HHS Public Access

Author manuscript

Proc IPDPS (Conf). Author manuscript; available in PMC 2011 June 10.

Published in final edited form as:

Proc IPDPS (Conf). 2010 April 19; 2010: 1–11. doi:10.1109/IPDPS.2010.5470407.

Optimization of Applications with Non-blocking Neighborhood Collectives via Multisends on the Blue Gene/P Supercomputer

Sameer Kumar[1], **Philip Heidelberger**[1], **Dong Chen**[1], **Michael Hines**[2]

Sameer Kumar: sameerk@us.ibm.com; Philip Heidelberger: philiph@us.ibm.com; Dong Chen: chendong@us.ibm.com; Michael Hines: michael.hines@yale.edu

[1] IBM T.J. Watson Research Center Yorktown Heights, NY, 10598

[2] Department of Computer Science, Yale University, New Haven, CT, USA

Abstract

We explore the multisend interface as a data mover interface to optimize applications with neighborhood collective communication operations. One of the limitations of the current MPI 2.1 standard is that the vector collective calls require counts and displacements (zero and nonzero bytes) to be specified for all the processors in the communicator. Further, all the collective calls in MPI 2.1 are blocking and do not permit overlap of communication with computation. We present the record replay persistent optimization to the multisend interface that minimizes the processor overhead of initiating the collective. We present four different case studies with the multisend API on Blue Gene/P (i) 3D-FFT, (ii) 4D nearest neighbor exchange as used in Quantum Chromodynamics, (iii) NAMD and (iv) neural network simulator NEURON. Performance results show 1.9× speedup with 32^3 3D-FFTs, 1.9× speedup for 4D nearest neighbor exchange with the 2^4 problem, 1.6× speedup in NAMD and almost 3× speedup in NEURON with 256K cells and 1k connections/cell.

I. Introduction

With the emergence of large massively parallel machines such as IBM Blue Gene/P [1], Cray XT4 [2] and XT5, and large clusters with low-latency interconnects (Infiniband [3] and Myrinet [4]), it is commonplace to run parallel applications on thousands of processors. For strong scaling applications, on large processor partitions, the problem size is quite small and often very small messages are exchanged. The Message Passing Interface [5] is a widely used programming paradigm for such massively parallel applications. The MPI 2.1 standard defines calls for point-to-point communication, remote memory access and collective communication. MPI has calls both for collective communication operations over the entire communicator and vector calls for communication to a subset of the nodes. These collective calls are limited as they are blocking and often optimized when a processor communicates with all or a large subset of the processors in the communicator. Applications must use MPI point-to-point operations to overlap computation and communication. But, MPI point to point operations can have relatively high overheads for short messages as incoming sends need to be matched with posted receives.

Applications such as three dimensional Fast-Fourier transform (FFT), NAnoscale Molecular Dynamics (NAMD) from University of Illinois [6], [7], and simulators for large scale

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

neural networks such as NEST [8], GENESIS [9], and NEURON [10] use algorithms where processors communicate with a neighborhood that is a subset of the processors in the communicator. We call such communication operations *neighborhood* communication operations. This size of the neighborhood can be a constant or can even be as large as the square-root of the number of nodes. However, it is typically much smaller than the size of the processor partition. NAMD and the neural network applications can also overlap communication of many short messages with computation. In this paper we explore optimizations for applications with neighborhood collective communication that can be overlapped with computation.

The Deep Computing Messaging Framework [11] (DCMF) is an open source messaging library from IBM for the Blue Gene/P (BG/P) machine. It is flexible and extensible to architectures other than BG/P and supports programming paradigms such as UPC [12], ARMCI [13] and Charm++ [14] in addition to MPI. We present the DCMF multisend interface that enables applications to send multiple messages in a single call. Data multicast, scatter and reduction are examples of multisends. These calls can also exploit special hardware features such as a global collective acceleration network on Blue Gene/P. Multisends are general and can benefit other architectures where sending several messages in a single call is more efficient than repeated send calls.

MPI collectives on Blue Gene/P are built on top of DCMF Multisend. The DCMF library is a flexible environment for exploring new algorithms for non-blocking and neighborhood collective collective communication operations. DCMF multisends can be used as a testing ground for proposals for new calls in MPI 3.0. In fact, multisends in DCMF have been discussed and have motivated MPI 3.0 proposals in the MPI Forum.

We present the novel record replay optimization to DCMF multisend, where a repeating application communication pattern can be built once and stored as a list of descriptors in a memory buffer. Successive communication calls have very low overheads since they only change head and tail pointers in the DMA unit to point to this memory buffer instead of building and injecting all the descriptors again and again for each communication phase.

We also present case studies with FFT, Lattice QCD, NAMD and NEURON to directly call the multisend interface to optimize neighborhood collectives that can be overlapped with computation.

