RC25088 (W1012-066) December 16, 2010
Computer Science

IBM Research Report

Optimizing MPI Collectives Using Efficient Intra-node Communication Techniques over the Blue Gene/P Supercomputer

Amith Mamidala[1], Ahmad Faraj[2], Sameer Kumar[1], Douglas Miller[2], Michael Blocksome[2], Thomas Gooding[2], Philip Heidelberger[1], Gabor Dozsa[1]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[2]IBM Systems and Technology Group
Rochester, MN 55901

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Optimizing MPI Collectives using Efficient Intra-node Communication Techniques over the Blue Gene/P Supercomputer

Amith Mamidala[1] Ahmad Faraj[2] Sameer Kumar[1] Douglas Miller[2]
Michael Blocksome[2] Thomas Gooding[2] Philip Heidelberger[1] Gabor Dozsa[1]
{amithr, faraja, sameerk, dougmill, blocksom, tgooding, philiph, gdozsa}@us.ibm.com
[1] IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598
[2] IBM Systems and Technology Group, Rochester, MN, 55901

Abstract

The Blue Gene/P (BG/P) system is the second generation in the line of massively large supercomputers that IBM built with a petaflop scalability footprint and with greater emphasis on maximizing the efficiency in the areas of power, cooling, and space consumption. The system consists of thousands of compute nodes interconnected by multiple networks, of which a 3D torus–equipped with direct memory access (DMA) engine–is the primary. BG/P also features a collective network which supports hardware accelerated collective operations such as broadcast, allreduce etc. BG/P nodes consist of four cache coherent symmetric multi-processor (SMP) cores. The message passing interface (MPI) is the popular method of programming parallel applications on these large supercomputers. One of BG/P's operating modes is the quad mode where the four cores can be active MPI tasks, performing inter-node and intra-node communication.

In this paper, we propose software techniques to enhance MPI collective communication primitives, MPI_Bcast and MPI_Allreduce in BG/P quad mode by using cache coherent memory subsystem as the communication method inside the node. Specifically, we propose techniques utilizing shared address space wherein a process can access the peer's memory by specialized system calls. Apart from cutting down the copy costs, such techniques allow for designing light weight synchronizing structures in software such as message counters. These counters are used to effectively pipeline data across the network and intra-node interfaces. Further, the shared address capability allows for easy means of core specialization where the different tasks in a collective operation can be delegated to specific cores. This is critical for efficiently using the hardware collective network on BG/P as different cores are needed for injection and reception of data from the network and for copy operation within the node. We also propose a concurrent data structure, Broadcast (Bcast) FIFO which is designed using atomic operations such as Fetch and Increment. We demonstrate the utility and benefits of these mechanisms using benchmarks which measure the performance of MPI_Bcast and MPI_Allreduce. Our optimization provides up to 2.9 times improvement for MPI_Bcast over the current approaches on the 3D torus. Further, we see improvements up to 44% and 33% for MPI_Bcast using the collective tree and MPI_Allreduce over the 3D Torus, respectively.

I. Introduction

The Blue Gene/P (BG/P) [1] Supercomputer allows for ultra scalability and is designed to scale to at least 3.56 petaflops of peak performance coupled with superior efficiencies in the areas of power, cooling and space consumption. BG/P consists of thousands of compute nodes and each of these nodes is composed of four processing cores. Unlike its predecessor, all the processing cores are arranged as a SMP with hardware managed cached coherency. The compute nodes are interconnected with three different types of networks with the 3D torus being the principal network for data communication. There is also the collective network which supports hardware accelerated collective operations. The primary mode of communication over BG/P is by exchanging messages via the standard Message Passing Interface (MPI) which has become ubiquitous in the high performance computing arena. One important and widely used mode of running the parallel applications over BG/P is to program all the processing cores to do message passing. In this context, it is important to not only consider the inter-node communication but also optimize intra-node communication performance in conjunction with the network communication.

MPI provides a wide variety of communication primitives. In particular, it provides for a rich set of collective operations which are extensively used in many scientific applications. In this paper, we focus on optimizing two widely used collective operations namely, Broadcast and Allreduce by allowing sharing of data via the cache coherent memory subsystem [1]. This is an essential step on BG/P as the DMA engine, though capable of saturating all the six torus links simultaneously [2], does not provide adequate bandwidth for the collective messaging tasks. Also, the processes communicating via memory can either use a separate mutually shared segment or directly access the memory of the peer's process. We refer to these approaches in this paper as the shared memory and shared address space based methods respectively.

Shared memory based methods for these collectives have been extensively studied by many researchers in the past [3], [4], [5], [6], [7], [8]. In these approaches either a flat or a circular buffer is deployed for staging the data. Data has to be either copied in or out of these buffers during the course of the operation. Synchronization is handled explicitly by setting or reading signaling flags from a shared area. These flags indicate whether a particular data item is either read or written into the data buffers. However, with the rising number of cores per node, effective synchronization techniques coupled

with shared data structures is needed to ensure safety and performance at the same time. This is because potentially several processes or threads can access these data buffers and providing convenient concurrent data structures is extremely important for programmability and performance. In this paper, we propose a concurrent Bcast FIFO for MPI_Bcast using atomic operations to enable safe enqueue and dequeue of data items. The FIFO can be designed on any platform supporting the fetch and increment atomic operation. However, mechanisms employing shared memory involve extra copying of the data from/into the staging buffers. On BG/P, where the processing cores are relatively slower, memory copy imposes a fundamental bottleneck in terms of achievable performance. Recently, several researchers have demonstrated the utility of accessing the peer's memory directly for improving intra-node MPI point-to-point and collective performance [9], [10], [11], [12], [13], [14]. These studies have concentrated mostly on boosting performance with in the node and do not take into account the network capabilities. BG/P features advanced hardware accelerated collective primitives both in torus and the collective tree network. However, efficiently leveraging these poses a challenge because:

a) data arriving via these channels has to processed fast enough, considering especially the lower clock frequency of the cores,

b) any extra copy across the network interface and intra-node buffers presents a bottleneck,

c) in the case of collective tree, the processing cores also have to inject/receive data into the hardware tree, further compounding the problem and

d) in the case of allreduce, the cores have to simultaneously do a local sum and broadcast of the data arriving from the network.

