MPI on Blue Gene/L : Designing an Efficient General Purpose Messaging Solution for a Large Cellular System

George Almási[1], Charles Archer[2], José G. Castaños[1], Manish Gupta[1], Xavier Martorell[1], José E. Moreira[1], William Gropp[3], Silvius Rus[4], and Brian Toonen[3]

[1] IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
{gheorghe,castanos,jmoreira,mgupta,xavim}@us.ibm.com
[2] IBM Rochester MN 55901
archerc@us.ibm.com
[3] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne IL 60439
{gropp,toonen}@mcs.anl.gov
[4] Computer Science Department
Texas A&M University
College Station TX 77840
rus@tamu.edu

Abstract. The Blue Gene/L supercomputer uses system-on-a-chip integration and a highly scalable 65,536 node cellular architecture to deliver 360 Teraflops of peak computing power. Efficient operation of the machine requires a fast, scalable and standards compliant MPI library. Researchers at IBM and Argonne National Labs are porting the MPICH2 library to Blue Gene/L . We present the current state of the design and project the features critical to achieving performance and scalability.

1 Introduction

In November 2001 IBM announced a partnership with Lawrence Livermore National Laboratory to build the Blue Gene/L supercomputer, a 65,536-node machine designed around embedded PowerPC processors. Through the use of system-on-a-chip integration [8], coupled with a highly scalable cellular architecture, Blue Gene/L will deliver 180 or 360 Teraflops of peak computing power, depending on utilization mode. Blue Gene/L represents a scalability jump of almost two orders of magnitude when compared to existing large scale systems, such as ASCI White [2], the Earth Simulator [4], Cplant [3] and ASCI Red [1].

MPICH2 [5], developed by researchers at Argonne National Laboratories, is an all-new implementation of MPI that is intended to support research into both MPI-1 and MPI-2. The MPICH2 design features optimized MPI datatypes, optimized remote memory access (RMA), high scalability, usability and robustness.

In this paper we present and analyze the software design for a fast, scalable and standards compliant MPI communication library, based on MPICH2 , for the Blue Gene/L machine. The rest of this paper is organized as follows. Section 2 presents a brief description of the Blue Gene/L supercomputer. Section 3 discusses the system software. Section 4 gives a high level architectural overview of the communication library, and Section 5 discusses the design choices we are facing during implementation. Section 6 describes the methodology we employed to measure performance, and preliminary results. We conclude with Section 7.

2 Blue Gene/L hardware overview

A detailed description of the Blue Gene/L hardware is provided in [7]. In this section we present a short overview of the hardware as background for the discussion on the design of the communication library.

The basic building block of the system is a custom system-on-a-chip that integrates processors, memory and communications logic in the same piece of silicon. A chip contains two 32-bit embedded PowerPC 440 cores. Each core has embedded L1 instruction and data caches (32 KBytes each); the L2 cache is 2 KBytes, is shared by the processors, and acts as a prefetch buffer for the 4 MBytes of L3 cache (also shared).

Each core drives a custom 128-bit “double” FPU: essentially a pair of conventional PPC 440 FPUs joined together in SIMD mode, that can perform four double precision floating-point operations per cycle, and load/store 128 bit operands. The theoretical peak FPU performance of the core pair is 5.6GFlop/s assuming the projected clock speed of 700 MHz.

The PPC440 cores are not designed to support multiprocessor architectures. Their L1 caches are not coherent and the architecture lacks atomic memory operations. To compensates, the chip provides a number of synchronization devices: a lockbox unit for fast atomic test-and-set operations, 8 KBytes of shared SRAM and an EDRAM scratchpad for inter-core data exchange, and a “blind device” for explicit cache management.

Large Blue Gene/L systems are built as shown in Figure 1. Two chips, together with about 256 MBytes of DDR memory, are placed on a *compute card*. Sixteen compute cards are plugged in a node board; 32 node boards fit into a cabinet. The complete target system has 64 cabinets, totaling 64K compute nodes and 16 TB of memory.

In addition to the 64K compute nodes, Blue Gene/L contains a number of I/O nodes (1024 in the current design), which are connected to a Gigabit ethernet and serve as control and file I/O concentrators for the compute nodes.