The following are the contributions of the paper:

- Motivations, concepts and performance analysis of the DCMF multisend interface
- The record replay optimization for applications with persistent communication patterns
- Case studies showing that four scientifically important applications can benefit from the multisend and persistent optimizations

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

A. Related Work

Non-blocking extensions to the collective calls in the MPI 2.1 standard have been presented in the NBC library [15], that provides an implementation of nonblocking collectives on top of MPI point-to-point communication. The authors are aware that non-blocking collectives are now a part of MPI 3.0, but, product implementations of MPI 3.0 may not be available for quite some time. An MPI 3.0 proposal for neighborhood collectives has been presented in [16]. Many-to-many in the context of the UPC programming paradigm is presented in [12]. The DCMF active message library for Blue Gene/P has been presented in [11] and the Component Collective Messaging Interface (CCMI) for optimized MPI collective communication operations is presented in [17]. CCMI algorithms and MPI collectives are built on top of the DCMF multisend interface. But, the motivations, concepts and performance analysis of the DCMF multisend interface have not been published before.

In this paper, we explore multisends and present case studies to optimize neighborhood collectives with multisends. Since the multisend interface is based on active messages, it is more general than the current MPI 3.0 proposal [16]. Multisends can optimize programming paradigms such as Charm++ in addition to MPI. We present the record replay optimization for persistent communication operations.

II. Multisend architecture

The DCMF multisend API is an active message interface. In an active message, the header packet carries the identifier of the dispatch handler function to be executed on arrival of the header packet. The dispatch handler typically returns a buffer to receive the message payload. The handlers are invoked by the DCMF runtime during a call to the progress engine via DCMF_Messager_advance or in a background communication thread. Multisends are hence one-sided, as the sender initiates the collective and the remote nodes get notified via callbacks.

Multisends take advantage of the fact that massively parallel architectures today have a network interface that offloads communication work from the main processor. For example, the BG/P architecture adds a Direct Memory Access (DMA) engine to facilitate injecting packets to the network and receiving packets from the torus network over its predecessor BG/L. To initiate data movement the processor core injects a descriptor (similar to descriptors in Infiniband) into an injection FIFO (first-in first-out buffer). While the first send incurs the full startup overhead of the software stack and building the full descriptor, successively altering the descriptor to change a few parameters can be done very efficiently. The multisend calls take the list of destination ranks as arguments and do not require pre-created communicators. The DCMF multisend API has two flavors of non-blocking calls shown below.

A. DCMF Multicast

In a DCMF_Multicast operation (Figure 1) a processor multicasts a buffer (srcbuf) to several destinations. This call injects a DMA descriptor for each destination and then immediately returns. Request objects are passed to the DCMF API to enable the

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

DCMF runtime to store internal message state. The cb_done callback is called by the progress engine when the multicast buffer has been sent to all the destinations. The DCMF_Multicast interface has active message semantics. On the destination processors (specified in ranklist) when the first packet of the multicast is received a dispatch handler function with signature cb_dispatchMulticast and identified by *dispatchid* is invoked. This dispatch function allocates a buffer to receive the multicast message payload. While the send to the first processor incurs the full startup overhead, for successive sends only the destination in the descriptor needs to be changed and then re-injected into the injection FIFO. DCMF_Multicast can also enable broadcasts on Blue Gene/P collective network and line broadcasts on the torus network.

As the DCMF multisend interface supports several overlapping collectives, there can be several multisend messages being sent and received at the same time on each node. The different multisend messages are identified by *connection* identifiers (conn_id in the API presented above). The node that initiates the DCMF_Multicast chooses a connection id based on the global properties of the collective operation. Connection identifiers extends the *tag* in MPI point-to-point communication to collectives.

MPI Broadcast on BG/P is implemented on top of DCMF_Multicast, via a spanning tree algorithm [17]. At each level in the spanning tree, processors call DCMF_Multicast to send data to the next level of the spanning tree.

A one sided broadcast, where the broadcast initiated by the root node is received in the background on the destination processors, can be easily implemented on top of DCMF_Multicast. Here the root will call DCMF_Multicast with the ranklist having all the destination processors. The broadcast payload is received in dispatch callbacks on the destination processors.

B. DCMF Manytomany

The DCMF_Manytomany call (Figure 2) enables efficient scatter, gather, and all-to-all operations. Here, one or more processors send data from base address sr-cbuf, sizes given by size_vec and from displacements given by displ_vec to the processors in the ranklist. The DCMF_Manytomany call also has active message based semantics. A dispatch handler with signature cb_dispatchManytomany and identified by dispatchid is invoked on the destination processors. This handler returns the receive buffer, sizes vector and displacement vector for the data from many different senders. The remote index parameter (*rIndex*) is the index in the receiver displacement and size vectors for a given sender. The MPI_Alltoallv collective operation on BG/P is implemented on top of DCMF_Manytomany by setting *rIndex* to the relative rank of the sender in the communicator. The DCMF_Manytomany call is more general than MPI_Alltoallv as it can efficiently implement sparse collective operations where each processor sends data to a small subset of processors in the communicator. In an MPI_Alltoallv, the sizes and displacement vectors have one entry for each processor in the communicator, which may result in branching and cache performance overheads to process long vectors with many zero elements. In the extreme case on a next-generation petaflop million processor machine, even initiating an MPI_Alltoallv will require 16MB of memory allocation for count and displacement vectors, which is significant. However, in

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

the DCMF_Manytomany the application only needs to pass displacement and size vectors for the destination ranks that have non-zero message sizes. The new rIndex parameter will be used to look up the the receiver's compressed sizes and displacement vectors to find the address where the sender's buffer is copied. Similar to DCMF_Multicast, the DCMF_Manytomany call also takes connection identifiers as an input parameter.