In the following sections of the paper, we propose algorithms that address these challenges. In particular, in this paper:

- we propose an integrated mechanism of doing shared address MPI_Bcast over BG/P Torus that utilizes light weight message counters for data synchronization and pipelining.
- we design a shared address and core specialization strategy that effectively leverages the collective network providing hardware broadcast capabilities. The key idea used in the approach is to delegate special tasks to each of the core and allow these tasks to be completed in a pipelined and asynchronous manner wherever possible.
- we develop a similar approach for MPI_Allreduce where different cores are used to perform local reduction and broadcast. A separate core handles the network protocol processing.
- we design software Bcast FIFO, a concurrent data structure which provides a convenient and general mechanism of broadcasting data over any system supporting a basic atomic fetch and increment operation.

The design of the schemes is integrated into the DCMF messaging stack [15], CCMI collective framework [16] and glued to MPICH [17]. We demonstrate the utility and benefits of these mechanisms using benchmarks measuring the performance of MPI_Bcast and MPI_Allreduce. Our optimizations provide up to 2.9 times improvement for MPI_Bcast in the quad mode over the current approaches on 3D torus. Further, we see improvements up to 44% and 33% for MPI_Bcast using the collective tree and MPI_Allreduce over the 3D torus, respectively. Our results also demonstrate the effectiveness of the software message counters to aid efficient pipelining.

In the following section of the paper, we provide the motivation of our work followed by the detailed background. We then describe the details of the collective algorithms for MPI_Bcast and MPI_Allreduce over the BG/P 3D torus and collective network. Finally, we evaluate the techniques proposed in the paper followed by the conclusion and future work.

II. Motivation

In this section, we explain the main motivation for the different designs proposed in the paper.

As discussed in the earlier section, copying costs play a decisive role in the performance of the collective operations. This poses a challenge on BG/P where due to power and efficiency considerations, each of the core is clocked at a lower frequency. It is to be noted that the same guiding principle also applies in the context of multicore computing where increasing parallelism and not the core frequency is required to obtain performance with low power dissipation. It is in this context that the shared address schemes play an important role. By shared address schemes we mean the mechanisms by which a process can directly access the memory of the peer process residing on the same node. This would involve, directly or indirectly the operating system services on the node. Recently, several researchers have shown the effectiveness and applicability of the concept on different architectures. Jin et al. demonstrated the kernel aided one copy schemes using LiMIC [11], a kernel module supporting MPI intra-node communication on a Linux machine, integrated into open source MVAPICH [18], [19], [20]. Brightwell et al. have shown the performance gains on the Cray XT using the SMARTMAP [9], [10], [21] strategy developed in the Catamount lightweight kernel. The performance results were shown using open source OpenMPI [22] over Cray.

These schemes were also illustrated by using Kaput [23], another kernel module in [13]. However, all these studies concentrated on the intra-node communication. There are several key advantages that shared address capability provides for designing collectives over massively parallel supercomputers which is elucidated in the following parts of the section. It is to be noted that most network hardware including BG/P, feature DMA engines that can move data across the processes. As we demonstrate, shared address capability is a useful communication method in such scenarios as well.

Some of the salient benefits of shared address methods are the following:

- Avoid extraneous copy costs thereby pushing the performance envelope of the collective algorithms
- Allow lightweight synchronization structures such as counters to effectively pipeline across the different stages of the collective : between network and shared memory and across different stages in the shared memory
- Avoid explicit global flow control across network and intra-node interfaces. Since the destination and source buffers are the application buffers, data is channelled directly in and out of these buffers. Avoiding staging buffers automatically solves the issue of explicit flow control mechanisms. However, care must be taken to pin the buffers in the memory during the operation. In

BG/P, by default all the application memory is always pinned in the memory.
- Allow easy means of core specialization where certain tasks can be delegated to one or more cores increasing the performance of the collective algorithms. As demonstrated in the following sections of the paper, this technique is critical to the performance of broadcast over the hardware collective network.
- in the case of allreduce, the cores are specialized to simultaneously do a local sum and broadcast the data arriving from the network. A dedicated core performs allreduce protocol processing over the torus network.

However, it must be noted that leveraging the shared address capability is dependent on the interfaces exposed by the operating system. On systems which do not support this functionality we propose a convenient and efficient data structure, Bcast FIFO for the broadcast operation. The data structure uses a basic fetch and increment atomic operation to support safe message enqueue and dequeue operations.

In the following sections of the paper, we propose different algorithms and communication mechanisms that leverage the benefits of shared address space and atomic operations. Please note that though we have conducted our studies over BG/P, the general idea can be easily applied to other networks as well. For example, InfiniBand [24] allows for RDMA mechanisms where the data can be directly placed in the application buffer. Several researchers [25], [26], [27] have observed good performance with these techniques. The ideas mentioned in this paper can be easily integrated with the RDMA mechanism of InfiniBand to provide efficient collectives.

III. Background

Overview of BlueGene/P : A BG/P compute node consists of 4 PowerPC 450 SMP cores embedded on a single ASIC, running at a clock frequency of 850MHz, and having access to 4GB of memory. There are three operating modes: symmetric multi-processing (SMP) mode where one process with up to four threads can be active; dual (DUAL) mode where two processes with up to two threads each can be active; and quad mode (QUAD) mode where all four processes are active and each gets 512MB of memory. In this paper, we focus on the quad mode and describe optimized collectives when four processes are launched per node.

BG/P comprises of three different interconnection networks: 3D torus, collective and global interrupt network connecting all the compute nodes. We focus on the 3D torus and collective network in this paper.

A. BG/P interconnection networks

3D Torus: The 3D torus is the primary network used in BG/P. It is dead-lock free and supports reliable packet delivery. Each node in the torus connects to six neighbors with links of 425MB/s raw throughput. It also provides for hardware accelerated broadcast wherein a deposit-bit can be set in the packet header allowing torus packets to be copied at intermediate nodes along the way to the destination (on torus line). This feature is extensively used in the collective algorithms over BG/P.

