2012 IEEE 26th International Parallel and Distributed Processing Symposium

PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer

Sameer Kumar[1], Amith R. Mamidala[1], Daniel A. Faraj[2], Brian Smith[2], Michael Blocksome[2], Bob Cernohous[2], Douglas Miller[2], Jeff Parker, Joseph Ratterman[2], Philip Heidelberger[1], Dong Chen[1] and Burkhard Steinmacher-Burow[3]

{sameerk,amithr,philiph,chendong}@us.ibm.com
[1]IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA

{faraja,smithbr,blocksom,bobc,dougmill,jjparker,jratt}@us.ibm.com
[2] IBM Systems and Technology Group
Rochester, MN, 55901

steinmac@de.ibm.com
[3] IBM Research and Development
Boeblingen, Germany, 71032

***Abstract*—The Blue Gene/Q machine is the next generation in the line of IBM massively parallel supercomputers, designed to scale to 262144 nodes and sixteen million threads. With each BG/Q node having 68 hardware threads, hybrid programming paradigms, which use message passing among nodes and multi-threading within nodes, are ideal and will enable applications to achieve high throughput on BG/Q. With such unprecedented massive parallelism and scale, this paper is a groundbreaking effort to explore the design challenges for designing a communication library that can match and exploit such massive parallelism In particular, we present the Parallel Active Messaging Interface (PAMI) library as our BG/Q library solution to the many challenges that come with a machine at such scale. PAMI provides (1) novel techniques to partition the application communication overhead into many contexts that can be accelerated by communication threads; (2) client and context objects to support multiple and different programming paradigms; (3) lockless algorithms to speed up MPI message rate; and (4) novel techniques leveraging the new BG/Q architectural features such as the scalable atomic primitives implemented in the L2 cache, the highly parallel hardware messaging unit that supports both point-to-point and collective operations, and the collective hardware acceleration for operations such as broadcast, reduce, and allreduce. We experimented with PAMI on 2048 BG/Q nodes and the results show high messaging rates as well as low latencies and high throughputs for collective communication operations.**

I. INTRODUCTION

The Blue Gene/Q supercomputer [1] comprises several architectural innovations at different levels of the system architecture. Each BG/Q node contains 18 compute cores, with each core having four hardware threads. One of the cores is a spare core, and another is reserved for the Compute Node Kernel (CNK) lightweight operating system, leaving 16 cores with up to 64 threads for application processing. These cores are connected via a crossbar switch to a shared L2 cache system consisting of 16 L2 cache banks (or slices). Further, to support this high concurrency on a single node, the L2 cache also enables atomic transactions on any arbitrary 8 byte aligned memory address on the node. The BG/Q nodes are connected via a five dimensional (5D) torus [2] designed to scale to 256 racks (256x1024 nodes). The 5D torus boosts the bisection bandwidth of the machine accelerating the performance of applications that have all-to-all communication such as FFT. Unlike its predecessors BG/L [3] and BG/P [4], the collective network on BG/Q is embedded in the 5D torus. The supported operations over the collective network are barrier, broadcast, reduce, and allreduce. Collective communication on contiguous rectangular subsets of nodes is also accelerated by the collective network by programming the *classroutes* of the hardware tree. These operations are extremely scalable. The projected Message Passing Interface (MPI) latencies for barrier and allreduce are expected to be under 9µs and 12µs respectively on 96 racks (96x1024 nodes) of BG/Q. MPI [5] will continue to be the primary inter-node communication mode for applications, while OpenMP is likely to be used within the nodes.

In this paper we present the Parallel Active Messaging Interface (PAMI) library that we use as a foundation to support MPI, and can also be used to efficiently enable other programming paradigms such as UPC [6] and ARMCI [7], and the parallel programming language Charm++ [8]. It is a challenge to design messaging libraries that enable applications to scale to millions of cores and over 10 million threads. Our design supports up to 64 processes per node and sixteen million MPI processes in the largest BG/Q configuration. However, having a large number of processes on a BG/Q node could exert pressure on node resources such as DRAM, network FIFOs and TLB entries, for certain classes of applications. Therefore, we expect hybrid programming models that have fewer MPI processes to achieve the best

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.73

performance on this architecture. However, with fewer processes per node, several threads can call the messaging library stressing the thread scalability of the library. MPI and the messaging libraries must deliver very high message rates for communicating messages from all threads efficiently. The MPI community has been progressing towards the next version of the MPI standard, MPI 3.0. The MPI 3.0 hybrid working group is exploring new concepts in Hybrid programming via *Endpoints* on each process rather than the processes communicating themselves. We have designed the PAMI *contexts* following the developments in the MPI Forum for fine grained communication among threads. The context "process-rank" pair is similar to an MPI 3.0 endpoint.

PAMI contexts can also enable background communication threads to accelerate communication processing. For example, with one MPI process per node (PPN) we can have up to sixteen contexts and sixteen acceleration communication threads. PAMI leverages hardware features of BG/Q nodes such as a low overhead wakeup mechanism to awaken the communication threads. The main application threads can hand off work to the communication threads via a PAMI_context_post function call to maximize messaging parallelism and drive high message rates even with MPI 2.2 style point to point communication. The parallelism extracted via PAMI needs to adhere to the MPI ordering rules which dictate the matching of a MPI send with that of the receive. Wildcard matching tags present an additional challenge which has to be dealt with carefully. In this paper, we explore strategies to map the MPI thread level support to PAMI endpoint parallelism. A hybrid MPI+OpenMP application, where typically the master thread initiates the communication calls, can benefit from the increased message rate.