C. Record Replay

In this section we present the novel record-replay optimization on Blue Gene/P, where a persistent communication pattern is recorded in a multisend call and then replayed via a single call. On BG/P, with normal point-to-point messages a descriptor must be constructed and copied into the DMA injection FIFO for each destination. In addition the DMA tail pointer in the DMA SRAM must be incremented to notify the DMA of the message send operation. The construction of descriptors and the increment of the tail pointer contribute to software startup overhead for each message. In DCMF_Multicast and DCMF_Manytomany calls, the descriptor is constructed only for the first destination and altered for each other destination resulting in better performance.

The record-replay optimization further optimizes software overhead as the replay requires only the head and tail pointers to be set in the DMA injection FIFO. The record-replay multisend implementation allocates a cumulative buffer to store injection descriptors for an application specified number of communication patterns. We added a new parameter *persistid* to the DCMF Multisend interface to allow the application to identify each persistent communication pattern. A single DMA injection FIFO is dedicated for persistent communication. Record-replay on the BG/P DMA is illustrated in Figure 3. During the record phase the head and tail pointers of the DMA point to the region of the cumulative buffer determined by the persistid. For each destination a descriptor is constructed and copied into the injection FIFO. During the replay, the head and tail pointers in the DMA FIFO are set to point to this region of pre-built descriptors. So, the replay can be done with very low overheads independent of the number of destinations in the multisend operation.

Since persistent messages are injected from different DMA injection FIFO than point-to-point messages, ordering with point-to-point messages cannot be guaranteed. Hence it is non-trivial to enable MPI persistent messages to take advantage of this optimization. However, applications that directly call the DCMF Multisend API can benefit from this record-replay optimization.

III. Benchmark and Application overview

We present the benefits of the multisend interface in the 3D-FFT and the 4D nearest neighbor benchmarks and two application case studies.

A. 3D Fast Fourier Transform

Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform and its inverse. In a three dimensional Fast Fourier Transform (3D-FFT) operation, FFT operations must be performed along the X,Y and Z dimensions. The 3D-FFT

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

operation is used in many scientific applications, such as in molecular dynamics Particle Mesh Ewald computation, quantum chemistry and distributed navier stokes.

The 3D-FFT is parallelized via a 2D pencil decomposition as the 1D decomposition has limited scalability. In the 2D decomposition, each processor has a subset of the 3D input complex numbers called a pencil. Each pencil has a subset of the data along two dimensions and all input points in the 3rd dimension. The 3D-FFT operation has three phases along X,Y and Z dimensions. In each phase of a 3D-FFT operation of size ($N \times N \times N$) on P processors, each processor has N^3/P data-points and exchanges messages with $\sqrt{P}$ other processors where the size of each message is $(N/\sqrt{P})^3$ elements. With the 2D pencil decomposition processors exchange data with row and column neighbors. During the forward FFT, each processor exchanges data first with its row neighbors in the X phase, and then column neighbors in the Y phase. With backward FFT, each processor exchanges data with its column neighbors during the Z phase and row neighbors during the Y phase.

At the limits of scalability, when number of processors P is $O(N^2)$, this 3D-FFT algorithm is communication bound. We explore four different 3D-FFT algorithms, i) MPI_Alltoallv on MPI_COMM_WORLD, ii) MPI_Alltoall on row and column sub-communicators, iii) DCMF_Manytomany, and iv) DCMF_Manytomany with record replay. We specifically chose the MPI_Alltoallv on MPI_COMM_WORLD algorithm to demonstrate scaling limitations in the MPI 2 standards, where vector collective calls must specify counts and displacements for all processors in the communicator. This scenario is critical to Lattice QCD, NAMD and NEURON, where the communication pattern cannot be mapped to collective calls on sub-communicators. In this paper, we show that the DCMF multisend calls have much better performance with subset neighborhood communication that cannot be mapped to collectives to sub-communicators.

B. 4D Near Neighbor Exchange

Quantum Chromo Dynamics (QCD) is the theory of strong nuclear force that binds the constituents of sub-nuclear matter, quarks and gluons into stable nuclei. Lattice QCD (LQCD) formulates this theory on a 4-dimensional space-time lattice. The simulations are typically dominated by a sparse linear matrix times vector solver, where the conjugate gradient (CG) method is used to solve a sparse linear Dirac operator. For this study, we have set up a simple micro-benchmark to emulate LQCD computation and communication characteristics for the standard Wilson fermion formulation.

