks). The

algorithm scans through the permuted list and picks a destination to send the next packet to. The permutation list is the same on each node, because the random number generator is seeded with the same number everywhere. However, every node starts from a different offset in the permutation list.

The permutation list is lazy-allocated and initialized from the group and communicator connection table when `MPI_Alltoall/MPI_Alltoallv` is first called on a particular communicator. The ranks list uses a single unsigned integer for each rank in the system. Thus, the maximum memory used in a virtual node mode, 64K node system is 2 655356 nodes 4 bytes = 512 KBytes. Lazy allocation potentially reduces memory requirements for MPI by not reserving memory for the permuted list unless it is needed.

`Alltoallv` is inherently unbalanced because there may be more data to send to some ranks than others. As the algorithm moves forward, the rank permutation list will contain more and more destinations that the sender has no more data to send to. To avoid excessive CPU overhead caused by scanning empty slots in the permutation list, nodes that have no more data to receive are removed from the permuted rank array by rearranging the array in place.

Randomization of the send destinations implies randomization of both packet receives and packet sends, which can be up to 240 bytes (8 cache lines) of payload. Rapid switching between destination strains the local memory subsystem. In order to make the best possible use of intelligent prefetching in the cache architecture, the algorithm injects multiple packets (from adjacent cache lines) to each destination before advancing to the next entry in the permutation list. This leads to a compromise where more packets per destination will ease the load on the memory subsystem, but potentially create more hotspots on the torus network. Empirically we found two packets per destination to lead to the best performance.

The `Alltoallv` algorithm is also small message aware, and will adjust the torus packet size to minimize latency when the amount of data exchanged between pairs of MPI ranks is less than a full packet worth.

3.2 Short-message collectives

The torus `Allreduce` and `Broadcast` algorithms described so far are designed for throughput. They sacrifice latency for better pipelining of concurrent data streams, and are really unsuitable for short messages. At the same time we expect the unprecedented number of processors in BlueGene/L to cause messages to become shorter, especially in strong scaling applications.

In this section we present an optimized `Allreduce/Barrier` algorithm designed for very short (one packet) short messages on rectangular communicators. The basic insight of the short message optimized algorithm is that *we can trade bandwidth against latency*: instead of the classic store-and-forward implementation of a reduction operation we broadcast all data to all nodes and replicate the necessary processing on all nodes. On BlueGene/L this is advantageous because it is possible to broadcast a packet to a line of nodes without store-and-forward using the deposit bit feature of the network.

However, since every node on the line is broadcasting its packet to everyone else, the line becomes full very quickly. The latency of this algorithm is determined not by the network latency, but either by the network's bandwidth or by the processing capability of the nodes.

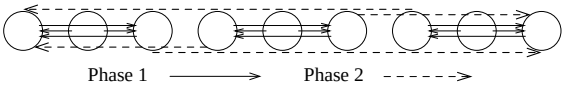


Figure 3: Two-phase hierarchical `Allreduce` on a single line

Using this algorithm `Barrier` can be implemented as a very simple `Allreduce` in which each node waits until it receives all incoming packets. Processing time is very low in this case, and therefore `Barrier` latency is determined by the network bandwidth. For floating point `Allreduce` the CPU processing time tends to be higher, so the algorithm will be limited by CPU overhead. In general the algorithm's latency can be expressed as

$L = S + (n \quad 1) \quad max(\frac{\|P\|}{BW}, T_{pkt})$,

where S is a constant overhead (CPU time spent in messaging library and network link latency), n is the number of nodes on the line, $\frac{\|P\|}{BW}$ is the time necessary for a packet to traverse a link (packet size divided by link bandwidth) and T_{pkt} is the CPU processing time for a received packet. The formula can be derived by looking at the nodes at one end of the line: it receives and processes $n \quad 1$ packets over a single link; hence the linear dependence on n.