This paper makes the following contributions:

- We present the Parallel Active Messaging Interface (PAMI) library, through which we answer many challenges in meeting BG/Q massive parallelism
- While PAMI is used as a foundation to support MPI, it can also be used to efficiently enable other programming paradigms such as UPC [6] and ARMCI [7], and the parallel programming language Charm++ [8]. This is done with PAMI client and context objects. A PAMI Client that is an independent network instance, while contexts provide independent communication channels that can be accessed from multiple threads
- We describe lockless algorithms to accelerate MPI message rate
- Novel techniques, that leverage the new architectural features in BG/Q such as the wakeup unit and the collective network, to optimize point to point and collective communication interfaces in PAMI
- This is the first effort that presents performance results on 2048 BG/Q nodes with 128K threads

A. Related Work

Active messages have also been explored in the runtimes for Myrinet such as GM and MX [17,18], LAPI over IBM/SP [21], and DCMF over BG/P[9]. The Common Communication Interface (CCI) [19] is similar to PAMI as it uses endpoints. PAMI differs from the above as the highest abstraction of a network instance is a *Client* that encapsulates all the resources associated with that network instance. PAMI supports multiple clients that can enable simultaneous co-existence of multiple programming model runtimes. This feature can be used to a mixed programming model, like the one explored by researchers in [22], where UPC and MPI were used to scale a memory bound application. In [16], the authors use parallel communication channels to speedup MPI message rate. PAMI extends and generalizes this notion of communication parallelism using PAMI *Contexts* and uses a new message handoff technique to accelerate message rate. Finally, lockless queues using atomic primitives have been studied by many researchers including [20]. PAMI uses the very scalable L2 atomic constructs described below for the high concurrency messaging operations.

II. BACKGROUND

A. BG/Q Overview

Each BG/Q node is comprised of 18 Power ISA A2 64-bit embedded low power processor cores running at 1.6 GHz. Each core has four hardware threads. The hardware threads have their own register files but share other resources such as the L1 and L2 caches. The A2 core can issue two concurrent instructions per cycle, one fixed and one floating point, but each thread can issue only one instruction per cycle. It implements in-order dispatch and execution of the instruction pipeline. The L1 total cache size is 32KB with the instruction and data caches of 16KB each. The L2 cache size is 32MB and is divided into sixteen slices and interconnected to the A2 cores by a crossbar switch. Moreover, each core has a local L1 prefetch unit that can prefetch cache lines from L2 ahead of time.

Scalable Atomic support in L2: BG/Q nodes support different atomic operations such as load-increment, store-update, etc for 64-bit integer words in DDR memory. These are implemented by special atomic addresses that are aliases to the L2/DDR memory. L2 atomics have significantly lower overheads than traditional mutexes. The L2 atomics are scalable with only a few extra cycles for each additional atomic request. L2 atomics are used in several places including lock-less queues and messaging counters that are used to track communication progress.

Wakeup unit: The main purpose of the wakeup unit is to increase application performance by avoiding software polling in A2. The wakeup unit can be programmed to track and recognize memory addresses written by any of the A2 cores, messaging unit, or other devices. It can also be configured to

recognize signals from the network and the A2 cores. The thread can be put into a wait via a special instruction until a desired event occurs. The thread is suspended until it receives a wakeup signal. While the thread is suspended, it does not use core resources such as pipeline slots, arithmetic units, and load/store resources.

B. BG/Q Network Architecture

Each link/port in the BG/Q 5D torus network [2] is capable of simultaneously sending and receiving data at a raw speed of 2GB/sec. As each packet has a 32 byte header and up to 512 bytes of payload, in 32B increments. With other overhead such as packet consistency checks and protocol packets, the maximum achievable throughput for application payload is 1.8GB/sec. Not only is the bidirectional bandwidth increased in BG/Q network compared to a lower dimensional torus with the same number of nodes, but also the 5 torus dimensions reduces the maximum number of hops to reach the farthest node. The five dimensions are labeled A, B, C, D and E with opposing directions indicated by "+" and "-". Each node in the torus has multiple injection and reception FIFOS, enough so that user point-to-point, user collective, system point-to-point and system collectives all have their own FIFOS. Unlike BG/P, on BG/Q the point-to-point network, the collective network and the Global Interrupt (GI) network all share the same torus network. The BG/Q network supports hardware acceleration for collectives such as barrier, broadcast, reduce, and allreduce for both MPI_COMM_WORLD as well as rectangular subcommunicators. This is provided via a classroute that allows the user to program the routes of the collective tree. Each classroute specifies the links that are the down tree inputs to the router and the uptree output. The local contribution is also included, and the tree can skip the contribution from a node depending on whether this bit is on/off. The number of classroutes in which a node can participate is 16; however some are reserved for system use. The collective network supports both integer and floating point operations such as add, min and max.

C. BG/Q Message Unit (MU) Architecture

The BG/Q MU is responsible for moving data between the memory and the 5D torus network. It supports three different point-to-point packet types: memory FIFO, RDMA read, and RDMA write. For all such packet types, the data transfer is initiated by writing a 64B descriptor into one of the MU injection FIFOs. Depending on the type of the packet, the data is either delivered into a MU reception FIFO or is directly written into the memory address included in the packet. BG/Q architecture provides an extensive array of 544 MU injection FIFOs (32 per core) and 272 MU reception FIFOs (16 per core). Also, there multiple message engines, compared to only two on BG/P, that operate in parallel for sending and receiving network packets. Together, all these capabilities of the BG/Q MU provide a high degree of communication parallelism for the application to use. Also, compared to BG/P, the collectives on BG/Q are RDMA capable. For example, an allreduce is performed by the MU sending RDMA write packets that are summed on the network and the result stored in destination buffers on the nodes, also via RDMA writes.

D. Compute Node Kernel Overview

The Compute Node Kernel (CNK) is a light weight kernel on BG/Q providing system interfaces to support efficient message passing operations.

Communication Thread (*commthread)*: In addition to conventional pthreads, CNK provides a special pthread (one per hardware thread) having extended low and high priority levels. This special pthread known as commthread is used to make progress on the various communication operations of the messaging libraries. The extended priorities allow the commthread to perform low-level communications operations without risk of being preempted (at highest priority) and also ensure the commthread is completely out of the way the rest of the time (at lowest priority). Commthreads are reserved for use by BG/Q messaging software.