Blue Gene/L has five different networks. The torus network is designed for communication among compute nodes, and is the primary network for the MPI library. The tree network encompasses all nodes and is used by MPI, by the control system for program loading and by file I/O. The other networks are the multipurpose Gigabit ethernet connecting a small number of nodes, the JTAG network used for booting and control and the global interrupt network used for job start, checkpoints and barriers.

The 64K nodes are organized into a partitionable 64x32x32 three-dimensional torus network. Each compute node contains six bi-directional torus links for direct connection with nearest neighbors. The network hardware guarantees reliable and deadlock-free, but unordered, delivery of variable length (up to 256 bytes) packets, using a minimal adaptive routing algorithm. It also provides simple broadcast functionality by depositing packets along a route. At 1.4 Gb/s per direction, the bisection bandwidth of a 64K node system is 360 GB/s. The I/O nodes are not connected to the torus network.

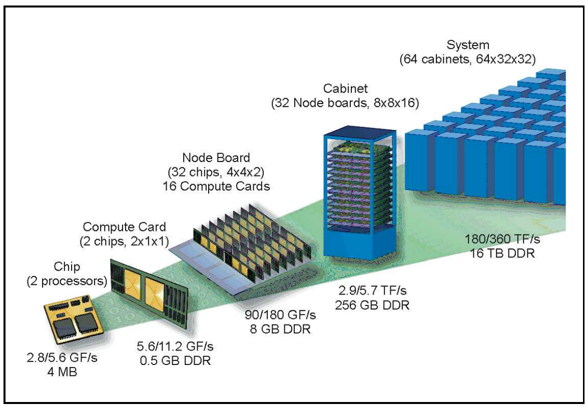

Fig. 1. High-level organization of Blue Gene/L

The tree network supports fast configurable point-to-point, broadcast and reductions of packets, with a hardware latency of 1.5 microseconds for a 64K node system. An ALU in the network can combine incoming packets using bitwise and integer operations, forwarding a resulting packet along the tree. Floating point reductions can be performed in two phases (one for the exponent and another one for the mantissa) or in one phase by converting the floating-point number to an extended 2048-bit representation. I/O and compute nodes share the tree network. Tree packets are the main mechanism for communication between these nodes.

3 Blue Gene/L System Software

The Blue Gene/L computational core is divided into partitions: self-contained and completely (electrically) isolated subsets of the machine. This isolation can be effected at midplane boundaries. The smallest physically isolated contiguous torus in Blue Gene/L consists of 512 compute nodes. It is possible to create smaller (128 node) contiguous meshes, but not tori, by selectively disabling certain torus and tree links on some chips.

The system software architecture is presented in Figure 2. The smallest unit independently controlled by software is called a processing set (or pset) and consists of 64 compute nodes and an associated I/O node. Components of a pset communicate through the tree network; file I/O and control are managed by the associated I/O node through the Ethernet network. The smallest logical entity that can run a job is made of two psets and forms a 128 node contiguous mesh. The whole Blue Gene/L computational core consists of 1024 psets.

The **I/O nodes** run an embedded Linux kernel (currently version 2.4.19) for PowerPC 440GP processors. The kernel has custom drivers for the Ethernet and tree devices. The main function of I/O nodes is to run a control program called CIOD (console I/O daemon) that is used by system management software and by the compute nodes’ file I/O operations.

The control software running on **compute nodes** is a minimalist POSIX compliant custom kernel called BLRTS that provides a simple, flat, fixed-size 256MB address space, with no paging, accomplishing a role similar to PUMA [17]. The kernel and application program share the same address space. In the current implementation, the entire torus network is mapped into user space while the tree network is partitioned between the kernel and the user. The kernel also provides support for a range of options of using the two processors on the chip, which will be discussed later in the paper.

The system management software provides a range of services for the whole machine, including machine initialization and booting, system monitoring, job launch and termination, and file I/O. System management is provided by external service nodes; from a system point of view the compute and I/O nodes are stateless. We chose to maintain all the state of a Blue Gene/L system using standard database technology, which naturally provides scalability, reliability, security, portability, logging, and robustness.