Collective Network: The collective network has a tree topology and supports reliable data movement at a raw throughput of 850MB/s. The hardware is capable of routing packets upward to the root or downward to the leaves, and it has an integer arithmetic logic unit (ALU). This makes it very efficient for broadcast and reduction operations. Note that there is no DMA on this network. Packet injection and reception on the collective network is handled by a processor core [15].

DMA Engine: BG/P improves upon BG/L hardware by adding a DMA engine that is responsible for injecting and receiving packets on the torus network and for local intra-node memory copies. The high performance DMA engine can also keep all six links busy, resulting in better performance of torus point-to-point and collective communication [2], [16], [15].

Direct Put/Get: The DMA engine allows the capabilities of direct put and get operations of message data to and from a destination buffer. In this mechanism, the host posts a descriptor to the DMA with the description of the source and destination buffers. Counters are also allocated to track the progress of the operation which are regularly polled by the processes cores. For every chunk of data read or written, the DMA would appropriately decrement the counter by the number of bytes transferred in the chunk. Together with the torus line broadcast capability, Direct Put allows for zero-copy collectives wherein the processor is not involved in any memory copy operations. Please note that such RDMA techniques have been studied over other networks such as InfiniBand in [27], [7]. However, in both these approaches, the data is channelled to a staging shared memory buffer and not to the application buffer directly.

B. Compute Node Kernel (CNK) on BG/P

CNK is a simple linux like kernel and runs one application at a time. However, an application can consist of up to four processes per node. In our case it would be four MPI tasks per node. It is a light weight operating system using small amount of memory to run and leaving the rest to the application. A very useful feature in CNK is the process window support. By using specialized system calls, the host kernel on BG/P allows for a process to expose its memory to another process. Using this mechanism a process can directly read/write the data from/to the source buffers of other process during the message transfer operations. This could potentially cut-down any extraneous copy costs that would be incurred otherwise. The exact mechanism is described in detail below.

Consider two processes A and B where process A wants to read from the virtual address VA_b of process B a length of n bytes. In order to perform this operation, process B translates VA_b into a physical address PA_b using a system call. Process A uses PA_b and makes another system call to map the physical memory region into its address space. The system call returns the new VA_a for this memory region. Each process is provided N Translation Look-aside Buffer (TLB) slots. By default, the value of N is three. Hence, mapping only a maximum of three memory regions is allowed in this context. This implies that there is one mapping allowed for each of the peer processes on the four core BG/P node. The sizes of the TLB slots can be configured ranging from 1MB, 16MB to 256MB. Using the largest TLB slot is advantageous if the application buffers used in messaging lie within 256M span of memory. Also, in the worst case, more than one mapping may be required to access one buffer if the buffer spans across multiple page boundaries. Adjustment to the

processor's TLBs must be done in privileged mode. In order to accomplish this, CNK exposes this functionality via a process window system call. Also, a traditional operating system would simply perform TLB faults to map the pages when they are accessed. However, the CNK approach is to map all TLBs needed for the application. CNK can take advantage of large, non-uniform TLB sizes to algorithmically determine a static best-TLB fit in the constrained 64 TLB slots of each PowerPC 450. CNK does this to avoid performance jitter in large scale applications. The aforementioned static TLB algorithm reserves N TLB slots for the usage of process windows. The value of N can be changed at the application load time.

IV. Communication Mechanisms

In this section, we first explain the mechanism of the Bcast FIFO. We describe the operation of the Point-to-Point FIFO using atomic Fetch and Increment operation which forms the basis of the design of the Bcast FIFO explained subsequently. We then describe the message counter mechanism.

A. Point-to-Point FIFO

By a Point-to-Point FIFO, we mean a process enqueues a data item into a shared FIFO and only one process dequeues that particular data item. The basic idea in the design of this FIFO is for the memory to be structured in the form of a shared FIFO where a first arriving process reserves a slot in the FIFO followed by the next process and so on. The required attributes of this FIFO are the following: a) Each process enqueues into a unique slot reserved by it. No two processes obtain the same slot in the FIFO. b) Messages are drained in the same order as they were enqueued in. The order of enqueueing is determined by the order of the reservations of the slots.

There are multiple ways in which a unique slot can be reserved by a process. One of the ways would be to use a mutex for the FIFO and obtain a unique slot. For example, each FIFO would have a counter associated with it. A process would increment this counter to obtain a unique slot id. A mutex would guarantee that accesses to this shared variable are serialized. However, one would incur the overhead of lock/unlock for every enqueue operation. Several researches have studied the benefits of using atomic operations for designing lock-free queues. Darius et al. [12] have demonstrated the effectiveness of compare and swap atomic operation to link the different data cells to implement a queue abstraction. The basic approach used in our technique is to use a fetch and increment operation which greatly simplifies the handling of the different queue elements. As shown in the figure 1, enqueueing a data element is accomplished by atomically incrementing the Tail and reserving a unique slot. However, the final location in the FIFO is determined by doing a (mySlot%fifoSize). The atomicity of the counter ensures that no two processes write to the same location in the FIFO. Also, before enqueueing the FIFO must be checked for free slots. The dequeue operation is handled in a similar fashion by incrementing the value of the Head. The particular item is accessed by doing a (Head%fifoSize) for obtaining the index into the FIFO once it is determined that valid data exists at the location pointed to by the Head.

B. Bcast FIFO

This FIFO is similar to the Pt-to-Pt FIFO for enqueueing the message. As shown in the figure 1, a given process increments the Tail atomically reserving a unique slot in the FIFO. Only if there is space it proceeds to write data into the FIFO. The amount of space is determined by checking if (mySlot-Head)$<$ fifoSize. If the condition is true, the data element is enqueued. Also, in addition, an atomic counter variable for this index is set to equal to (n-1) which is the count of all the processes that would dequeue this data element. The enqueue operation is complete when the process finishes the write completion step. The FIFO differs in the way the message is dequeued. Except for the process inserting a message into the FIFO, all the others need to read the message in order for it to be dequeued from the FIFO. The current index of the head is obtained by doing (Head%fifoSize) and after the process reads the message it also decrements the atomic counter which was initialized to (n-1) before. After this value reaches zero, the last arriving process completes the dequeue operation and the message is effectively removed from the FIFO by atomically incrementing the Head.