Wilson fermion LQCD is simulated in 4 dimensions. An optimized code usually sustains about 20% to 30% of peak floating point performance [18]. We assume that the sustained performance is 25% of peak performance to calculate the cycle count for the computation in the benchmark. Assuming that the local 4D volume per processor is N^4, the total computation per CG iteration is about 1300 floating point operations per site on the Blue Gene/P PowerPC 450 core. The compute phases are split into two phases due to an even-odd preconditioning. Associated with each compute phase, half of the lattice sites on 8 surfaces of the local 4D volume need to transfer data to their corresponding nearest neighbors in the 4D topology. There are a total of 8 transfers (only 6 remote transfers are modeled in the

Author Manuscript

benchmark) of $\frac{N^3}{2} * 12$ double-precision numbers per computing phase, where most of the can be overlapped with computation. In addition to the two computing phases, there are also two global sums required by the CG algorithm.

The different compute and communication phases of LQCD benchmark are shown in Figure III-B. This benchmark actually models only a 3D exchange; the exchange in the 4th dimension is assumed to be intra-node in which the cores simply access each others computed results in a shared memory region. We also explored the above 4D near neighbor benchmark with the DCMF_Manytomany API that is a single call to which lists of source and destination processors, buffer counts and offsets, and a list of offsets to the destination buffers are passed in as parameters. A performance comparison with MPI is presented in Section IV.

C. NAnoscale Molecular Dynamics

NAMD (NAnoscale Molecular Dynamics) is a production molecular dynamics (MD) application for biomolecular simulations that include assemblages of proteins, cell membranes, and water molecules. In a biomolecular simulation, the problem size is fixed, and a large number of iterations need to be executed in order to understand interesting biological phenomenon. Hence, we need MD applications to scale to thousands of processors, even though the individual time step on one processor is quite small.

NAMD uses a hybrid strategy that combines spatial decomposition with force decomposition, and couples it with the dynamic load balancing framework of Charm++. The dominant computation in molecular dynamics is that of computing non-bonded forces i.e. electrostatic and Van der Waals forces between all pairs of atoms. The potential $O(N^2)$ all-pairs algorithm is optimized to O(N log(N)) complexity by using the notions of a cutoff radius rc, and separation of computation of short-range and long range forces. For each atom, the non-bonded forces due to atoms within rc are calculated explicitly. The long-range forces due to the atoms outside this radius are calculated using the Particle Mesh Ewald (PME) algorithm [19]. The PME algorithm has a forward and inverse 3D-FFT operation resulting in an O(N log(N)) complexity. Even with this splitting, most of the computation cost is due to explicit calculation of non-bonded forces within the cut-off radius.

Charm++ [14] is an object-oriented message driven parallel programming language. The NAMD application is built using Charm++ and gains performance and scalability by overlapping several different computation phases. The application represents computation in terms of C++ objects. The Charm++ scheduler executes different object methods when messages arrive on the network.

Figure 5 illustrates the NAMD computation, which begins with the patches (Charm++ objects that store the atom state) multicasting the atom coordinates to the compute objects. The compute objects do the force calculation and then the forces are reduced back to the patches, where the forces are integrated and velocities, energies and new positions are computed. The patches also initiate PME computation to the PME objects and that is fully overlapped with the cutoff real-space computation. PME in NAMD has six computation and

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

communication phases. First the atom coordinates are used to construct the charge grid, that is then scattered to the Z dimension PME-Z compute objects. The PME-Z compute objects perform a 1D forward FFT along the Z dimension, compute a transpose and then send messages to the Y-dimension PME computes. The PME-Y compute objects perform an FFT along the Y dimension and then sends messages to PME-X compute objects that perform a 1D FFT along the X dimension. The result of the 3D-FFT is used for the Ewald calculation followed by an inverse 3D-FFT with X,Y and Z phases to compute the long range forces. The PME Z computes send the forces back to the patches for integration.

While the FFT communication can be implemented as a blocking MPI_Alltoall on row and column sub-communicators, the remaining communication phases coordinate multicast, force reduction and patch to PME communication are neighborhood collectives that cannot be mapped to sub-communicators. Hence, the production NAMD software only uses non-blocking point-to-point messages for these neighborhood communication patterns and computation and communication is overlapped.

We extended the NAMD software to enable both the real-space communication and the PME communication to take advantage of the DCMF multisend API calls. The atom-coordinate multicast was enhanced to use DCMF_Multicast, while the PME communication was built upon the DCMF_Manytomany call. As both the DCMF_Multicast and the DCMF_Manytomany calls are non-blocking, all the different phases of the application can overlap with each other. The PME computation in NAMD is optimized via six different overlapping calls to DCMF_Manytomany on six different connections. We defined a new CmiDirectManytomany interface in Charm++ (based on the CkDirect [20] direct data placement API in Charm++) to take advantage of the DCMF_Manytomany call. We also developed a new PME kernel in NAMD that builds on the CmiDirectManytomany interface.

We use the standard 92K atom APoA1 benchmark which to demonstrate the benefit of multisends on Blue Gene/P. Computational biologists typically use molecular systems with tens of thousands of atoms similar to the ApoA1 benchmark. Past work on NAMD [6] has shown scalable performance with multiple time-stepping, where the PME computation is executed every two or four time steps. However, with the DCMF multisend calls, the NAMD application scales to 16K processors even when PME is computed every time step (Section IV).