Linear dependence on n is obviously not good for scaling. However, we can mitigate bandwidth with store-and-forward latency by employing a two-phase hierarchical algorithm as depicted in Figure 3. This algorithm uses broadcast in the subgroups; the nodes at the ends of the subgroups then become representative for the group and broadcast results to the other nodes. In the two-phase algorithm depicted in the figure latency can be calculated as follows:

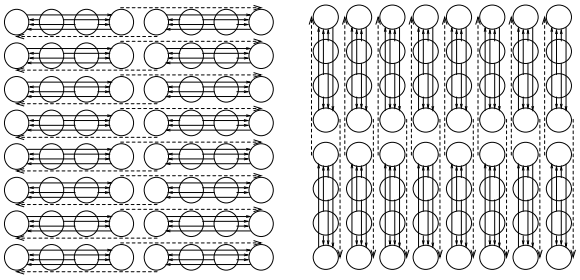
$L = 2 \quad S + (m \quad 1 + n \quad 1) \quad max(\frac{\|P\|}{BW}, T_{pkt})$,

where m is the size of a node group and n is the number of groups. Note that the static overhead S is incurred twice because partial results have to be re-injected into the network.

The algorithm can be trivially expanded to rectangular meshes of arbitrary dimensions by executing multiple rounds, one for each new dimension. In the first round each processor performs the algorithm along the first dimension. In the second round the partial results are combined along the second dimension; and so on. Figure 3.2 illustrates the algorithm in an 8 8 2D mesh with horizontal and vertical phases. Since virtual node mode can be thought of as operating in a 4-dimensional mesh, the algorithm translates to virtual node mode without modification.

The optimal number of subgroups and group size in the hierarchical algorithm are determined by both the size of the message and by the dimensions of the communicator the algorithm is performed in. Our current implementation limits the algorithm to messages no longer than a single torus packet (we intend to address this issue in the future). We have found through measurement that a group size of 4 is suitable for `Allreduce` and a group size equal to the dimension is good for `Barrier`.

Algorithms for global reduction and barrier synchronization have been extensively studied in [15, 7, 13, 10]. MPICH2 uses a recursive doubling algorithm [17] for short `Allreduce` type operations and the dissemination algorithm [11] for



(a) Horizontal phase (b) Vertical phase

Figure 4: Multi-phase hierarchical `Allreduce` on 2D mesh

`Barrier`. Both algorithms try to optimize the number of hops that the message data has to traverse. Our algorithm is superior because it optimizes "software hops" instead of "network hops" by exploiting special capabilities available only on the BlueGene/L torus network.

4. COLLECTIVE AND GLOBAL INTERRUPT NETWORK COLLECTIVES

In this section we describe the MPI collective communication implemented for the BlueGene/L *collective* and *global interrupt* networks.

`MPI_Barrier` on the global interrupt wires: we have implemented a non-blocking barrier for MPI using the global interrupt network. The global interrupt barrier works on 32, 128, 512, and multiples of 512 nodes.

`MPI_Allreduce` and `MPI_Bcast` on the *collective* network: the *collective* network routes packets upward to the root and/or downward to the leaves as desired. It comes with a fixed point arithmetic unit in every node. The operation performed by the arithmetic unit is determined by the type of the packet.

For instance, `MPI_Bcast` is implemented by idling the arithmetic unit. The logical root of the broadcast sends the message up to the physical root of the collective network, which then re-broadcasts the message to everyone else. Collective network hardware takes care of the proper routing. Pipelining at a packet level insures minimum latency and maximum bandwidth.

Fixed point versions of `MPI_Allreduce` operations, such as addition, maximum search or even `MAXLOC`, can be implemented by just feeding the collective network with packets with the correct operation type. The collective network hardware performs combine operations as the packet streams converge on the route from the leaves to the root. Packets reaching the root are turned back and re-broadcast to all leaves.

`MPI_Barrier` can be trivially implemented on the collective network by each node injecting a combine packet into the collective network and waiting for the response from the root. The contents of the packet and the performed operation do not matter.

The situation is somewhat more complex in the case of floating-point `Allreduce` operations. Because the collective network hardware can only perform fixed-point operations, our implementation of floating-point `Allreduce` must deal with the complexities of IEEE floating-point representation in software. This costs CPU cycles, resulting in lower bandwidth. The MPI collective operations for BlueGene/L collective network only work on `MPI_COMM_WORLD`.