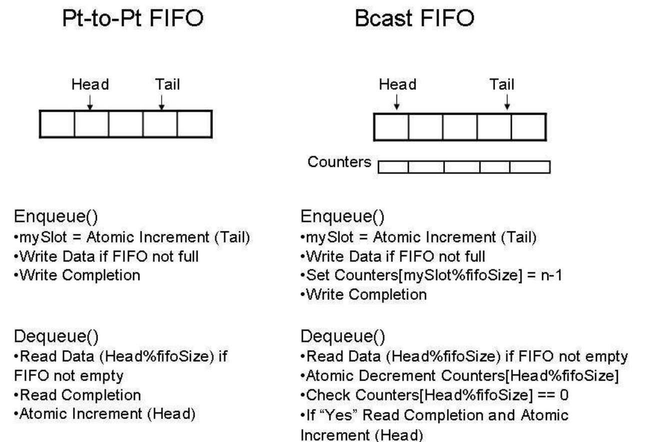

Fig. 1: Bcast FIFO

C. Message counters with direct copy

Message Counters are convenient and effective method of tracking the progress of data transfer operations. As explained in the earlier section, BG/P extensively uses the counter mechanism to monitor the status of different network operations. We have explored this technique at the software level for doing a direct copy of data in the broadcast operation. The central idea adopted in our approach is to dedicate a counter for a given broadcast and whenever the data arrives in the buffer, it is incremented by the total number of bytes received in the buffer. The detailed logic for this operation is presented in the next section.

V. Communication Algorithms

In this section, we present the collective algorithms that address the issues raised in the earlier sections of the

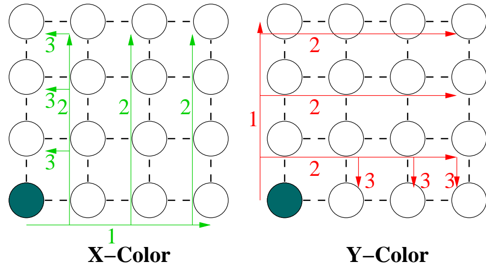

Fig. 2: Multi-color rectangle algorithm

paper. It is to be noted that depending on the message size, either the Torus or the Collective network based algorithms perform optimally. The Torus network is superior for large message collectives where the six torus links together provide more bandwidth than the Collective network. The Collective network is optimal for short to medium messages where the latency dominates the performance model.

A. Integrated Broadcast over Torus

We first describe the current algorithm that is used in doing large message broadcast over Torus. We then explain the integration of the intra-node communication mechanisms described in this paper to provide an efficient method of doing a broadcast operation.

1) Current Approaches:: The collective algorithms on BG/P are designed in a manner to keep all the links busy to extract maximum performance. This is accomplished by assigning unique connection ids to each of the links and scheduling the data movement on each connection. Specifically, these are referred to as the multi-color algorithms.

BG/P messaging stack uses *multi-color spanning tree* algorithms for broadcast on the torus network. These algorithms take advantage of the three or six edge-disjoint routes from the root of the collective to all other nodes in 3D meshes and tori, respectively. figure 2 illustrates a multi-color rectangle broadcast on a 2D mesh. The root is at the bottom left corner. On a 2D mesh, all root nodes have two edge disjoint spanning trees to all other nodes. In phase 1 of the X-color broadcast, the root sends data first along the X dimension using the deposit bit feature. The X-neighbors of the root then forward this data along the Y dimension in phase 2. On BG/P 3D torus, there are six such spanning trees which correspond to a peak throughput of 2250 MB/s which is close to peak performance. For quad mode scenario, an extra fourth dimension is added to these multi-color spanning tree algorithm. This dimension corresponds to the data movement within the node. Also, note that DMA is involved in moving the data across the different phases. Though the DMA is capable of keeping all the six links busy of a 3D torus node, it is not enough to concurrently transfer the data within the node along with the network transfers.

2) Proposed Algorithm: The basic idea behind the algorithms explained below is to effectively extend the concept of connection explained above within a node. The mechanisms for allowing this is described below. We postulate the techniques for doing this. The first technique streamlines pipelining but does not avoid extra copying. The second technique allows for both.

Shared Memory Broadcast using Bcast FIFO:

In this design, the master process forwards the data to its peers using the Bcast FIFO, presented in the earlier section of the paper. The mechanism works as follows: once a chunk of data is received from the Torus network into the application buffer, the master process enqueues the data element into the Bcast FIFO using the interface methods mentioned in the last section of the paper. The data is packetized if it is more than the FIFO slot size. Apart from the actual data, metadata information associated with the data is also copied into the same FIFO slot. The metadata includes the number of data bytes copied into the slot and the connection id of the global broadcast flow. In this fashion broadcast streams from multiple connections can be multiplexed into the same FIFO.

Shared Address Broadcast using Message Counters:

The basic idea behind the technique proposed is to receive the broadcast data from the network in one of the processes' application data buffer. We designate this process as the master process. The master after receiving the network data notifies other processes about the arrival of data. The arrived data is copied out directly from the application buffer of the master process. This is possible by using the System Memory Map calls which enables a process to view the memory of the other process. In our case, this would be the master process exposing its memory region to all the other processes on the node.

This design relies on data coming in order into the application buffer. Thus, it is applicable only in the context of data flow following connection semantics. As shown in the figure 3 the basic idea in our approach is to use a counter that is visible to all the processes. This approach seamlessly integrates with the collective algorithm over the Torus Network and cuts down the extraneous copy costs. As described earlier the network DMA also uses a counter mechanism to track the number of bytes read/written using the Direct Get/Put strategies. In our approach, we extend this technique at the software level by mirroring the contents of the DMA counters into the shared counter variables indicated as S/W counters in the figure. Note that the buffers indicated in the figure are partitioned four way for data streams flowing from X+, X-, Y+ and Y- directions respectively.