CNK Support for Shared Address space: To aid message passing within the node, CNK provides global virtual addresses within the node. These addresses are aliases to the virtual addresses of the processes and can be used by any process on the node to read the memory locations of its peers. CNK provides a separate global virtual to physical address translation table containing the global addresses of all the processes on the node. This capability eliminates extra copies in the message passing operations between processes on the same node, both for point-to-point and collective operations.

III. PARALLEL ACTIVE MESSAGING INTERFACE

In this section, we explain the challenges involved in designing efficient communication libraries for meeting the requirements of the high level programming models, such as MPI. Parallel Active Messaging Interface (PAMI) builds upon the techniques used in the DCMF library on BG/P [9] and LAPI [10] on PowerPC systems. In addition, it also enables concurrency in messaging operations by taking advantage of the novel features of the BG/Q architecture.

A. Supporting multiple programming models

PAMI Client: A client can be thought of as an independent network interface with its own set of network and communication resources. It encapsulates all communication data structures, such as contexts and endpoints, communication progress models, and the network/messaging unit resources such as access to the collective tree. Each programming language runtime should create a PAMI client to make PAMI library calls. Figure 1 gives an overview of

PAMI's compontents. A client may instantiate one or more communication contexts. A context is a collection of software communication devices, where progress is made by an application thread or communication thread. Progress is made via the PAMI_Context_advance call, that is thread unsafe and thread safety is a responsibility of higher level software. A thread-safe work queue provides an efficient lock-less hand-off mechanism between application threads and communication threads. Each software device in a context manages a physical partition of the hardware resources and can be advanced independent of other software devices. For example, the shared memory hardware is used by the shared memory device to drive intranode communication, whereas the torus network hardware is used by the MU device to implement off-node communication.

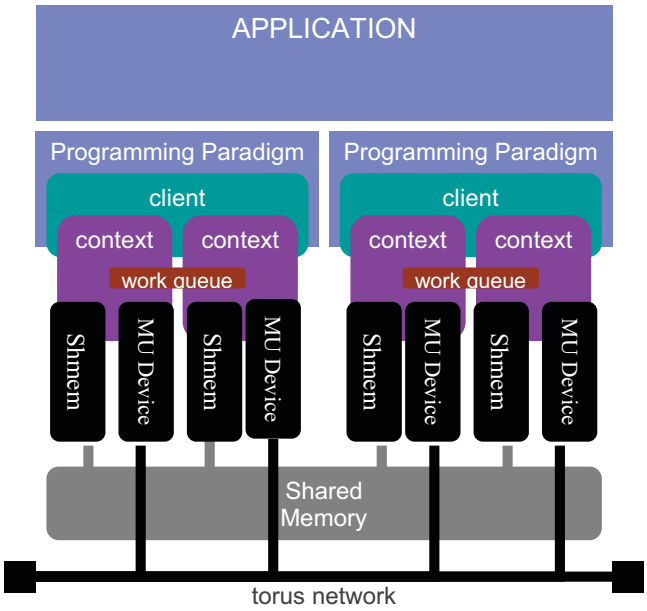

Figure 1. Overview of PAMI

B. PAMI messaging parallelism & concurrency

PAMI Context: A context defines a unit of thread parallelism. Messaging operations are initiated and progressed in the context independent of other co-existing contexts. For example, two different threads can be pinned to two different contexts to achieve independent concurrent communication. Initiating a messaging operation involves either posting a work request via the PAMI_Context_post call to the context to be progressed later, or directly posting to the injection FIFOs of the MU via the PAMI_Send call. The work request queues use L2 atomic operations to provide a low-overhead highly concurrent approach without the use of an explicit lock. Hence, multiple threads can post requests to a single context using this approach. Two threads can simultaneously advance or send messages on two different contexts. If the two threads must simultaneously access (e.g. perform send or collective operation) the same context, then they need to lock/unlock the context. Alternatively, if communication threads are enabled, the main threads can post a work function to a lock-less queue to be executed on the communication thread.

Lockless queues: The L2 Atomic operations provide convenient and scalable atomic constructs that can be used to design communication queues for different message passing operations. One of the supported L2 Atomics operations is "bounded increment". This combines an atomic load-and-increment with a compare against bounds, enabling atomic allocation of elements to a fixed-sized array used to implement a fast scalable queue. This fixed-sized array is enhanced with an overflow queue to handle cases when the array is full. The overflow queue is accessed through mutexes.

PAMI Endpoint: An Endpoint is used to designate a communication address in PAMI. Addressing is not based on processes or tasks but rather on Endpoints within the process. This can be used to provide finer grain addressing within a process that allows different threads to be pinned or attached to specific endpoints, thereby providing communication across different threads. Hybrid programming models and the hybrid proposals for MPI-3.0 would directly benefit from this approach.

C. Exploiting Communication Threads

Communication threads are helper threads that perform background "advance" on one or more PAMI contexts, thus providing communication parallelism in applications. They also drive the MPI progress engine. These threads are designed to be automatic in nature, such that when an application thread is spawned (or ready to run) on the same hardware thread, the commthread will voluntarily yield (change to lowest priority) and allow the application thread to run. During execution, commthreads detect the condition where no communications are going on and will execute a PPC wait instruction using the Wakeup Unit, thus eliminating any impact on other compute threads. This will ensure that compute threads get full access to the node resources when no communications are happening.