Machine size	Torus topology	Machine size	Torus topology
32	4 4 2	64	8 4 2
128	8 4 4	256	8 8 4
512	8 8 8 (T)	1024	8 8 16 (T)
2048	8 16 16 (T)	4096	8 32 16 (T)

Table 1: Torus topologies of diff. machine sizes

5. PERFORMANCE

To measure the performance of our optimized MPI collectives we wrote a set of micro-benchmarks targeted towards testing performance on the network topologies that were important to us. Because the type and amount of collective communication vary significantly across the whole application spectrum, using real application benchmarks is less straightforward for quantifying the performance of individual collective operation. Therefore, we used micro-benchmarks instead of real application benchmarks in our performance analysis. Our micro-benchmarks measured the latency and bandwidth of each MPI collective operation individually. All benchmarks were run on a 4096-node BlueGene/L system installed at IBM's Rochester site. For scalability measurements we booted smaller partitions inside the large machine. Table 1 shows all the partition sizes and topologies we used. Note that partition sizes below 8 8 8 can only be booted in a mesh configuration. All other partition sizes, denoted by a "T" in the table, are torus configurations with wrap-around links on all three dimensions.

5.1 Bandwidth of MPI_Bcast

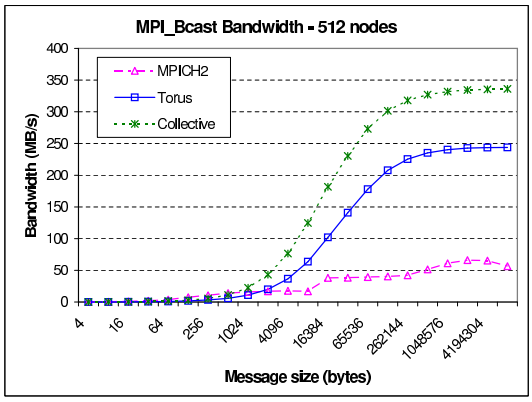
The torus broadcast implementation has a target bandwidth equivalent to 3 network links on a 3D mesh. In fact, its performance is bound by CPU overhead and memory bandwidth, and therefore is limited to less than 2 network links. The collective network offers performance close to the full bandwidth of the collective network, 337 MBytes/s, as shown in Figure 5(a).

One unforeseen aspect of optimizing `Bcast` was that the decision whether to apply an optimized algorithm has to be taken globally, across the whole communicator (in order to avoid situations in which some nodes in the communicator decide to use the optimized algorithms but others don't). The decision involves a round of short `Allreduce` that precedes the actual broadcast, driving latency up for all optimized operations. Figure 5(b) focuses on message lengths of less than 4 KBytes and shows the default MPICH2 implementation outperforming the optimized ones for message lengths of up to 2 KBytes. There is obvious room for improvement here.

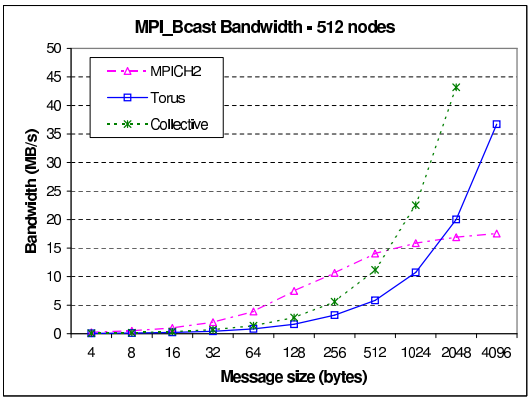
5.2 Bandwidth of MPI_Allreduce

We present a separate set of numbers for fixed point and floating point implementations of `MPI_Allreduce`.