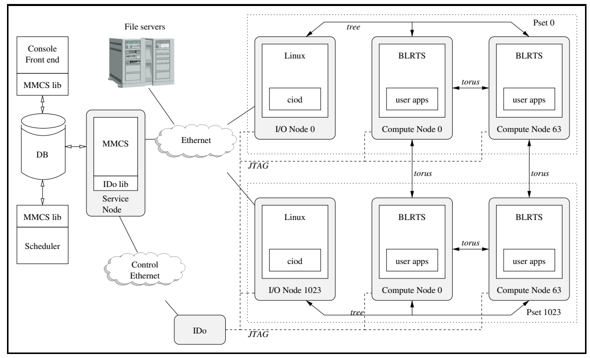

Fig. 2. Blue Gene/L system software

Job execution in Blue Gene/L is accomplished through a combination of I/O nodes and service node functionality. When submitting a job for execution in Blue Gene/L , the user specifies the desired shape and size of the partition to execute that job. Each compute node executes exactly one compute process of the parallel job. The scheduler selects an appropriate set of compute nodes to form the partition. The compute (and corresponding I/O) nodes selected by the scheduler are configured into a partition by the service node using the control network. We have developed techniques for efficient allocation of nodes in a toroidal machine that are applicable to Blue Gene/L [13].

Once a partition is created, job launch and file I/O are accomplished via messages passed between the compute node and its control node over the tree network, using a point-to-point packet addressing mode. This functionality is provided by the I/O node for all compute nodes in a pset.

4 Communication Software Architecture

The Blue Gene/L communication software architecture is divided into three layers. At the bottom is the *packet layer*, a thin software library that allows access to network hardware. At the top is the MPI library. Traditional systems have many other layers of software in between. The relatively high bandwidth/FLOP ratio of the Blue Gene/L design requires low software overhead for communication; therefore a single layer, called the *message layer*, glues the system together.

4.1 Packet layer

The Blue Gene/L chip network hardware is a set of memory mapped FIFO registers and device control registers (DCRs). The torus/tree packet layer is a thin layer of software designed to abstract and simplify access to hardware. It abstracts FIFO registers into torus and tree *devices* and presents an API consisting of essentially three functions: initialization, packet send and packet receive. The packet layer provides a *mechanism* to use the network hardware but doesn't impose any *policies* on how to use it.

Some restrictions imposed by hardware are not abstracted at packet level for performance reasons. For example the length of a torus packet must be a multiple of 32 bytes, and can be no more than 256 bytes. Tree packets have exactly 256 bytes.

Packets sent and received by the packet layer have to be aligned to a 16 byte address boundary, to enable the efficient use of 128 bit loads and stores to the network hardware through dual floating point registers.

All packet layer send and receive operations are non-blocking, leaving it up to the higher layers to implement synchronous, blocking and/or interrupt driven communication models. In its current implementation the packet layer is stateless.

4.2 Message layer

The message layer is an active message system [19, 10, 15, 18] built on top of the packet layer that allows the transmission of arbitrary buffers among compute nodes. Its architecture is shown by Figure 3:

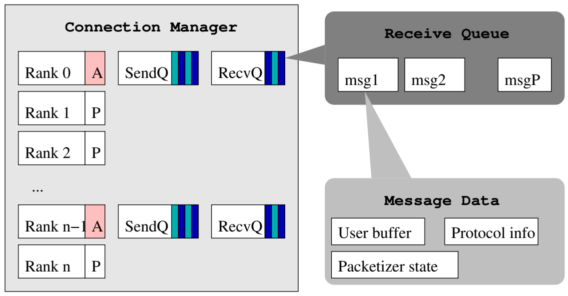

Fig. 3. The message layer architecture

The connection manager controls the overall progress of the system and contains a list of virtual connections to other nodes.

Each **virtual connection** is responsible for communicating with one peer. The connection has a send queue and a receive queue. Outgoing messages are always sent in order. Incoming packets, however, can arrive out of order: the message layer has to determine which message a packet belongs to. Thus, each packet has to carry a message ID.

Message buffers are used for sending and receiving packets belonging to the same message. A message buffer contains the state of the message (in progress, complete etc.). It also has an associated region of user memory, and a *packetizer/unpacketizer* that is able to generate packets or to place incoming packets into memory. Message buffers also handle the message protocol (i.e. what packets to send when).

[Un]packetizers drive the packet layer. Packetizers build and send packets out of message buffers; unpacketizers re-constitute messages from the component packets. Packetizers also handle the alignment and packet size limitations imposed by the network hardware.