D. Neural Networks

From the viewpoint of communication, large scale spiking neural networks consist of computational units called cells connected by one-way delay lines to many other cells. Cells generate logical events, called spikes, at various moments in time, to be delivered to many other cells with some constant propagation delay which can be different for different connections. Cells generally send their spikes to thousands of cells and receive spikes from thousands of cells. Each cell has a unique, often random, set of connections to other cells.

When the number of connections per cell is a significant fraction of the number of processors, most processors need most spikes. Furthermore, the interval between spikes generated by given cell is much larger than the minimum cell-to-cell connection delay. So

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

storing generated spikes in a buffer during an integration interval equal to the minimum cell-to-cell connection delay and exchanging spikes at the end of each integration interval using MPI_Allgather is usually better than using point to point exchange methods. However, when the number of processors is much greater than the number of connections per cell, four considerations suggest that the MPI_Allgather method will exhibit poor scaling for very large neural network simulations which, nevertheless, have sparse cell to target processor connectivity. First, MPI_Allgather itself requires twice the time when the number of processors doubles. At the limits of scalability, spike exchange must send at least a spike count and two bytes per spike to each destination, and so the allgather buffer does not become smaller when the number of processors doubles. Second, all incoming processor buffers must be examined for spikes, even if the spike count for a given source processor is 0. Third, every incoming spike requires a search in a table for whether or not the spike is needed by at least one cell on the processor. Fourth, it is not possible to overlap computation and communication. None of these issues apply with the multisend method.

Figure 6 illustrates the multisend method data flow from when a spike is generated on the source processor to when that spike is enqueued on one of the destination processors. The minimum delay integration interval (during which spikes are generated by cells but are not needed on other processors until a subsequent interval) is divided into two equal subintervals, call them A and B. Then a spike generated in subinterval A immediately initiates a DCMF_Multicast that proceeds in the background and overlaps with computation through the completion of subinterval B. At the end of subinterval B, a loop over MPI_Allreduce is entered until the number of spikes sent and received from the previous subinterval A is equal. The DCMF_Multicast has enough space (16 bytes) in its header packet to contain the integer source cell identifier, the current A or B subinterval, and double precision spike-time so no packet assembly is needed on the target machine.

IV. Performance Study

We performed an experiment to measure the performance of Scatterv, where the root sends 16 bytes to a subset of the nodes and zero bytes to the remaining nodes in the communicator. Figure 7(a) illustrates performance results when the subset of nodes that receive non-zero bytes from the root is increased from 1 to 2048. Note when the size of this subset is small, the DCMF_Manytomany techniques perform significantly better than MPI. Even when all nodes receive 16 bytes from the root, the persistent optimization is 2.7× faster than MPI. We have measured the overhead of DCMF_Manytomany to be 1.8us for the first send and 0.24us for each additional destination. With persistent manytomany, the first send has an overhead of 2.3us and there is no additional software overhead for sends to more destinations. The performance of persistent manytomany is mainly driven by the torus network throughput. We designed a similar experiment to measure the performance of MPI_Allgatherv. Here a subset of nodes have send counts of 16 bytes, while, the remaining nodes send zero bytes. Figure 7(b) shows the performance of MPI and DCMF multisend techniques as the subset of nodes that send 16 bytes is increased from 1 to 2048. Observe that the persistent optimization has the best performance here.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

3D-FFT—We measured the performance of the 3D-FFT operation using MPI_Alltoallv and the DCMF manytomany calls. The benchmark reported half the time to do the pair of forward and inverse FFTs. Figures 8(a) and 8(b) present the performance results for the 32^3 and 64^3 problem sizes on different processor partition sizes. MPI_Alltoallv on MPI_COMM_WORLD does not scale as several entries in the count and offset lists are zero. We found that MPI_Alltollv has an overhead of about 60ns for each zero element in the send and receive count vectors resulting in limited scalability. The algorithms MPI_Alltoall on a sub-communicator, DCMF_Manytomany and record replay scale with the number of nodes. Since MPI_Alltoall uses adaptive routing, packets can arrive out of order. Hence the MPI_Alltoall implementation internally calls barrier before the alltoall payload messages are exchanged. Since the 2D decomposition algorithm for 3D-FFT synchronizes all processors after each FFT, we eliminated the barrier in our DCMF_Manytomany implementation. In addition the DCMF multisend calls are at a lower level than MPI resulting in lower overheads. Hence, the DCMF algorithms outperform MPI. The best performance for the 32^3 and 64^3 FFTs are 32, 64 microseconds respectively and is achieved with the record replay many-to-many algorithm. This performance corresponds to speedups of 1.9× and 1.8× over the best MPI sub-communicator performance, and speedups of 1.3× and 1.4× over the standard DCMF_Manytomany version.

4D Near Neighbor Exchange—Table I shows performance of the 4D nearest neighbor QCD emulation benchmark on 512 nodes of Blue Gene/P. As the communication pattern is near-neighbor and the MPI_Allreduce time on the collective network only changes slightly with node count, these results are representative of performance on large BG/P machines. The table presents time for the total iteration (run), the iteration computation (comp) and link transmission time at peak throughput (wire). We found MPI has best performance in eager-mode and only those results are presented in the table. Observe that when $N = 2$, the benchmark is communication bound and the DCMF_Manytomany results in a speedup of 1.9× over MPI, as messages do not incur overheads of building 6 descriptors and matching six incoming messages to the posted receives. Notice that the network transmission time is much lower than the computation time in all cases and should be fully overlapped with computation.