The counter object consists primarily of two fields: a) Base address of the data buffer b) Total bytes written into the buffer. The master process initializes the counter with the base address of its buffer and also sets the total bytes written to zero. Once the master is notified by the network about the reception of the next chunk of bytes in the data stream, it increments the Total bytes by the same amount. The other processes poll the counter value and test whether it has been incremented. All these processes maintain a local counter which is used to compare against the counter value at the master. Once they observe an increment in the counter, they copy the arrived chunk of bytes from the master and increment their own local counters. There is also an atomic completion counter which is initialized to zero by the master. All the processes increment this counter after they finished copying the data from the master. Once this counter equals to n-1 where n is the total number of processes, the master can go ahead and start using his buffer. This method is more effective and convenient than the earlier technique of using Bcast FIFO. However, message counters are applicable only to contiguous data flows.

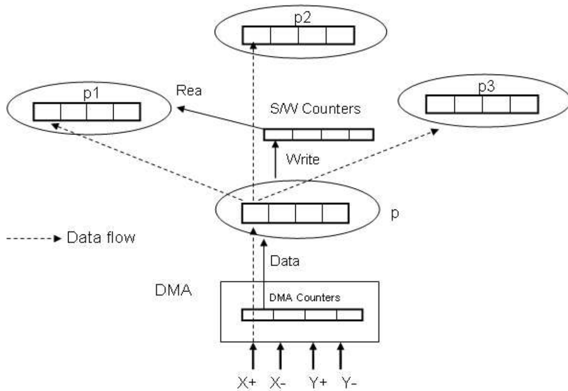

Fig. 3: Broadcast using Message Counters

B. Integrated Broadcast over Collective Network

In this section of the paper, we deal with small and medium message broadcast which uses the collective network. We first explain the current algorithms followed by the new algorithms proposed in the paper.

1) Current Approaches: The current algorithms use the fast hardware allreduce feature (math units) of the collective network. The root node injects data while other nodes inject zeros in a global OR operation. In SMP mode, two cores within a node are required to fully saturate the collective network throughput. Hence, two threads (the main application MPI thread and a helper communication thread) inject and receive the broadcast packets on the collective network. In QUAD mode, the DMA moves the data among the cores of each node. This can occur using the memory FIFO and direct put DMA schemes.

2) Proposed Algorithms: As discussed in the previous section, BG/P has a very efficient mechanism of broadcast using the tree hardware. Efficiently leveraging this tree requires attention to load balancing across the different tasks. In the following sections we describe the schemes to achieve this.

Shared Memory broadcast over Collective network: In this simple and basic design the data from the tree is transferred into a buffer shared across all the nodes. The same core accessing the collective network does both the injection and reception of the data. The received data is placed in a shared memory segment from where it is copied over by the other processes on the node. This optimization works for short messages where the copy cost is not a dominating factor in the performance of the collective operation.

Shared Address broadcast over Collective network: As discussed earlier, shared address capability cuts down the copy costs boosting the performance. Also, another benefit of this capability is that it allows easy means for the cores to specialize in certain tasks to extract the maximum possible performance from the underlying hardware.

As shown in the previous section, effectively utilizing the tree requires two independent tasks injecting and receiving data into and from the collective network respectively. The simple design described in the earlier section is not optimal for medium to large messages as the same master core does both the injection and reception, one after another. Stemming from the idea of using two threads to access collective network, a similar approach can be applied with two MPI processes. An injection process injects data into the collective network and a separate reception process copies the network output into the application buffer. However, distributing the data across all the processes in a node poses a problem. Directly using the Shared Memory techniques creates a scenario where either the injection or the reception process or both are loaded more than the other two processes as described below.

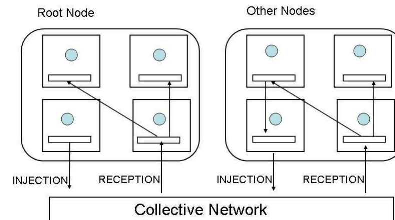

Fig. 4: Broadcast using Collective Network

Assume that the reception process receives data into a shared memory segment. This data can be copied over by the two idle cores. However, both the injection and the reception process have to simultaneously copy the data into their own application buffers as well. This slows down the injection and reception rate, significantly degrading the performance. Similar scenarios occurs where the reception process receives data directly into its own buffer. Since there is excess of memory bandwidth relative to the tree, the two idle cores can be delegated tasks in the collective operation. We demonstrate the utility of shared address mechanism to solve this problem.

Consider a system of N BG/P nodes where on each node, four MPI processes are launched with local ranks of zero to three. In the example figure 4 shown, N is equal to two. Assume that the broadcast operation is initiated by the global root whose local rank corresponds to 0. We designate all the processes with local rank zero from all the nodes as the injection processes. All the processes with local rank one would be the reception processes. However, unlike the Shared Memory approach, the data buffers involved in the operation are directly the application buffers. For example, the global root injects from its application buffers. All the local rank one processes receive the data directly into their final buffers. Once a chunk of data is copied into its application buffer, it notifies the other two processes with local ranks two and three. It uses a software shared counter mechanism described earlier. These two processes copy the data directly from the application buffer of process with local rank one. Further, the process with local rank two makes an additional copy into the application buffer of the injection process, which has the local rank of zero. The extra copy is not a problem as the memory bandwidth is at least twice that of the collective network. We demonstrate that this approach delivers the best performance in the performance analysis section of the paper.

C. Integrated Allreduce over Torus

In this final algorithm discussed in the paper, we propose a core specialization strategy to effectively use the network and memory resources to boost the performance of allreduce operation for large messages. Similar to the

earlier sections, we first describe the current algorithm followed by the new proposed method.