The three main functions implemented by the message layer API are `Init`, `advance` and `postsend`. `Init` initializes the message layer. `advance` is called often to ensure that the message layer makes progress, i.e. sends the packets it has to send, checks the torus hardware for incoming packets and processes them accordingly. `postsend` allows a message to be submitted into the send queue. Other functions, not described here, allow the implementation of application-supported checkpointing.

Just like packet layer functions, message layer functions are non-blocking and designed to be used in either polling mode, or driven by hardware interrupts. Completion of a send, and the begin and end of a receive are all signalled through *callbacks*. Thus, when a message is sent and is ready to be taken off the send queue the `senddone` function is invoked. When a new message starts arriving, the `recvnew` callback is invoked; at the end of reception `recvdone` is invoked.

4.3 MPICH2

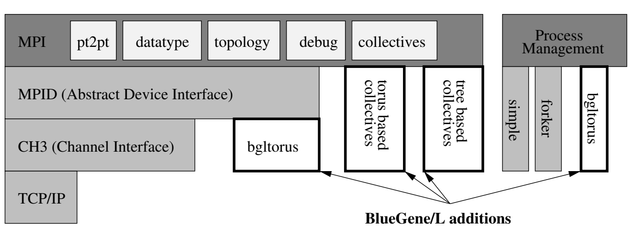

Fig. 4. The Blue Gene/L MPI roadmap

The large number of MPI processes on a Blue Gene/L machine create scalability problems for most MPI implementations we considered for this machine. MPICH2 , currently under development at Argonne National Laboratories, is an MPI implementation designed from the ground up for scalability to hundreds of thousands of processors. Figure 4 shows the roadmap of developing an MPI library for Blue Gene/L . MPICH2 has a modular build, and therefore the Blue Gene/L port consists of a number of plug-in modules, leaving the code structure of MPICH2 intact.

Point-to-point messages. The most important addition of the Blue Gene/L port is an implementation of ADI3, the MPICH2 Abstract Device Interface [12]. A thin layer of code transforms e.g. MPI `Request` objects and `MPI_Send` function calls into calls into sequences of message layer `postsend` function calls and various message layer callbacks.

Process management. Another part of the Blue Gene/L port is related to the process management primitives, documented in [6]. In MPICH2 process management is split into two parts: a process management interface (PMI), called from within the MPI library, and a set of process managers (PM) which are responsible for starting up and terminating down MPI jobs and implementing the PMI functions.

MPICH2 includes a number of process managers suited for clusters of general purpose workstations. The Blue Gene/L process manager makes full use of its hierarchical system management software, including the CIOD processes running on the I/O nodes, to start up and shut down MPI jobs. The Blue Gene/L system management software is expressly designed to deal with the scalability problem inherent in starting up, synchronizing and killing 65,536 MPI processes.

When complete, the system management software will also allow us to completely implement most of the PMI functions specified by MPICH2 . Currently our PMI implementation is only minimally functional, and e.g. only allows a default mapping of MPI ranks to torus coordinates; when fully functional, the PMI will be coupled with the system manager's implementation of `mpirun` to allow users to ask for specific physical topologies and MPI rank assignments to run their jobs on.

Optimized collectives. MPICH2 has default implementations for all MPI collectives, and therefore becomes functional the moment point-to-point primitives are implemented. The default implementations are oblivious of the underlying physical topology of the torus and tree networks. **Optimized** collective operations can be implemented for communicators whose physical layouts conform to certain properties.

Building optimized collectives for MPICH2 involves several steps. First, the process manager interface will be expanded to allow the calculation of the torus and tree layouts of particular communicators. Next, a list of optimized collectives, for partilar combinations of communicator layouts and message types, will be implemented. The best implementation of a particular MPI collective will then be selected based on the type of communicator involved (as calculated using the process manager interface).