A great advantage of communication threads is that they allow for communication/computation overlap. Figure 2 demonstrates the usage of communication threads. When the main thread reaches a data movement or communication phase in the application, it can generate a work request (addressed to a particular context) and post it to the lock-less work queue, which resides in a "wake up" region/unit of shared memory. The work request can include processing memory FIFO packets, moving data within a node, or performing reduction arithmetic. A pool of communication threads would be in a "wake up wait" state, which is a special instruction that halts processing for a particular hardware thread. The instruction causes the thread to consume no power nor generate heat while in that state. The wake up unit is programmed to monitor the specific wake up region containing the work queue. Once a work request is posted to the queue, the threads

wake up from the wait state and resume processing. The thread that owns the context addressed by the work request will call the advance routine which basically dequeues the work request, performs the actual work specified in it, and then invokes the call back function, which sets the completion conditions for the main thread that posted the work. If there are no more incoming work requests addressed to this particular thread, it goes back to the wake up wait state. Note that all of these events are happening in the background while the main thread is doing useful computation. At some point, the main thread will poll to see if the work request has been satisfied

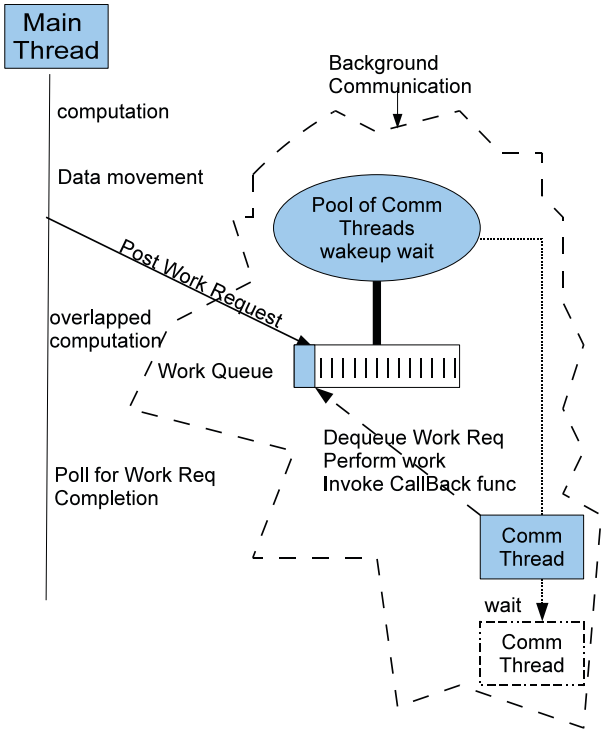

Figure 2. Exploiting communication threads

D. Collective Acceleration

In BG/Q, collective operations such as barrier, broadcast, reduce, and allreduce are directly supported by the collective network embedded in the 5D torus. As explained earlier, classroutes provide the routing tree information for the packets to travel up the tree to the root and then down tree to all participating nodes of the collective. In addition, the classroutes can be programmed to work on sub communicators which are contiguous rectangles (e.g. lines, planes or cubes).

On BG/Q, the collective operations are RDMA capable and the data that is being operated upon is directly read from or written to the memory. This eliminates extra copies, improving the performance of the operation. Moreover, together with the shared memory, integrated protocols can be designed to support effcient messaging when more than one process is running on a node, as explained in the next section.

Classroutes are a limited resource on BG/Q, so MPI applications that use a large number of communicators will not be able to use the Collective Network for all of them, even if all are rectangular. PAMI supports the ability to "optimize" and "deoptimize" a communicator for the Collective Network, such that an active set of communicators can access and reuse available classroutes. This feature is exported to MPI users via MPIX extensions.

E. MU Device Software

As mentioned previously, BG/Q has a sufficient number of hardware resources to enable concurrent communication per thread. The 544 injection FIFOs and 272 reception FIFOs are partitioned across the PAMI contexts with each context having exclusive access to its own set of resources, thereby eliminating any need for locking and critical section protection. The MU software creates independent entities for each context that have an independent set of protocol data structures and addressing mechanisms. Messages are sent using the various protocols associated with these contexts, the important ones being the *eager* protocol for short messages via memory FIFOs and *rendezvous* for long messages. Eager protocol has lower latency since it does not have a handshake phase. However, it has lower throughput as the message payload must be copied from the memory FIFO to the application buffer. In the rendezvous protocol, remote get is used to directly transfer the data from the source node to the destination node's buffer. The progress for these protocols that involves posting/building descriptors and polling for the incoming messages is done on a per-context basis. Also, to maintain MPI ordering, injection FIFOs are pinned statically for each destination so that the same FIFO is used every time for a given destination. Eager messages and rendezvous headers use deterministic ordered routing to be matched with receives on the destinations in order. We have explored optimized algorithms for active message send as well as one-sided put and get over the MU hardware.

F. Shared memory software

With multiple processes per node and several applications exhibiting locality characteristics, shared memory based communication has been a popular method to extract good performance within the node. In addition, using shared memory in BG/Q enables applications to take advantage of the high throughput available in the L2 cache for intra-node communication.

The shared memory software uses lockless queues that take advantage of L2 atomic increment instructions. To enable good memory scaling, each process owns only one queue to

which others atomically write into. It is important to note that we use the wakeup unit for advancing both the MU and shared memory communication paths. This eliminates the need for a thread to actively poll for the message to arrive, therefore saving processor cycles.

G. Memory optimizations

To reduce the memory requirements, we've developed space efficient *topology* structures in the PAMI library to handle a range of ranks and importantly defined an *axial topology* which defines the range of the ranks emanating from a given node. These are used both for COMM_WORLD and sub communicators.

IV. MPI OVER PAMI

We extended MPICH2 [11] from Argonne National Laboratory with a pamid device that implements the MPICH2 ADI [12] and makes PAMI API calls for both point-to-point and collective communication. PAMI provides low-level point-to-point protocols for messaging across the different endpoints of different source and destination processes or tasks. These protocols are context-scoped and progress is made by advancing each individual context by the application thread or the commthread. The protocols are active message based and a dispatch is triggered on the remote endpoint upon message arrival. The dispatch function in the pamid device looks up the list of posted receives and if a match is found, it returns a buffer to the PAMI library to receive the message. If a match is not found, an entry is created in the unexpected queue, and a buffer is allocated to receive the message. Inter-node messaging uses the MU, while L2 Atomic based shared memory queues are used for intra-node communication.