1) Current Approach: The basic idea in the algorithm used is to pipeline the reduction and broadcast phases of the allreduce. A ring algorithm is used in the reduction followed by the broadcast of the reduced data from the assigned root process. Similar to the broadcast algorithm, a multicolor scheme is used to select three edge-disjoint routes in the 3D torus both for reduction and broadcast. This scheme is not optimal as redundant copies of data are transferred by the DMA for the reduction operation. Also, the DMA cannot keep pace with both the inter- and intra-nod data transfers. As we demonstrate below, Shared address messaging overcomes this issue by delegating specialized tasks to different cores.

2) Proposed Algorithm: The allreduce operation can be decomposed into the following tasks: a) network allreduce b) local reduce and c) local broadcast. The data is first locally reduced followed by a global network reduction. The reduced data after arriving into the node is broadcasted locally. The central idea of the new approach is to delegate one core to do the network allreduce operation and the remaining three cores to do the local reduce and broadcast operation. Since there are three independent allreduce operations or three colors occurring at the same time, each of the three cores is delegated to handle one color each. The data buffers are uniformly split three way and each of the cores works on its partition. The exact mechanism is described below.

Assume that the pipeline unit used for reduction and broadcast used be Pwidth bytes. As soon as the operation starts, each of the core starts summing up the first Pwidth bytes from each of the four processes application buffers. All the application buffers are mapped using the system call interfaces, and no extra copy operations are necessary. The cores then inform the master core doing the network allreduce protocol via shared software message counters. The network protocol is exactly identical to its SMP counterpart where there is only one process per node. Once the network data arrives in the application receive buffer of the master core, it notifies the three cores. The other three cores start copying the data into their own respective buffers after they are done with reducing all the buffer partitions assigned to them.

VI. Performance Study

Our performance study is conducted on BG/P hardware on two racks in the quad mode equaling a total of 8192 processes. The study focuses on the following objectives.

- examine the performance of the different shared address collective algorithms developed for BG/P
- study the benefits of shared address collectives over collective network using core specialization
- demonstrate the benefits of shared address collectives over BG/P torus using message counters
- evaluate shared address allreduce over BG/P torus
- demonstrate the benefits of shared memory Bcast FIFO over BG/P torus

The micro-benchmark shown in Figure 5 is used to evaluate algorithms performance for MPI_broadcast on BG/P hardware with up to 16K nodes. A similar benchmark us used to measure the performance of MPI_Allreduce.

```
(1) elapsed_time = 0;
(2) for (i = 0; i < ITERS; i++)
(3)     MPI_Barrier(comm);
(4)     start = MPI_Wtime();
(5)     MPI_Bcast(...);
(6)     elapsed_time += (MPI_Wtime() - start);
(7) elapsed_time /= ITERS;
```

Fig. 5: Measuring performance of MPI_Bcast

A. Integrated Broadcast over Collective Network

In this section, we demonstrate the benefits of using shared address and shared memory algorithms for MPI_Bcast over the collective network which provides hardware accelerated broadcast support for MPI_COMM_WORLD. As discussed earlier, two main algorithms are used to improve the performance of broadcast in the QUAD mode of operation where four MPI processes are launched per node. In the first algorithm, referred to as the 'CollectiveNetwork + Shmem', a shared memory segment is used to transfer messages from the hardware to the four cores on the node. The second algorithm, referred to as the 'CollectiveNetwork + Shaddr', describes the shared address algorithm described in the section V. The first algorithm is for latency optimization and is used for short messages. As shown in the figure 6, it provides a 5.83 us latency for 8192 processes broadcast operation. It adds an overhead of 0.4 us over the hardware network broadcast, referred to as the 'CollectiveNetwork + SMP' for the intra-node protocol processing. The SMP mode gives the basic network latency as only one process is launched per node and it directly accesses the collective hardware. Also, the 'CollectiveNetwork + shmem' algorithm improves the performance considerably over the 'CollectiveNetwork + DMA' which uses the DMA to move data within the node after it is received over the collective network. The DMA moves the data to the memory FIFO of the core in this algorithm.

The second algorithm which is based on the shared address space and core specialization improves the performance of medium messages. As shown in the figure 7, the 'CollectiveNetwork+ Shaddr' algorithm outperforms over all the QUAD mode algorithms. The SMP mode algorithm, 'CollectiveNetwork + SMP' is described for reference. The other algorithms referred to in the figure are the 'CollectiveNetwork + DMA FIFO' and 'CollectiveNetwork + DMA Direct Put' which uses the DMA for moving data within the node. DMA can either transfer the data to the memory FIFO of the cores or directly place the data in the application memory using Direct Put primitives. The shared address scheme proposed in the paper improves the bandwidth throughput of medium messages up to 45% for message size of 128K bytes. Figure 8 shows the affect of system call overhead on the performance of the shared address schemes. Note that each mapping of an application buffer involves two system calls: to obtain the physical address from the virtual address and to map the physical address to the virtual address. If these system calls are invoked repeatedly by the application, it contributes to a big source of overhead. In our schemes, we internally cache the buffer information if the same buffer is repeatedly used in the application. Several open source software stacks follow such similar schemes, for e.g. MVAPICH [20] and OpenMPI [22] use the approach for MPI stacks over InfiniBand to avoid the overhead of