- The torus hardware can be used to efficiently implement broadcasts on contiguous 1, 2 and 3 dimensional meshes, using a feature of the torus that allows depositing a packet on every node it traverses (as mentioned in Section 2). Collectives best suited for this implementation e.g. `Bcast`, `Allgather`, `Alltoall`, `Barrier`, all involve broadcast in some form.
- The tree hardware can be used for almost every collective that is executed on the `MPI_COMM_WORLD` communicator, including some reduction operations. Integer operand reductions are directly supported by hardware. IEEE compliant floating point reductions can also be implemented by the tree using separate reduction phases for the mantissa and the exponent.
- Non `MPI_COMM_WORLD` collectives can also be implemented using the tree, but care must be taken to ensure deadlock free operation. The tree is a locally class routed network, with packets belonging to one of a small number of classes and tree nodes making local decisions about routing. The tree network guarantees deadlock-free simultaneous delivery of no more than two class routes. One of these routes is used for control and file I/O purposes; the other is available for use by collectives.

Optimized collectives are not yet implemented, since our priority is to achieve a fully functional MPI implementation before the Blue Gene/L hardware arrives.

5 Design Decisions in the Message Layer

The message layer's design was driven by the requirement to find a low overhead solution with good scaling properties that interfaces the MPI library with the packet layer. The design was influenced by specific Blue Gene/L hardware features, such as

- the network's reliability,
- packetization and alignment restrictions,
- out-of-order arrival of torus packets, and
- the existence of non-coherent processors in a chip.

These hardware features, together with the requirements, led us to design decisions that deserve closer examination.

5.1 The impact of hardware reliability

The Blue Gene/L network hardware is completely reliable. Once a packet is injected into the network, hardware guarantees its arrival at the destination unless a non-correctable error condition occurs at one of the nodes (resulting in an abort and checkpoint restart). The Blue Gene/L message layer does not implement a packet recovery protocol. This decision allows for better scaling and a large reduction of software overhead, but also introduces problems related to checkpointing and recovery from software failures.

5.2 Dealing with network hardware: packetizing and alignment

The packet layer requires data to be sent in 256 byte chunks aligned at 16 byte boundaries. This forces the message layer to either optimize the alignment of arbitrary buffers or to copy memory to/from aligned data buffers.

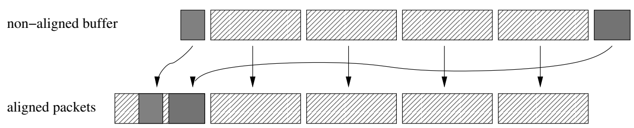

Fig. 5. Packetizing non-aligned data

Figure 5 illustrates the principle of optimizing the alignment of long buffers. The buffer is carved up into aligned chunks where possible. The two non-aligned chunks at the beginning and at the end of the buffer are copied and sent together. This strategy is not always applicable, because the alignment *phase* (i.e. the offset from the closest aligned address) of the sending and receiving buffers may differ. MPI has no control over the allocation of user buffers. In such cases at least one of the participating peers, preferrably the sender, has to adjust alignment by performing memory to memory copies. For rendezvous messages the receiver can send back the *desired alignment phase* with the first acknowledgement packet.

The alignment problem only affects zero copy message sending strategies. If either peer in a message exchange uses memory-to-memory copies for any reason, that memory copy can absorb the cost of re-alignment.

5.3 Out-of-order packets

The routing algorithm of the torus network allows packets from the same sender to arrive to the receiver out of order. The task of re-ordering packets falls to the message layer.

Packet order anomalies affect the message layer in one of two ways. The simpler case occurs when packets belonging to the same message are received out of order. This affects the way in which packets are re-assembled into messages, and the way in which MPI matching is done at the receiver (since typically MPI matching information is in the *first* packet of a message).

Packet order reversal can also occur to packets belonging to different messages. To prevent the mixing of packets belonging to different messages, each packet has to carry a message identifier. This identifier is maintained by the sender's virtual channel, and incremented for every message sent. To comply with MPI semantics, the receiver is responsible to present incoming messages to MPI in strictly increasing order of the message identifier.

5.4 Cache coherence and processor use policy

As mentioned before, each Blue Gene/L compute node incorporates two non cache-coherent PowerPC 440 cores sharing the main memory and the torus and tree devices. Several scenarios for using these CPUs have been proposed.

- *Heater mode* does not do anything with the second processor beyond booting it and putting it into an idle loop. Heater mode is easy to implement, because it sidesteps all the issues related to the lack of cache coherency of the cores, and to resource sharing.
- *Virtual node mode* allows the co-existence of the two processors but severly limits interaction between the two: the applications running on them have separate allocations of memory and communication resources. Virtual node mode doubles the available processing power available to the user at the cost of halving all other resources per process. It is well suited for computation-intensive jobs that require little in the way of memory or communication. Neither the torus nor the tree networks are suitable for communication between two processes on the same chip. A shared area of memory is called for, either in a non-L1-cached portion of the main memory or in the scratchpad memory area (described in Section 2). The shared memory can be viewed as a virtual torus/tree device, and we plan to implement a virtual packet layer over this shared memory to send/receive packets in the same way as with other nodes.