A. Multi threaded MPI over PAMI

By default, in MPICH2, each call has a global lock to protect access to shared resources such as receive queues, request allocators, and network resources. Such an implementation is thread safe, but has limited scalability due to the global lock. We explored fine grained locking and lockless techniques in MPICH2 [13,16]. We extended request allocators by creating thread private pools to minimize locking overheads. We also leveraged parallelism from PAMI contexts to hand off the work in MPI_Isends to build and inject MU descriptors to a communication thread. The source PAMI context is computed by hashing the destination rank and communicator id, and the destination context on the remote node is computed by hashing the source MPI rank and communicator id. Thus, all the messages between two processes use the same source and destination contexts for a given communicator. This preserves MPI ordering for the messages as PAMI only orders messages between endpoints. Since the destination ranks can be hashed to different contexts, concurrency is available to the messages sent to different remote destinations or using different communicators.

Parallelizing the MPI_Irecv call is trickier. The default MPICH2 receive queue is serial and needs to be protected. We have explored parallel receive queues with a separate queue for a subset of source nodes. However, a wildcard any-source receive can serialize receive processing as it must be matched before all receives posted after that wildcard. We observed that the algorithms to process wild cards can be very complex and significantly limit performance in the presence of wild cards. As wildcard receives are very commonly used in Blue Gene/Q applications, we used the default MPICH2 receive queue algorithm with a low overhead L2 atomic mutex to serialize access to it. The remainder of receive processing, such as the processing of incoming packets and copying packet payload to user buffers, is parallelized on different communication threads.

The biggest challenge was to optimize the MPI_Waitall operation. The MPI_Waitall is executed on the main thread that polls completion counters in request objects completed by communication threads. There may be cache thrashing between the main thread and communication thread while accessing request objects, resulting in poor performance. We designed a two phase waitall algorithm. In the first phase, the MPI request IDs are converted to MPICH2 request object pointers via a hash function. The execution of this step takes tens of processor cycles per request to complete. We overlap the hash function computation with the load of another request's completion counter that is likely to be a cache miss. If the requests have not completed they are inserted into a queue and polled for completion in the second phase. As applications typically post several send and receive several messages before calling waitall, the two phase approach enables the request hashing overheads to overlap with the cache misses of already completed requests.

We use the thread level in the MPI_Init_thread call to determine the level of thread parallelism required by the application. If MPI_THREAD_MULTIPLE is requested, communication threads are automatically enabled to speedup message rate. There is also an environment variable available for applications that do not generally use that threading model.

B. MPI Collectives over PAMI

MPI collectives such as MPI_Barrier, MPI_Bcast MPI_Reduce, and MPI_Allreduce directly use the hardware collective network to achieve superior scaling in terms of latency and bandwidth. Collective performance is further aided by the RDMA enabled collectives cutting down any extra copies, unlike the memory FIFO where the packets have to be transferred to the application buffers from FIFOs. Also, using the shared memory within each BG/Q node, collective operations can be accelerated when more than one process is running on a node. The RDMA and shared memory allow seamless integration of intra-node and inter-node protocols, boosting the performance of these collective operations. For MPI_Barrier, we use the fast L2 atomics and the global

interrupt network to provide very low-overhead barrier across the entire machine. Moreover, we utilize the shared address approach on a node to get the best performance for broadcast and allreduce when running with more than one process per node. In the following, we detail the shared address approach.

C. Shared Address Collectives over BG/Q

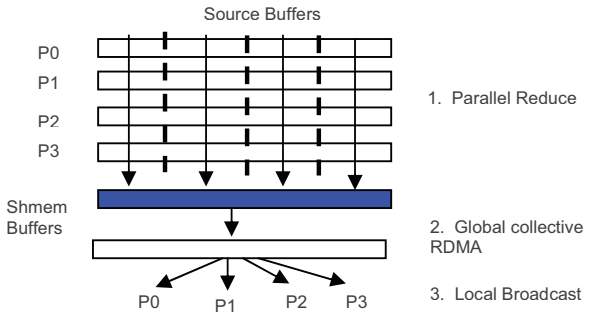

Figure 3. Short allreduce with parallelization

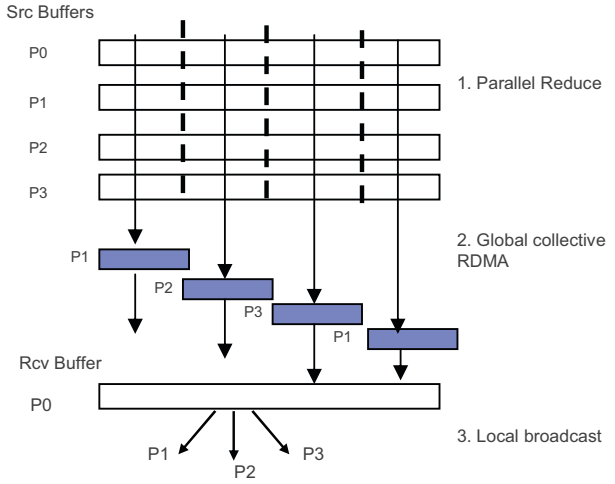

Figure 4. Long allreduce with parallelization

As copy costs dominate the intra-node performance of a collective with multiple processes per node, we deploy the "shared address" approach to eliminate any extraneous movement of data within the node. Using global addresses within the node, a process can read the data from its peers. This feature is very useful in implementing collectives such as MPI_Bcast and MPI_Allreduce. In MPI_Bcast, a master process from each node is designated to post RDMA descriptors to the collective network and the data directly arrives to its own buffer. Thereafter, other peers on the node can directly copy the data arrived using the global virtual address of the master.