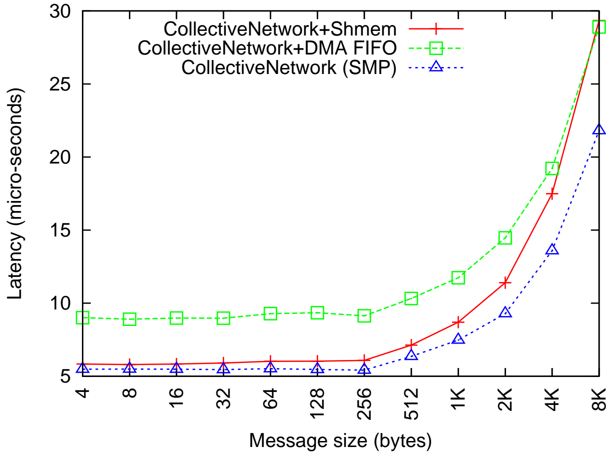

Fig. 6: Latency of MPI_Bcast

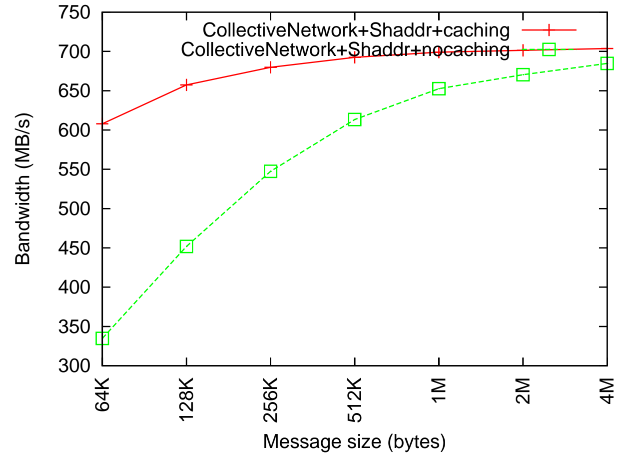

Fig. 8: Overhead of System Call Overhead

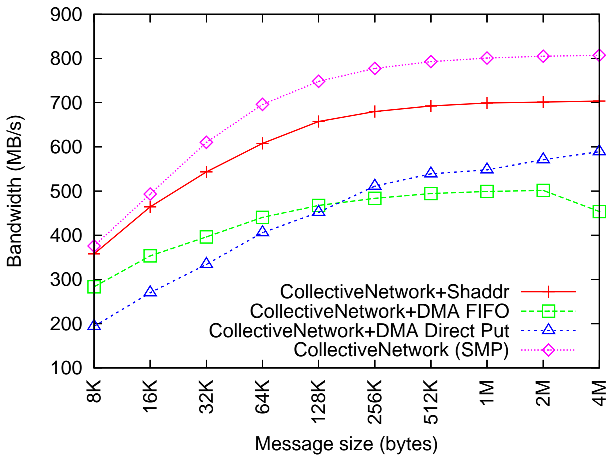

Fig. 7: Bandwidth of MPI_Bcast

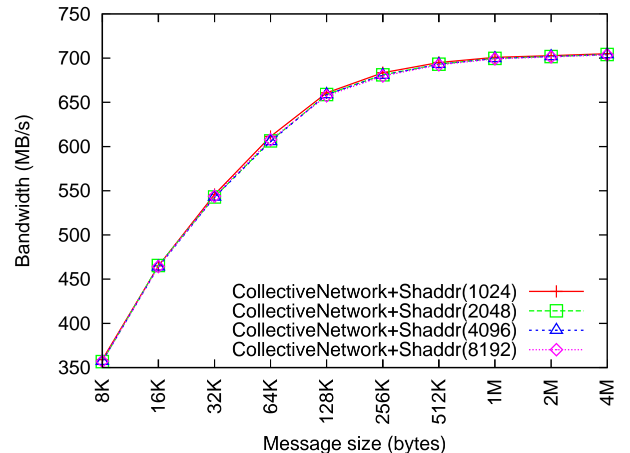

Fig. 9: Performance with increasing scale

pinning and unpinning the buffers. Finally, as shown in the figure 9, the algorithm scales well for different process configurations. This is owing to the fact that the collective network provides very good scalability and performance with increasing number of processes. A detailed study is published in [2].

B. Integrated Broadcast over Torus

In this section, we provide the details of the algorithms designed to do MPI_Bcast for medium to large messages. The first algorithm is referred to as 'Torus + Shaddr'. It is based on the shared address concept and described in section V. It uses the light weight message counters to effectively pipeline across network and intra-node. The second algorithm referred to as 'Torus + FIFO' uses the concurrent Bcast FIFO structure described in section IV. These algorithms are compared to the current best performing algorithm, 'Torus Direct Put' which involves the DMA to do the message transfer both within the node and as well as outside the node.

As shown in the figure 10, the 'Torus + FIFO' scheme improves the performance of broadcast by a factor of 1.4 for 2M message size. This is primarily because of concurrent data transfers intra-node by the processing cores and the DMA moving the data from the node to the Torus network. Please note that in this scheme a common FIFO is used to multiplex the data from all the six edge disjoint routes of data flow. As shown in the figure 10, the 'Torus + Shaddr' scheme performs the best for large messages. The algorithm is able to achieve a 2.9 speedup at 2M message size. The scheme also gives good improvement for messages at the lower end, 64K message size, and is within 15% of the peak possible for this message in the SMP mode. This is primarily because of the low overhead and light weight message counters which are used for synchronizing the data movement intra-node and over the torus. Also, note that the performance drops in the end for the 'Torus + Shaddr' schemes. This is due to the L2 cache size which is 8MB in size.

C. Integrated Allreduce over Torus

In this section, we describe the performance of the MPI_Allreduce algorithm described in section V. The algorithm uses three cores to exclusively do the reduction and broadcast of data. Each color or connection is

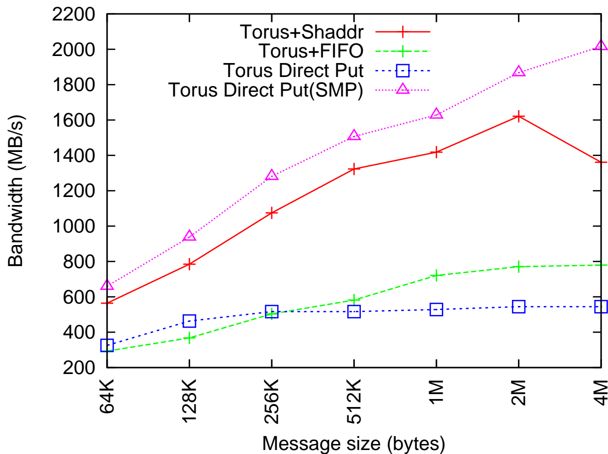

Fig. 10: Bandwidth of MPI_Bcast

assigned to a different core. There is one core dedicated to doing only the network protocol processing. As seen in the table I, we observe performance benefits across the different messages but the algorithm is mostly useful for large messages. As shown in the table, the algorithm provides about 33% improvement for 512K doubles.