NAMD—Figure 9 shows a performance comparison between the standard NAMD/Charm++ version built on top of the DCMF point-to-point API and an optimized version with calls to the DCMF multisend API. We did not consider the MPI version of Charm++ as past work [6] has shown that the native Blue Gene version performs better than Charm++ built using MPI calls. The APoA1 benchmark is simulated here. It has a 108×108×80 charge grid that has to be FFTed for the PME calculation. As mentioned in Section III, transferring charges from Patch objects to the PME objects and the PME force reduction back to the patches are both neighborhood communication patterns that cannot be mapped to sub-communicators. PME in the production NAMD software is implemented via point-to-point messages to enable overlap of computation and communication. However, the production NAMD software does not scale beyond 4096 processors when PME is executed every time step, while DCMF_Manytomany enhancement to PME scales to 16384 processors with a timestep of 1.84ms, a speedup of 1.6× over the point-to-point implementation.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

NEURON performance results—To focus on the strong scaling behavior of communication time limited models, we used networks with N = 256K cells designed to minimize computation time. Each cell is randomly connected to approximately M = 1k or 10k other cells with a 1 ms connection delay and each cell intrinsically fires with a uniform random interval between 10 and 20 ms. The precise number of input connections to each cell was drawn from a discrete uniform random distribution from M−2 to M+3. Computation time of the cell is proportional to the sum of number of input spikes to the cell and number of generated spikes and consists, for each input and generated spike, mostly of evaluation of an exponential function to determine the value of its single state variable and evaluation of a logarithm function to update the time the next output spike is generated. Since each cell is associated with a distinct statistically independent but reproducible random generator, the specific random network topology and specific random firing of each cell is independent of number of processors or distribution of cells among the processors. Simulation runs are for 200 ms. I.e. 200 integration intervals for the MPI_Allgather spike exchange method and 400 A-B integration subintervals for the multi-send method. The particular random instance of the 256K cell, 1k connections/cell model is such that 3,369,556 spikes are generated by the network. Moreover, on 8K processors the 32 cells on processor 0 each send spikes from 884 to 1003 other processors.

Table II summarizes these performance results for 8K, 16K, and 32K processors. For each of the three methods, the run time and the computation time are shown. The rows with 10k connections/cell increase the number of connections per cell by a factor of 10. With the number of connections per cell, computation scales linearly, ie with 10k connections/cell the computation increases 10 fold over 1k connections/cell. Communication time for the MPI_Allgather method is unchanged as the number of connections is increased, while communication time for DCMF_Multicast increases linearly with the number of connections/cell. Moreover, the computation time halves as number of processors doubles and is reasonably consistent across methods.

With 1k connections per cell, the DCMF_multicast and record-replay algorithms out-perform MPI_Allgather for 16K and 32K processors. The record-replay multicast optimization is better than standard DCMF_Multicast, by 1.3× with 1k connections on 32768 processors. However, with 10k connections per cell, the allgather method has the best performance. This is because MPI_Allgather on MPI_COMM_WORLD takes advantage of the collective network, that results in higher throughput than the DCMF_Multicast algorithms on the torus network. On larger processor partition sizes, we expect DCMF_Multicast algorithms to have better performance than MPI_Allgather even with 10k connections.

A. Discussion

The performance results from our case studies show that DCMF_Multicast and DCMF_Manytomany multisend APIs have significantly better performance than both point-to-point messages (LQCD and NAMD) and MPI collective operations (NEURON), for neighborhood collective communication operations that cannot be mapped to collectives on sub-communicators. Even in the case of the 3D-FFT, where the communication pattern can

Author Manuscript

be implemented as MPI collective calls on sub-communicators, the DCMF Multisend calls outperform MPI as they have flexible semantics and can avoid a barrier. In addition, the multisend calls are non-blocking and can be overlapped with the application computation. As the record replay optimization eliminates startup overheads, it further improves application performance over DCMF multisend. Its performance gains are demonstrated in 3D-FFT and Neuron.

We expect several other applications with neighborhood collective operations to take advantage of the multisend calls. The concepts behind the multisend interface are general and can be extended to other architectures with DMA units that benefit from lower overheads to inject many descriptors in one call over repeated send operations.

V. Conclusions

We presented the multisend interface as a powerful low-level interface for applications with neighborhood collectives that can be overlapped with computation. We presented some limitations in the current MPI standard and hope the approach and performance results in this paper can strengthen the need for optimized neighborhood collective operations in MPI 3.0. As the multisend calls have active message semantics, are they are not limited to just MPI and programing paradigms such as Charm++ and UPC can take advantage of them as well. We optimized the performance of the 3D-FFT benchmark, 4D near neighbor exchange, molecular dynamics application NAMD and the neural network simulator NEURON by inserting direct calls to the DCMF Multisend API with impressive performance gains. We also presented a persistent record replay multisend optimization that can make the processor overhead of the communication operations very small and showed benefits in the neural network and 3D-FFT applications. Our current implementation of record replay only uses one injection FIFO. We plan to explore multiple DMA injection FIFOs to further improve its performance.