Figure 6(a) compares the bandwidth of three implementations of `MPI_Allreduce` sum of integers. It is immediately

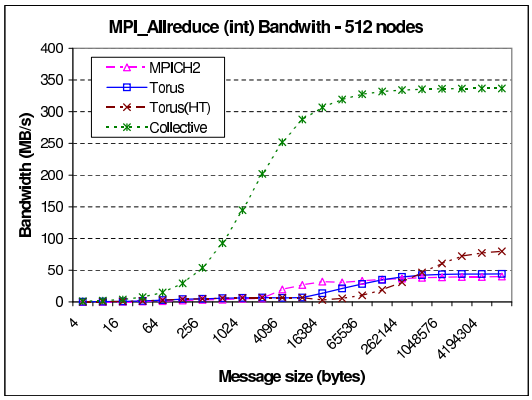


(a)

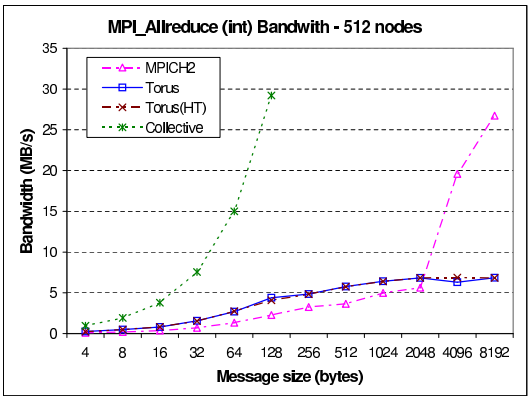


(b)

Figure 5: Bandwidth comparison of `MPI_Bcast`

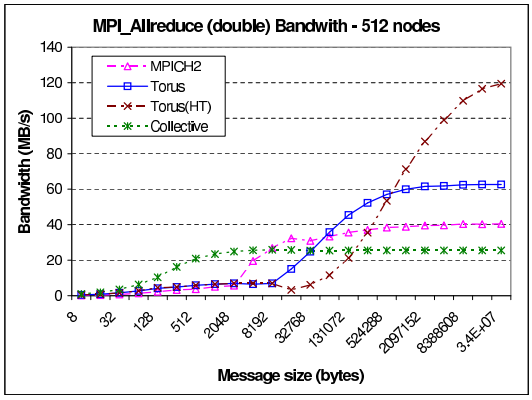


(a)

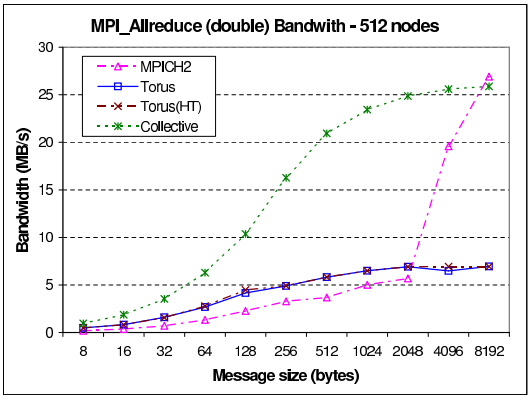


(b)

Figure 6: Bandwidth comparison of `MPI_Allreduce(int)`

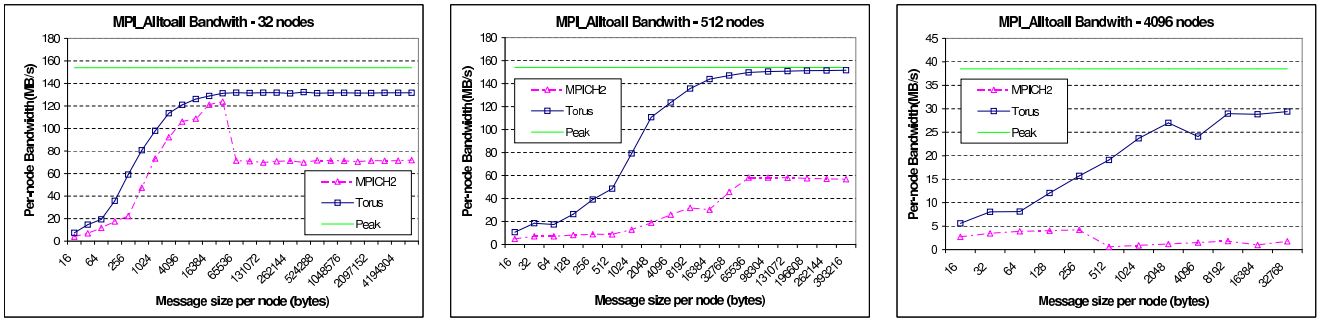


(a)



(b)