For MPI_Allreduce, we would have an extra logical step of doing the local math within the node. Depending on the message size, we use two different approaches for performing math and pipelining with the network operations. For short messages (Figure 3), the basic idea is to parallelize the local math and inject a single network descriptor describing the entire local result obtained. Once again, all masters from all nodes are responsible for injecting descriptors and polling on the counters, checking for the arriving data. The network sum from the collective network arrives directly into the master's receive buffer as we use the RDMA write feature. The other peers wait for the master and copy the final result directly from the master's receive buffer. For large messages, we use pipelining across the local math, network allreduce, and local broadcast to get the best performance. To do this, each process operates on a slice of buffers as shown in Figure 4 and reports to the master after it is done. The master injects all the slices and the ordering of injection is maintained across all the masters of the nodes. The result is copied from the master's buffer in the same manner as described above.

V. PERFORMANCE ANAYLYSIS

Our performance study measures the performance of the PAMI and MPI libraries on production-level BG/Q hardware with 2048 nodes. We used micro benchmarks to measure latency and throughput of both point to point and collective communication as well as for measuring messaging rate. We present results from 1 to 16 MPI processes per node and 32 in some cases. Although 64 processes per node mode is supported, the current stage of the implementation focuses on functionality and is not optimized.

TABLE 1. PAMI half round trip for 0B message

	Single Threaded Latency
PAMI Send Immediate	1.18us
PAMI Send	1.32us

TABLE 2. MPI half round trip for 0B message

MPI Library	**Thread Mode**	**Comm. Thread Disabled**	**Comm. Thread Enabled**
Classic	Thread Single	1.95us	N/A
Classic	Thread Single	2.28us	8.7us
Thread Opt.	Thread Multiple	2.5us	N/A
Thread Opt.	Thread Multiple	2.96us	3.25us

Tables 1 and 2 show the half round trip latency for a zero byte message. The latency of the PAMI library is 1.18µ using the PAMI_SendImmediate call that copies application payload into an internal buffer and also sends the message if injection FIFO resources are available. This call is designed for short messages. The latency in the MPI library is 1.95us. MPI overheads are higher than PAMI, as MPI libraries must match receives with incoming packets. In addition, there are

overheads to construct request objects, hash functions to convert communicator and request identifiers to internal object pointers.

The classic MPI library has a global lock for all library calls. The thread-optimized library uses thread pools and lock-free techniques and acquires a mutex only while accessing a shared resource such as the receive queue. When the MPI library is initialized as MPI_THREAD_SINGLE, the classic library has the lowest overheads as the global locks are disabled. As the thread optimized library has memory synchronization calls to keep memory state consistent with the communication threads, it has higher overheads in MPI_THREAD_SINGLE.
As the classic library lacks fine grained locks, it must acquire the PAMI context locks to make progress on PAMI context resulting in higher latency in the presence of communication threads.

We ran a PAMI benchmark to measure the message rate of the PAMI library. Here each process on a reference node communicates with a peer process on a neighboring node. The neighboring nodes for the processes on the reference node are evenly distributed on the ten torus links out of a node. The performance results are presented in Figure 5. Observe we achieve 107 million messages per second with 32 processes per node. We also ran a modified Sequoia [14] message rate benchmark to measure the performance of the MPI classic library without communication threads. Figure 5 presents performance results where the processes on the reference node communicate with one neighbor process on a neighboring node. The maximum performance achieved is 22.9 million messages per second (MMPS) at 32 processes per node. Much of the difference between the PAMI and MPI message rates is due to overheads such as tag matching and other overheads in MPI processing.

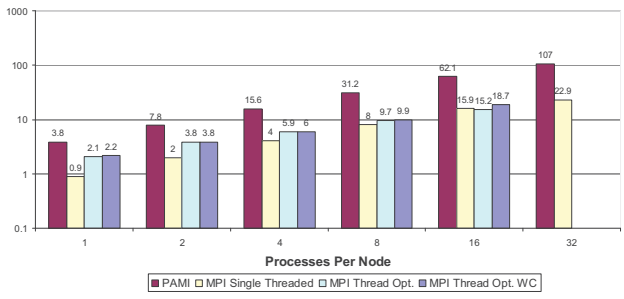

Figure 5. PAMI and MPI message rate (MMPS) on 32 nodes.

Figure 5 also shows the performance enhancements resulting from the techniques that use communication threads to accelerate message rate from 1 to 16 processes per node. Right now, we do not enable communication threads at 32 processes per node. Here, each process communicates with more than one neighbor process on different neighboring nodes. We also add a barrier after all MPI receives have been posted to eliminate unexpected messages. The barrier overhead is included in the message rate presented. We present the performance of both MPI_Irecv calls with source ranks and wild cards. We see a speedup of 2.4x for one process per node (PPN) where the most number of communication threads are available. With more processes per node, there are fewer communication threads per process resulting in lower speedups. The best performance of 18.7 MMPS is achieved when PPN=16 and communication threads are enabled.

TABLE 3. MPI neighbor send + receive throughput (MB/s) for 1MB message varying number of neighbors

Num. of Neighbors	MPI Eager	MPI Rendezvous
1	3267	3333
2	3360	6625
4	6676	13139
10	8467	32355

Table 3 presents the bi-directional nearest neighbor throughput for a reference node (with one MPI process) and an increasing number of neighbors up to 10, each on a different link. For rendezvous messages that use RDMA hardware capability we achieve 90% of peak network throughput. As eager messages are processed on the receiver by copying payload from packets in the memory FIFO to application buffers, the maximum achieved throughput is lower.