Acknowledgments

We would like to thank Gheorghe Almasi, Gabor Dozsa, Mark E. Giamapapa, Charles Archer, Michael Blocksome, Jeremy Berg, Bob Cernohous, Ahmad Faraj, Thomas Gooding, Douglas Miller, Jeff Parker, Joseph Ratterman, Brian Smith, Carl Obert, Craig Stunkel, and Robert Wisniewski for their support in the design and development of DCMF on the Blue Gene/P machine. The work presented in this paper was funded in part by the US Government contract No. B554331.

We would like to thank Prof Laxmikant Kale, Eric Bohm, Abhinav Bhatele and Chaoi Mei from University of Illinois for guidance and support in developing and optimizing the NAMD application on Blue Gene/P. We would also like to thank Fred Mintzer and Dave Singer for granting access to the WatsonP Blue Gene/P machine.

The neural network portion of this paper was supported by NIH grant NS11613 and the Blue Brain Project. We thank Henry Markram of the Brain Mind Institute, EPFL, Lausanne Switzerland for access to the EPFL Blue Gene/L during initial prototyping of the multisend method and Felix Schürmann, also at the Brain Mind Institute, for helpful discussions about the use of multisend in a neural network context. We thank the DEISA Extreme Computing Initiative for granting access to a BG/P where preliminary simulations testing NEURON with early versions of DCMF_Multicast were carried out.

References

1. IBM Blue Gene Team. Overview of the Blue Gene/P project. IBM J Res Dev. 52 Jan. 2008

2. Alam, SR; Kuehn, JA; Barrett, RF; Larkin, JM; Fahey, MR; Sankaran, R; Worley, PH. Cray XT4: an early evaluation for petascale scientific simulation. SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing; ACM; 2007. 1–12.
3. InfiniBand Architecture Specification Release 1.0. Infini-Band Trade Association; Portland, Ore: 2000.
4. Boden, Nanette J; Cohen, Danny; Felder-man, Robert E; Kulawik, Alan E; Seitz, Charles L; Seizovic, Jakov N; Su, Wen-King. Myrinet: A Gigabit-per-Second Local Area Network. IEEE Micro. 15 (1) :29–36. 1995;
5. Forum MPI. MPI-2: Extensions to the message-passing interface. 1997 http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
6. Kumar S, Huang C, Zheng G, Bohm E, Bhatele A, Phillips JC, Yu H, Kalé LV. Scalable Molecular Dynamics with NAMD on Blue Gene/L. IBM Journal of Research and Development: Applications of Massively Parallel Systems. 52 (1/2) :177–187. 2008;
7. Phillips, JC; Zheng, G; Kumar, S; Kalé, LV. NAMD: Biomolecular simulation on thousands of processors. Proceedings of the 2002 ACM/IEEE conference on Supercomputing; Baltimore, MD. September 2002; 1–18.
8. Morrison A, Mehring C, Geisel T, Aertsen A, Diesmann A. Advancing the boundaries of high-connectivity network simulation with distributed computing. Neural Comp. :1776–1802. 2005
9. Howell F, Dyrhfjeld-Johnsen J, Maex R, Goddard N, Schutter ED. A large scale model of the cerebellar cortex using PGENESIS. Neurocomputing. 32 :1041–1046. 2000;
10. Migliore M, Cannia C, Lytton W, Markram H, Hines M. Parallel network simulations with NEURON. 21 :119–129. 2006;
11. Kumar, S; , et al. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. The 22nd ACM International Conference on Supercomputing (ICS); 2008.
12. Nishtala, R; Almasi, G; Cascaval, C. Performance without pain = productivity: data layout and collective communication in upc. PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming; New York, NY, USA. ACM; 2008. 99–110.
13. Nieplocha J, Carpenter B. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. Lecture Notes in Computer Science. 1586 1999;
14. Kale, LV, Krishnan, S. Charm++: Parallel Programming with Message-Driven Objects. In: Wilson, GV, Lu, P, editors. Parallel Programming using C++. MIT Press; 1996. 175–213.
15. Hoefler, T; Lumsdaine, A; Rehm, W. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07; Nov. 2007; IEEE Computer Society/ACM;
16. Hoefler, T; Traeff, JL. Sparse Collective Operations for MPI. Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop; May 2009;
17. Kumar S, et al. Architecture of the Component Collective Messaging Interface. Proceedings of Euro PVM/MPI. 2008
18. Vranas, Pavlos; Bhanot, Gyan; Blumrich, Matthias; Chen, Dong; Gara, Alan; Giampapa, Mark; Heidelberger, Philip; Salapura, Valentina; Sexton, James C; Soltz, Ron. The BlueGene/L supercomputer and quantum ChromoDynamics. Proceedings of the 2006 ACM/IEEE conference on Supercomputing;
19. Darden T, York D, Pedersen L. Particle mesh Ewald. An N·log(N) method for Ewald sums in large systems. JCP. 98 :10089–10092. 1993;
20. Bohm, E, Chakravorty, S, Jetley, P, Bhatele, A, Kale, LV. Tech Rep 08–12. Parallel Programming Laboratory, Department of Computer Science, University of Illinois; Urbana-Champaign: Jul, 2008 CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