Figure 7: Bandwidth comparison of `MPI_Allreduce(double)`

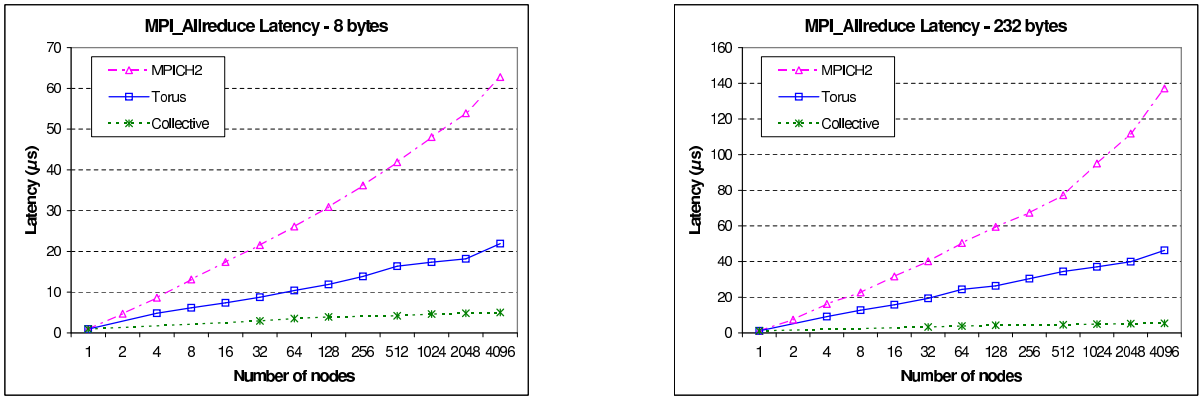


(a) (b) (c)

Figure 8: Per-node bandwidth comparison of `MPI_Alltoall`

Machine topology	Aggr. peak (MB/s)	Message size per node (Bytes)									Best % Peak
		16	64	256	1024	4096	16384	64KB	256KB	1MB	
4x4x2	4928	237	619	1892	3133	3871	4123	4220	4196	**4215**	86
8x4x2	4928	566	1272	2817	3908	4459	4566	4618	4596	**4603**	94
8x4x4	9856	1330	2705	5524	7740	8724	9003	9053	9076	**9087**	92
8x8x4	19712	2727	4354	9905	14458	16538	16802	16935	**16954**		86
8x8x8(T)	78848	5463	9462	20015	40548	63261	73689	76622	**77525**		98
8x16x8(T)	78848	8751	12936	26845	47945	58518	61557	**62536**			79
8x16x16(T)	157696	14155	21867	48745	103111	126053	130628	**135465**			86
8x32x16(T)	157696	22860	32910	64398	97069	98616	**118120**				75

Table 2: Aggregate bandwidth of torus `MPI_Alltoall`



(a) (b)

Figure 9: Latency comparison of `MPI_Allreduce` for short messages

evident that the collective network achieves a performance close to the theoretical maximum as in `MPI_Bcast`. The torus version of `Allreduce` with Hamiltonian reduction path, denoted as Torus(HT), is the next best for long messages, but suffers from high latency which makes it less advantageous for short messages. Figure 6(b) highlights the short-message portion of the performance comparison, in which the Torus and Torus(HT) cases used the short-message torus allreduce algorithm with simple multi-packet extension for message length up to 8 KBytes. It is clear that our optimization work here is not finished yet.

For `MPI_Allreduce` with double precision floating-point numbers, the collective network implementation has a much lower bandwidth. This is because the network does not provide operations on floating point numbers. We implemented a two-phase algorithm instead which parses the exponents first, calculates maximums, shifts mantissas into position, performs fixed-point allreduce on the mantissas and finally re-arranges the results into IEEE compliant double precision floating point representation. This requires a lot of CPU overhead, resulting in low performance.

In contrast, both torus versions (with and without Hamiltonian path) perform much better than in the integer case because of a design quirk in the network that allows network to floating point number transfers at a much higher rate.

These performance changes are obvious in figures 7(a) and 7(b). The MPICH2 implementation behaves similarly for integer and double numbers because it does not utilize the network tricks we employed for the optimized `Allreduce`.