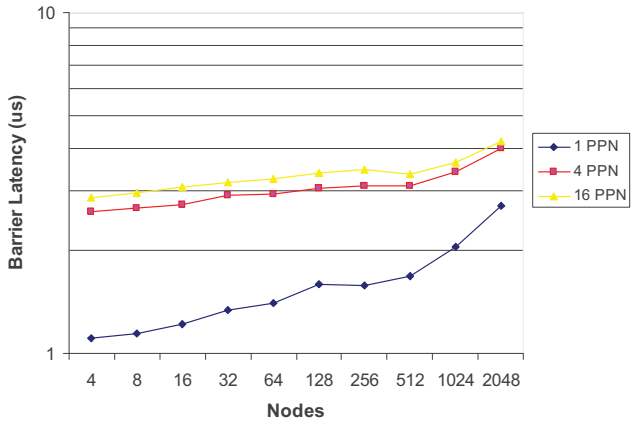

Figure 6. MPI Barrier Latency

The performance of the barrier collective via the global interrupt (GI) barrier network is presented in Figure 6. On 2048 nodes, MPI barrier latency is 2.7µs, 4.0µs and 4.2µs for PPN=1, 4, and 16 respectively. Observe barrier overhead in our MPI library is small even with 16 processes per node, as the local barrier is implemented via the scalable L2 atomic increment operation.

Performance of MPI_Allreduce double sum of a single double is presented in Figure 7. For PPN=1, 4, and 16, the respective latencies on 2048 nodes are 5.5µs, 5.0µs, and 5.3µs. Figure 8 shows the MPI_Allreduce throughput on 2048 nodes. We achieve a throughput of 1704MB/sec that corresponds to 95% of peak with PPN=1 for an 8MB allreduce. At PPN of 4 and

16, the MPI library achieves a peak throughput of 1693MB/s (94% of peak) for a 2MB allreduce and 1643 MB/s (91%) for a 512KB allreduce respectively. For larger messages, the send and receive buffers spill out of the L2 cache and must be read and stored to DDR respectively. So the performance of allreduce is driven by DDR throughput which is lower than the level-2 cache.

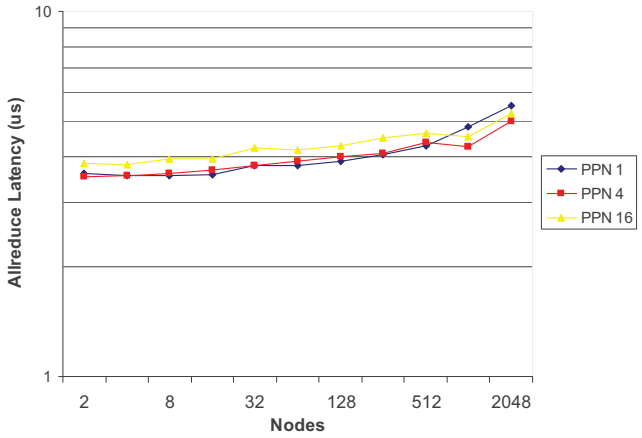

Figure 7. MPI Allreduce (MPI_DOUBLE, MPI_SUM) latency

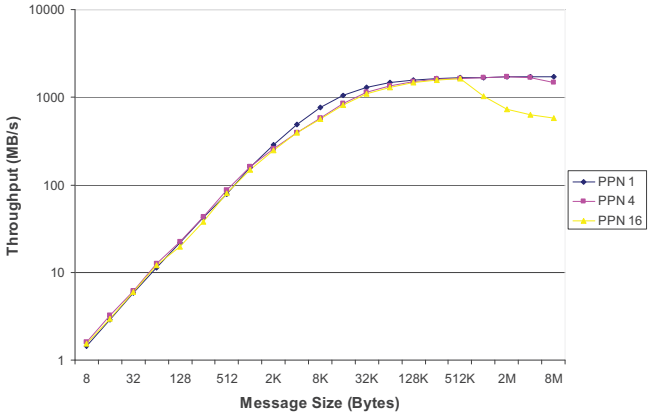

Figure 8. Allreduce throughput via collective network on 2048 nodes (MPI_DOUBLE, MPI_SUM)

The performance of collective network broadcast is presented in Figure 9. With PPN=1, we achieve a performance of 1728MB/sec which is close to 96% of hardware peak for a 32MB broadcast. At PPN=4, the best performance is 1722MB/s with a 4MB buffer size, while at PPN=16, we achieve a peak throughput of 1701MB/s for a 1MB buffer. The performance for large messages at PPN=4 and 16 saturates as the broadcast data spills out of the L2 cache and the performance is driven by DDR throughput.

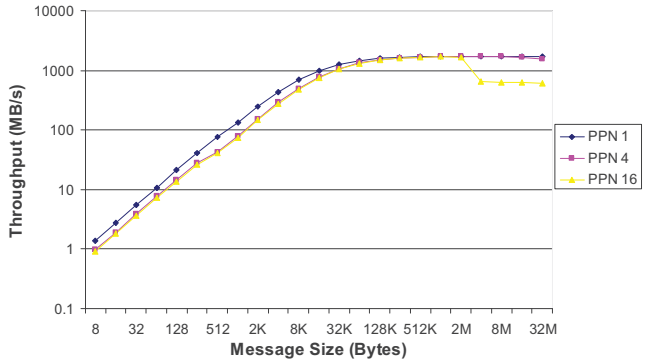

Figure 9. Broadcast throughput via collective network on 2048 nodes

To improve broadcast performance, by up to a factor of nearly 10, we also implemented a 10-color rectangle broadcast, where the root sends data to all the remaining nodes in the 5D torus via 10 edge disjoint spanning trees [15]. The peak throughput of this algorithm is 18 GB/s. Figure 10 shows the performance of the rectangle algorithm on 2048 nodes of BG/Q. With PPN=1, the maximum achieved throughput is 16.9 GB/sec, about 94% of peak network throughput. At four and sixteen processes per node, the incoming broadcast data has to be copied into four or sixteen buffers and this copy rate determines the throughput of the broadcast. Again, for large messages, the broadcast buffers spill out of the L2 cache and DDR throughput determines broadcast performance.