– *Co-processor mode*, envisioned by the designers of Blue Gene/L as the default mode of operation, assigns one processor to computation and another to communication, effectively overlapping them by freeing the main ("compute") processor from communication tasks.

The main obstacle to implementing co-processor mode efficiently is the lack of cache coherence between the processors doing computation and communication. A naive way to avoid this issue is to set up a shared memory area and a virtual torus device just like in virtual node mode; the computate processor treats the virtual torus device, backed by shared memory buffers, as the only torus device in the system. The communication processor is reduced to moving data between the real device and the virtual device.

The naive implementation of co-processor mode still requires the compute processor to packetize and copy data into the shared memory area. However, reads and writes to/from the shared memory area can be done about four times faster than to/from the network devices, reducing the load on the compute processor by the same amount.

For better performance, however, it is necessary to de-couple the process of message processing from the compute processor even more. The communication processor is able to touch application memory assuming that the correct cache invalidation/flush protocol is observed by both the compute and communication processors. Before sending an MPI message the compute processor has to insure that the user's buffer has been flushed to main memory. The communication processor accesses all main memory in non-cached mode, because there is no benefit from temporal reuse for what is essentially a network DMA engine streaming data. When receiving a message, the compute processor has to invalidate the cache lines associated with its receive buffer before allowing the communication processor to fill it in with data.

Special care must be taken when receiving data that is not aligned at cache line boundaries, because the compute processor may inadvertently touch data on the same cache line as the received buffer while the message is being received, resulting in data corruption. We expect not to receive such data using the communication co-processor.

There are also other situations when the communication co-processor should not be used at all: for short blocking messages latency is the main issue and offloading the compute processor doesn't bring any benefits. In this case even heater mode may work better than co-processor mode.

5.5 Scaling issues and virtual connections.

In MPICH2 point to point communication is executed over virtual connections between pairs of nodes. Virtual connections are established between each pair of nodes in a lazy manner. In a 65,536 node machine the large number of such connections maintained by every participating peer limit scalability.

Because the network hardware guarantees packet delivery, virtual connections do not have to execute a per-connection wake-up protocol when the job starts. Thus startup time on the Blue Gene/L machine will be a constant, not a linear function of the number of participating nodes.

Another factor limiting scalability is the amount of memory needed by an MPI process to maintain state for each virtual connection. The current design of the message layer uses only about 50 bytes of data for every virtual connection for the torus coordinates of the peer, pointer sets for the send and receive queues and state information. Even so, 65,536 virtual connections, add to 3 MBytes of main memory per node, more than 1% of the available total (256 MBytes), just to maintain the connection table.

5.6 Application-supported checkpointing

There is no software in place to re-send packets lost on the network during checkpointing. The torus and the tree hardware to be emptied of packets prior to taking a checkpoint snapshot.

For system-initiated checkpoints this means querying the network hardware to make sure that no packets are in the system. But application driven checkpoints do not have access to low level network hardware. Because packets on the torus network may arrive out of order, the only way to guarantee that no packet is in transit is to checkpoint in a state where the receive queues of all virtual connections are empty. Checkpoint markers have to be sent, and acknowledged, by every virtual connection to their peers. Application initiated checkpointing implies a barrier operation.

5.7 Transmitting non-contiguous data

The MPICH2 abstract device interface allows non-contiguous data buffers to percolate down to message layer level, affording us the opportunity to optimize the marshalling and unmarshalling of these data types at the lowest (packet) level. Our current strategy centers on `iovec` data structures generated by utility functions in the ADI layer.

5.8 Communication protocol in the message layer

Early in the design we made the decision to implement the communication protocol in the message layer for performance reasons. Integration with MPICH2 is somewhat harder, forcing us to implement an abstract device interface (ADI3) based port instead of using the easier, but less flexible channel interface [12]. In our view the additional flexibility gained by with this decision is well worth the effort.