5.3 Bandwidth of MPI_Alltoall

Personalized communication is bandwidth intensive. Performance is ultimately limited by the shape of the network. For a mesh of size $m \quad n \quad p$ the theoretical maximum network `Alltoall` bandwidth is $\frac{4L}{max(m,n,p)}$ (where L is the bandwidth of a single link). For a torus the formula is $\frac{8L}{max(m,n,p)}$.

The formula is based on cross-section bandwidth: in any dimension of the mesh half the nodes will want to communicate to the other half. In the first dimension this amounts to sending $\frac{m \times n \times p}{2}$ messages over the total available cross-section bandwidth of 2 $n \quad p \quad L$. Tori have twice the number of links, hence the cross-section bandwidth doubles.

Figures 8(a), 8(b) and 8(c) compare the per-node bandwidth of the default (MPICH2) and optimized `MPI_All2all` implementations for three machine sizes. The straight lines at the top denote theoretical peak bandwidth, taking into consideration all factors like packet payload. The graphs show up the deficiencies of the MPICH2 implementation on this network. MPICH2 switches between algorithms at 256 Bytes and again at 32 KBytes. Figure 8(c) shows the switch from a store-and-forward algorithm to an all-post-and-receive algorithm. Performance drops because of the MPI overhead involved in posting so many messages. Figures 8(a) and 8(b) show the second switch, from all-post to a pairwise send/receive algorithm. Performance then depends on how the order of sends and receives interacts with the physical network topology.

Table 2 shows how peak `Alltoall` bandwidth of the optimized algorithm scales with machine size. The best achieved bandwidth is highlighted. In all cases we achieve more than 75% of peak bandwidth, and the ratio seems better for machine topologies that are closest to cubic. Also note the low message size (less than 1 KBytes) for which half of the peak bandwidth is achieved. This makes the algorithm suitable for short messages.

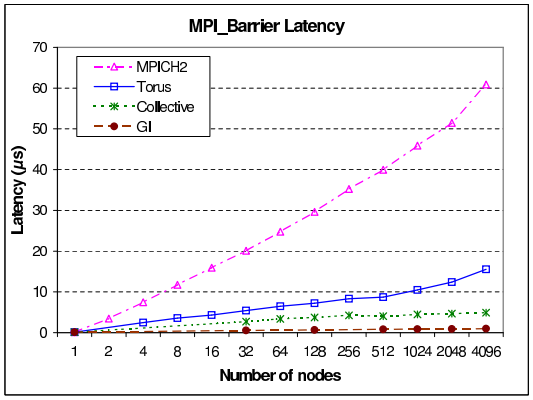


Figure 10: Latency comparison of `MPI_Barrier`

5.4 Latency of short-message MPI_Allreduce

Figure 9(a) displays scaling properties of multiple implementations of short-message `MPI_Allreduce`. We compare the optimized implementations using the collective and torus networks with the default MPICH2 implementation. The message length is 8 bytes. Figure 9(b) shows the same three algorithms running on a fixed size partition but with variable message size (up to 232 bytes). The torus optimized version of `MPI_Allreduce` is faster then the MPICH2 version and scales better. The collective network implementation of `MPI_Allreduce` always has the lowest latency and the best scalability for short messages.

5.5 Latency of MPI_Barrier

Figure 10 compares the latency of four implementations of `MPI_Barrier`. Similar to the latency comparison of `Allreduce`, the torus implementation performs much better than the default MPICH2 implementation in terms of both latency and scalability. The collective network `Barrier` implementation scales even better than the torus `Barrier` and the execution time is less than 5 μs for up to 4096 nodes. The GI `Barrier` implementation has the lowest latency of around 1 μs. Because the collective and GI `Barrier` are not applicable to all machine configurations they have fewer data points than the torus and MPICH2 `Barrier`.

6. CONCLUSION AND FUTURE WORK

Tables 3 and 4 summarize the performance of the MPI collectives discussed in this paper. $\frac{B}{2}$ denotes the message size where *half* of the maximum bandwidth is achieved.