TABLE I: Allreduce throughput

Doubles	New (MB/s)	Current (MB/s)
16K	145.35	120.7
32K	191.25	161.5
64K	215.9	196.35
128K	258.4	227.8
256K	289.0	234.6
512K	322.15	241.4

VII. Conclusions and Future Work

The Blue Gene/P Supercomputer (BG/P) is designed to scale to petaflops of peak performance while maximizing efficiencies in the areas of power, cooling and space consumption. BG/P consists of thousands of compute nodes interconnected by the primary network, 3-D Torus transporting data across the nodes. It also features a collective network providing efficient hardware collectives such as broadcast, allreduce etc. Each of the nodes is made up of four embedded cores, arranged as an SMP. The primary mode of running applications over BG/P is via the standard Message Passing Interface (MPI). It is possible in these applications for messages to be exchanged both inside and across the nodes, especially when all the cores are used for MPI tasks.

In this paper, we proposed software techniques to enhance MPI Collective communication primitives, MPI_Bcast and MPI_Allreduce in BG/P quad mode by using cache coherent memory subsystem as the communication method inside the node. The paper proposes techniques utilizing shared address space wherein a process can access the peer's memory by specialized system calls. Apart from cutting down the copy costs, such techniques allow for designing light weight synchronizing structures in software such as message counters. These counters are used to effectively pipeline data across the network and intra-node interfaces. Further, we demonstrate that the shared address capability allows for easy means of core specialization where the different tasks in a collective operation can be delegated to specific cores. This is critical for efficiently using the hardware collective network on BG/P as different cores are needed for injection and reception of data from the network and for copy operation within the node. We also proposed a concurrent data structure, Bcast FIFO which is designed using atomic operations such as Fetch and Increment. We have demonstrated the utility and benefits of all these mechanisms using benchmarks which measure the performance of MPI_Bcast and MPI_Allreduce. Our optimization provides up to 2.9 times improvement for MPI_Bcast in the quad mode over the 3D torus over the current approaches. Further, we show improvements up to 44% and 33% for MPI_Bcast using the collective tree and for MPI_Allreduce over the 3D Torus.

In our future work, we intend to extend the mechanism to other collectives such as MPI_Gather and MPI_Allgather which can also potentially move large volumes of data. Moreover, as shared address mechanisms gain prominence it becomes important to standardize the interfaces for effectively using these capabilities across different platforms and applications.

VIII. Acknowledgments

We would like to acknowledge Robert Wisniewski and Craig Stunkel for their useful comments and suggestions. Also, we want to thank the rest of the Blue Gene team. We would like to thank the government for providing the opportunity to work on this project.

References

[1] IBM Blue Gene Team, "Overview of the Blue Gene/P project," *IBM J. Res. Dev.*, vol. 52, January (2008). http://www.research.ibm.com/journal/rd/521/team.html.

[2] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations," in *IEEE Hot Interconnects*, 2009.

[3] R. Graham and G. Shipman, "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008.

[4] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[5] M.-S. Wu, R. Kendall, and K. Wright, "Optimizing collective communications on SMP clusters," in *International Conference on Parallel Processing*, 2005.

[6] A. Mamidala, R. Kumar, D. De, and D. K. Panda, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics," in *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, May 2008.

[7] A. R. Mamidala, A. Vishnu, and D. K. Panda, "Effi cient Shared Memory and RDMA based design for MPI Allgather over Infi niBand," in *Proceedings of Euro PVM/MPI*, 2006.

[8] A. Mamidala, L. Chai, H.-W. Jin, and D. K. Panda, "Effi cient SMP-Aware MPI-Level Broadcast over Infi niBand's Hardware Multicast," in *Communication Architecture for Clusters (CAC) Workshop*, 2006.

[9] R. Brightwell, "A Prototype Implementation of MPI for SMARTMAP," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008.

[10] R. Brightwell, T. Hudson, and K. Pedretti, "SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor," in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08), Austin, TX*, 2008.

[11] H.-W. Jin, S. Sur, L. Chai, and D. Panda, "LiMIC: support for high-performance MPI intra-node communication on Linux cluster," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, 2005.

[12] D. Buntinas, G. Mercier, and W. Gropp, "Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem," in *International Symposium on Cluster Computing and the Grid*, May 2006.

[13] 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.

[14] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem," in *Euro PVM/MPI 2006 Conference*, 2006.

[15] S. Kumar and et al., "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer.," in *The 22nd ACM International Conference on Supercomputing (ICS)*, 2008.

[16] S. Kumar and et al., "Architecture of the Component Collective Messaging Interface," in *Proceedings of Euro PVM/MPI*, 2008.

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "MPICH: A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, September 1996.

[18] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems," in *Int'l Conference on Parallel Processing (ICPP '08)*, 2008.

[19] L. Chai, A. Hartono, and D. K. Panda, "Designing An Efficient MPI Intra-node Communication Support for Modern Computer Architectures," in *Int'l Conference on Cluster Computing*, 2006.

[20] The Ohio State University, *MVAPICH:High Performance MPI over InfiniBand and iWARP*.

[21] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood, "Implementation and performance of Portals 3.3 on the Cray XT3," in *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, 2005.

[22] *OpenMPI:Open Source High Performance Computing*.

[23] P. Carns, "Kaput, Kernel Module for copying data between processes," 2004.

[24] *InfiBniBand Trade Association. InfiniBand Architecture Specification,Release 1.1 November,2002*.

[25] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," in *In the Proceedings of 17th Annual ACM International Conference on Supercomputing*, 2003.

[26] "High Performance RDMA Protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, 2006.

[27] S. Sur, U. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda, "High Performance RDMA Based All-to-all Broadcast for InfiniBand Clusters," in *International Conference on High Performance Computing*, 2005.