The most important reason for abandoning the channel interface was the need for a custom protocol in the message layer. The protocol design is crucial because it is influenced by virtually every aspect of the Blue Gene/L system: the reliability and out of order nature of the network, scalability issues and latency and bandwidth requirements.

- Because of the reliable nature of the network no acknowledgements are needed. Thus a simple “fire and forget” eager protocol is a viable proposition; any packet out the send FIFO can be considered safely received by the other end.
- A special case of the eager protocol is represented by single-packet messages, which should be handled with a minimum of overhead to achieve good ping-pong latency.
- The main limitation of the eager protocol is the inability of the receiver to control incoming traffic. For high volume messages the rendezvous protocol is called for, possibly the optimistic form implemented in Portals [9].
- The message protocol is also influenced by out-of-order arrival of packets. The first packet of any message contains information not repeated in other packets, such as the whole message’s lenght and MPI matching information. If this packet is delayed on the network, the receiver is unable to handle the subsequent packets, and has to allocate temporary buffers or to discard the packets, with obvious undesirable consequences.
  This problem does not affect the rendezvous protocol because the first packet is always explicitly acknowledged and thus cannot arrive out of order. For short messages, where the rendezvous protocol is not desirable, the problem can be mitigated in several ways. The torus network permits us to assign higher priority to the lead packet of a message, lowering the probability of an out-of-order situation. However, the problem can only be fully handled by replicating the extra information in all packets of a message, at the price of substantial (10-20%) loss of bandwidth.
- The most promising solution to the out-of-order problem for mid-size messages involves a variation on the rendezvous protocol that replicates the MPI matching information in the first few packets belonging to a message, and requires the receiver to acknowledge the first packet it receives. The number of packets that have to carry extra information is determined by the average roundtrip latency of the torus network. The sender will not have to stop and wait for an acknowledgement if it is received before the allotment of special packets carrying extra information has been exhausted.

6 Simulation Framework and Measurements

BGLSIM is a functionally correct multichip Blue Gene/L simulator based on the MAMBO [16] project. BGLSIM is not architecturally accurate. All instructions take one one cycle to execute. Up to 2,000,000 instructions per second are simulated on a 1GHz Pentium III machine. The high execution speed has enabled us to run multi-chip simulations of up to 512 simulated nodes.

BGLSIM is our primary vehicle for software development. It is equipped with an implementation of HPM [11] which allows us to measure the number of instructions executed by regions of instrumented code.

This section illustrates the measurement methodology we are using to drive our design decisions, and how we are planning to optimize the implementation of the MPI port. The numbers presented here are current as of April 2003, and were measured with the first version of the Blue Gene/L port of MPICH2 that was able to run in the multichip simulation environment. We were interested in measuring the software overhead in the MPICH2 port and in the message layer. The workloads for our experiments consisted of a subset of the NAS parallel benchmarks [14], running on 8 processors.

Figure 6 shows a simplified call graph for sending a blocking MPI message, with the functions of interest to us highlighted. We instrumented these functions, and their counterparts on the receive end, with HPM library calls. HPM counted the average number of **instructions** per invocation.

Table 1 summarizes the measurements. The left panel in the table contains measurements for the high level functions of the MPICH2 port. As the table shows, blocking operations (`MPI_Send` and `MPI_Recv`) are not very good indicators of software overhead, because the instruction counts include those spent waiting for the simulated network to deliver packages. The numbers associated with non-blocking calls like `MPI_Isend` and `MPI_Irecv` are a much better measure of software overhead.

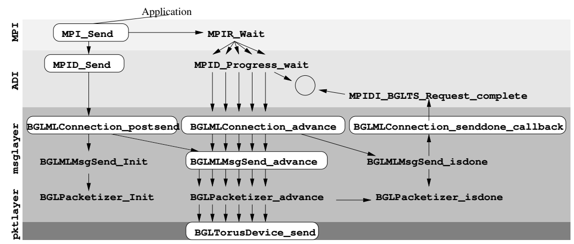

Fig. 6. The callgraph of an `MPI_Send()` call

Overhead (insns.)	FT	BT	SP	CG	MG	IS	LU
`MPI_Send`				11652	10479	3746	7129
`MPID_Send`				1759	1613	1536	1744
`MPI_Isend`		2043	2162				
`MPID_Isend`	1833	1782	1901				
`MPI_Irecv`		541	542	549	564	536	557
`MPID_Irecv`	280	279	280	293	308	280	301
`MPI_Recv`							13811
`MPID_Recv`							406