Our optimized MPI collective implementations are superior to the default ones because they exploit knowledge of the physical network topology and are tuned to use performance features of the hardware and to avoid things that carry heavy performance penalty (like memory copies). We spent a lot of time optimizing the collectives, and we are a long way from being done. The extreme scale of the BlueGene/L and the inherent cost of operating it make it worthwhile to develop these algorithms even if they were applicable only to this machine. Time will tell whether the lessons we learned will be applicable to other systems. We suspect that it will – more and more large machines are built to

Collectives	MPICH2		Torus		Tree	
	BW	$\frac{B}{2}$	BW	$\frac{B}{2}$	BW	$\frac{B}{2}$
MPI_Allreduce(i)	40.0	4KB	79.6	512KB	336.7	2KB
MPI_Allreduce(d)	40.5	4KB	119.4	1MB	25.5	256B
MPI_Alltoall(/n)	58.3	8KB	151.6	1KB	-	-
MPI_Bcast	66.0	16KB	243.8	65KB	336.1	16KB

Table 3: Bandwidth (MB/s) summary of MPI collectives on a 512-node BlueGene/L system

Collectives	MPICH2		Torus		Tree		GI
	16B	256B	16B	256B	16B	256B	
MPI_Allreduce	42.3	78.4	19.7	52.8	4.22	4.75	-
MPI_Bcast	15.9	24.0	73.8	78.5	45.2	45.7	-
MPI_Barrier	40.0		8.72		4.04		0.82

Table 4: Latency (μs) summary of MPI collectives on a 512-node BlueGene/L system

compensate for the slowdown in increase of individual CPU performance.

Our ongoing research effort to further optimize BlueGene/L MPI collective communication is threefold: (a) to support a more complete set of MPI collectives; (b) to optimize performance for a larger subset of topologies, i.e. other than MPI_COMM_WORLD and rectangular communicators, and (c) to address the performance deficiencies of the current implementations by e.g. deploying the second processor to help with CPU intensive tasks in our collective implementations.

7. ACKNOWLEDGMENTS

We would like to thank Gabor Dozsa and the anonymous reviewers for their helpful comments.

8. REFERENCES

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

[2] N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.

[3] G. Almasi, C. Archer, J. G. Castaños, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Rattermann, N. Smeds, B. Steimacher-burow, W. Gropp, and B. Toonen. Implementing MPI on the BlueGene/L supercomputer. In *Proceedings of Euro-Par 2004 Conference*, Lecture Notes in Computer Science, Pisa, Italy, August 2004. Springer-Verlag.

[4] G. Almasi, C. Archer, J. Gunnels, P. Heidelberger, X. Martorell, and J. E. Moreira. Architecture and performance of the BlueGene/L Message Layer. In *Proceedings of the 11th EuroPVM/MPI conference*, Lecture Notes in Computer Science. Springer-Verlag, September 2004.

[5] G. Almasi, 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. 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] M. Barnett, R. J. Littlefield, D. G. Payne, and R. A. van de Geijn. Global combine on mesh architectures with wormhole routing. In *International Parallel Processing Symposium*, pages 156–162, 1993.

[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. K. S. Gupta and D. K. Panda. Barrier synchronization in distributed-memory multiprocessors using rendezvous primitives. In *Proceedings of the 7th IEEE International Parallel Processing Symposium – IPPS'93*. IEEE Press, 1993.

[11] D. Hensgen, R. Finkel, and U. Manbet. Two algorithms for barrier synchronizatio. *International Journal of Parallel Programming*, 17(1):1–17, February 1998.

[12] 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.

[13] D. K. Panda. Global reduction in wormhole k-ary n-cube networks with multidestination exchange worms. In *IPPS: 9th International Parallel Processing Symposium*. IEEE Computer Society Press, 1995.

[14] R. Rabenseifne. A new optimized mpi reduce algorithm. High-Performance Computing-Center, University of Stuttgart, November 1997. http://www.hlrs.de/mpi/myreduce.html.

[15] R. Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, June 2004.

[16] R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. In *Proceedings of the 11th EuroPVM/MPI conference*. Springer-Verlag, September 2003.

[17] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 2005.

[18] 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.

[19] 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.

[20] J. Watts and R. Van De Geijn. A pipelined broadcast for multidimensional meshes. *Parallel Processing Letters*, 5(2):281–292, 1995.