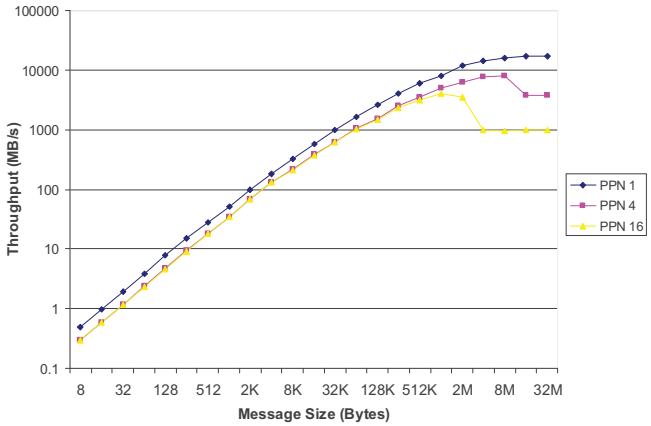

Figure 10. Broadcast throughput on 2048 nodes via the multi-color rectangle broadcast algorithm

VI. SUMMARY

We presented the thread optimized and highly scalable PAMI messaging library with performance results on the BG/Q machine. The MPI and PAMI libraries achieve message rates of 107 and 22.9 MMPS respectively via micro benchmarks. We exploit the wakeup unit and L2 atomic features of the

BG/Q compute node to accelerate MPI message rate via communication threads. We achieve a speedup of 2.4x for message rate with one process per node. Our collective performance results show under 4.2us latency for barrier and under 5.5us latency for allreduce double sum on 2048 nodes. We achieve high percent of peak for both nearest neighbor and collective throughput. The maximum broadcast throughput achieved is 16.9GB/sec. Due to L2 and DDR contention, collective throughput may decrease at 16 processes per node. This suggests that applications should be threaded and be run on fewer processes per node to get the best performance on this architecture. The collective network on BG/Q is enabled for both MPI_COMM_WORLD and rectangular sub communicators.

In the future we would like to explore performance optimizations for other collective operations such as all-to-all, scatter and gather. We would also like to explore new algorithms for irregular sub-communicators.

ACKNOWLEDGEMENTS

We would like to thank Robert M Senger, Yutaka Sugawara, Noel Eisley, and Martin Ohmacht on providing technical support for the Blue Gene/Q hardware design. We would like to thank Charles Archer and Pat Mccarthy for PAMI technical support. We would like to thank Bryan Rosenburg and Thomas Gooding for help and guidance with CNK. In addition, we would also like to thank Carl Obert, Robert Wisinewski and George Chiu. The work presented in this paper was funded in part by the US Government contract No. B554331.

REFERENCES

[1] The IBM Blue Gene Team. "The Blue Gene/Q Compute Chip," Presented at Hot Chips Conference http://www.hotchips.org/archives/hc23/HC23-papers/HC23.18.1-manycore/HC23.18.121.BlueGene-IBMBM_BQC_HC23_20110818.pdf, Aug. 17-19, 2011.

[2] D. Chen, N. A. Eisley, P. Heidelberger, R. M Senger, B. Steinmacher-Burrow, Y. Sugawara, S. Kumar, J. J. Parker, V. Salapura and D. L Satterfield. The Blue Gene/Q Interconnection Network. To appear in Proceedings of Supercomputing SC'11, Seattle Washington, 2011.

[3] A. Gara, M. A. Blumrich, D. Chen, G. L. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L System Architecture. IBM Journal of Research and Development, 49(2/3):195–212, 2005.

[4] IBM Blue Gene Team. Overview of the Blue Gene/P project. IBM J. Res. Dev., 52(1/2), January (2008).

[5] M. P. I. Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpiforum. org/docs/mpi-20-html/mpi2-report.html.

[6] T. El-Ghazawi, W. Carlson, and J. Draper. UPC specification, 2003. http://upc.gwu.edu/documentation.html

[7] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. Lecture Notes in Computer Science, 1586, 1999.

[8] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, Parallel Programming using C++, pages 175–213. MIT Press, 1996.

[9] S. Kumar, G. Dozsa, G. Almasi, D. Chen, P. Heidelberger, M. E. Giampapa, M. Blocksome, A. Faraj, J. J. Parker, J. Ratterman, B. Smith and C. J. Archer. The Deep Computing Messaging Framework. In Proceedings of International Conference on Supercomputing, Kos, Greece, 2008.

[10] M. Banikazemi, R. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. IEEE Transactions on Parallel and Distributed Systems, 12(10):10811093, 2001

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. MPICH: A high-performance, portable implementation of the mpi message passing interface standard. Parallel Computing, 22(6):789–828, September 1996.

[12] W. Gropp and E. Lusk. MPICH ADI Implementation Reference Manual, August 1995.

[13] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, Bronis R. de Supinski and R. Thakur. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In Proceedings of the 2010 IEEE International Conference on Cluster Computing.

[14] Sequoia Benchmark Codes. *https://asc.llnl.gov/sequoia/benchmarks/*

[15] S. Kumar et. al. Architecture of the Component Collective Messaging Interface. International Journal of High Performance Computing Applications, February 2010 vol. 24 no. 1 16-33.

[16] G. Dozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman and R. Thakur. Enabling Concurrent Multithreaded MPI Communications on Multicore Petascale Systems. In Proceedinds of the 17th European MPI Users's Group Meeting (Euro MPI 2010), September 2010.

[17] Myricom. Myrinet Express (MX): A hight performance, lowlevel, message-passing interface for Myrinet, July 2003. http://www.myri.com/scs/MX/doc/mx.pdf.

[18] Myrinet Software and Documentation Home Page. Myricom. Myricom: GM, MX, MPICH-GM, MPICH-MX and Sockets-GM. http: //www.myri.com/.

[19]Common Communication Interface http://www.olcf.ornl.gov/center-projects/common-communication-interface/

[20] D. Buntinas, G. Mercier, and W. Gropp, "Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI," in *International Conference on Parallel Processing*, 2006.

[21] M. Banikazemi, R. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. IEEE Transactions on Parallel and Distributed Systems, 12(10):1081–1093, 2001

[22] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. ACM International Conference on Computing Frontiers (CF), May 17-19, 2010, Bertinoro, Italy