Overhead (insns.)	FT	BT	SP	CG	MG	IS	LU
`postsend`	1107	1271	1401	1230	1114	1220	1265
`senddonecb`	115	115	115	115	115	115	115
`recvnewcb`	445	344	353	349	335	341	328
`recvdonecb`	16179	418	333	267	150	204	127
`advance`	2181	1643	1781	1429	1669	2865	955
`msgsend_adv`	671	653	648	620	556	642	594
`dispatch`	520	518	516	598	661	533	620

Table 1. Software overhead measurements for MPICH2 and message layer functions

The right panel in the table contains data for message layer functions. `postsend` is the function called to post a message for sending; it includes the overhead for sending the first packet. `senddonecb` is called at the end of every message send. It shows the same number of instructions in every benchmark. `recvnewcb` (called for every new incoming message) has a slightly higher overhead because this is the function that performs the matching of an incoming message to the requests posted in the MPI request queue. The `recvdonecb` numbers show a high variance, because in certain conditions this callback copies the message buffer from the unexpected queue to the posted queue. In our measurements this happened in the FT benchmark.

The remaining two lines in the right panel of the table represent the amount of instructions spent by the message layer to get a packet into the torus hardware (`msgsend_adv`) or out of the torus hardware (`dispatch`).

An `MPI_Isend` call in the BT benchmark takes about 2000 instructions. Out of these, the call to `postsend` in the message layer accounts for 1300 instructions. `postsend` calls `msgsend_adv` to send the first packet of the message. `msgsend_adv` spends an average of 671 instructions sending the packet. Thus the software overhead of `MPID_Send` can be broken down as $2000 - 1300 = 700$ instructions spent in the MPICH2 software layers, $1300 - 671 = 629$ instructions spent in administering the message layer itself and 671 instructions spent to send every packet from the message layer.

The above reasoning points at least one place to where the message layer can be improved. The minimum number of instructions necessary to send/receive an aligned packet is 50. However, the message layer spends more than 650 instructions for the same purpose, partially because of suboptimal implementation, alignment adjustment through memory copies and packet layer overhead. We are confident that a better implementation of the message sender/receiver can reduce the packet sending overhead by 25-50%.

7 Conclusions

In this paper we have presented a software design for a communications library for the Blue Gene/L supercomputer based on the MPICH2 software package. Because of the high bandwidth per chip speed ratio of the machine, the design concentrates on achieving very low software overheads. Concentrating on point-to-point communication, the paper presents the design decisions we have already made and the simulation-based methodology we are planning to use to drive our design.

References

1. ASCI Red Homepage. `http://www.sandia.gov/ASCI/Red/`.
2. ASCI White Homepage. `http://www.llnl.gov/asci/platforms/white`.
3. Cplant homepage. `http://www.cs.sandia.gov/cplant/`.
4. Earth Simulator Homepage. `http://www.es.jamstec.go.jp/`.
5. The MPICH and MPICH2 homepage. `http://www-unix.mcs.anl.gov/mpi/mpich`.
6. Process Management in MPICH2. Personal communication from William Gropp.
7. N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.
8. G. Almasi et al. Cellular supercomputing with system-on-a-chip. In *IEEE International Solid-state Circuits Conference ISSCC*, 2001.
9. R. Brightwell and L. Shuler. Design and Implementation of MPI on Puma portals. In *In Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.
10. 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.
11. L. DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
12. W. Gropp and E. Lusk. MPICH Abstract Device Interface Version 3.3 Reference Manual: Draft of October 17, 2002. `http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf`.
13. E. Krevat, J. Castanos, and J. Moreira. Job scheduling for the blue gene/l system. In *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 38–54. Springer, 2002.
14. Nas parallel benchmarks. `http://www.nas.nasa.gov/Software/NPB`.
15. 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 1999*, 1995.
16. H. Shafi, P. Bohrer, J. Phelan, C. Rusu, and J. Peterson. Design and Validation of a Performance and Power Simulator for PowerPC Systems. *IBM Journal of Research and Development*, 2003.
17. 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. In *In Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995.
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.