DCMF_Multicast (		cb_dispatchMulticast(	
dispatchid,	/* Recv dispatch handler id */	recvRequest,	/*Buffer for recv msg. state*/
request,	/*Buffer to store msg. state*/	srcrank,	/* rank of the sender */
cb_done,	/* Callback that is invoked when multicast completes */	bytes,	/* Size of the multicast*/
		....	
conn_id,	/* Connection identifier tag*/	conn_id,	/*Connection Identifier tag */
persistid,	/* Identify the persistent communication pattern */	....	
		....	
srcbuf,	/* source buffer */	rcvbuf,	/* Buffer for multicast payload*/
size ,	/* size of the message*/	rcvsize,	/* Size of the message */
ranklist,	/* Array of dst. ranks*/	recv_done,	/*Recv completion callback called when all bytes have arrived */
nranks,	/* number of destinations */		
.....	/* Other parameters */	......	/* Other Parameters */
);		);	

Figure 1.

The DCMF_Multicast API and the receive dispatch callback

```
DCMF_Manytomany (                                          cb_dispatchManytomany (
    dispatchid,   /* Recv dispatch handler id */               ....
    request,      /* Buffer to store msg. state*/              recvRequest,  /*Buffer for msg state*/
    cb_done,      /* Callback that is invoked                  rcvbuf,       /*Base address of recv buffer*/
                  when many-to-many completes*/                rcvlens,      /* Vector of recv lengths */
    conn_id,      /* Connection Identifier tag */              conn_id,      /*Connection Identifier tag*/
    persistid,    /*Id of persistent comm. pattern*/           rcvdispls,    /* Vector of displacements*/
    rIndex,       /* location on the receiver where            ....
                  this sender's message is placed*/            nranks,       /*Num ranks to recv from */
    srcbuf,       /* Base address of source buffer */          ......
    size_vec,     /* Vector of bytes to each dst.*/            recv_done,    /*Callback to be called when
    displ_vec,    /* Displacements to each dst. */                           all data has arrived */
    ranklist,     /* List of destination ranks */              ....
    nranks,       /* Number of ranks */                        ....
    ....          /* Other parameters */                       ....
);                                                         );
```

Figure 2.

The DCMF_Manytomany API and the receive dispatch callback

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript

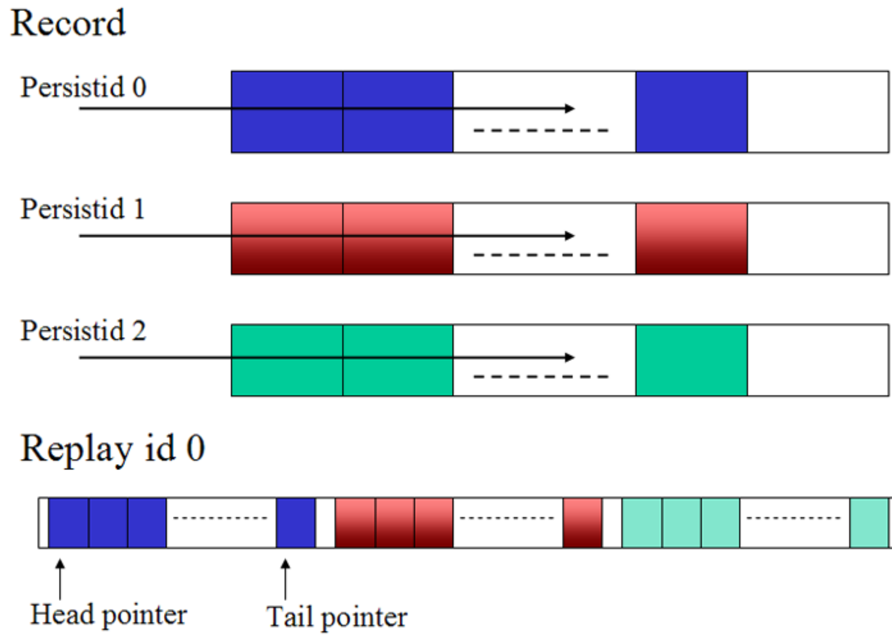


Figure 3.
Record Replay on Blue Gene/P

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

LQCD loop over iterations:	//Lattice QCD computation loop
Post_receives_0	//Post receives for phase 0
Isend_0	//Send 6 torus near neighbor messages for phase 0
Compute_0	//Phase 0 compute $\frac{(1300*N^4)}{2}$ cycles
Wait_0	//Phase 0 wait
Post_receives_1	//Post receives for phase 1
Isend_1	//Send 6 torus near-neighbor messages for phase 1
Compute_1	//Phase 1 compute $\frac{(1300*N^4)}{2}$ cycles
Wait_1	//Phase 1 wait
Allreduce_0	//Phase 0 CG global sum
Allreduce_1	//Phase 1 CG global sum

Figure 4.
Lattice QCD benchmark computation and communication phases

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

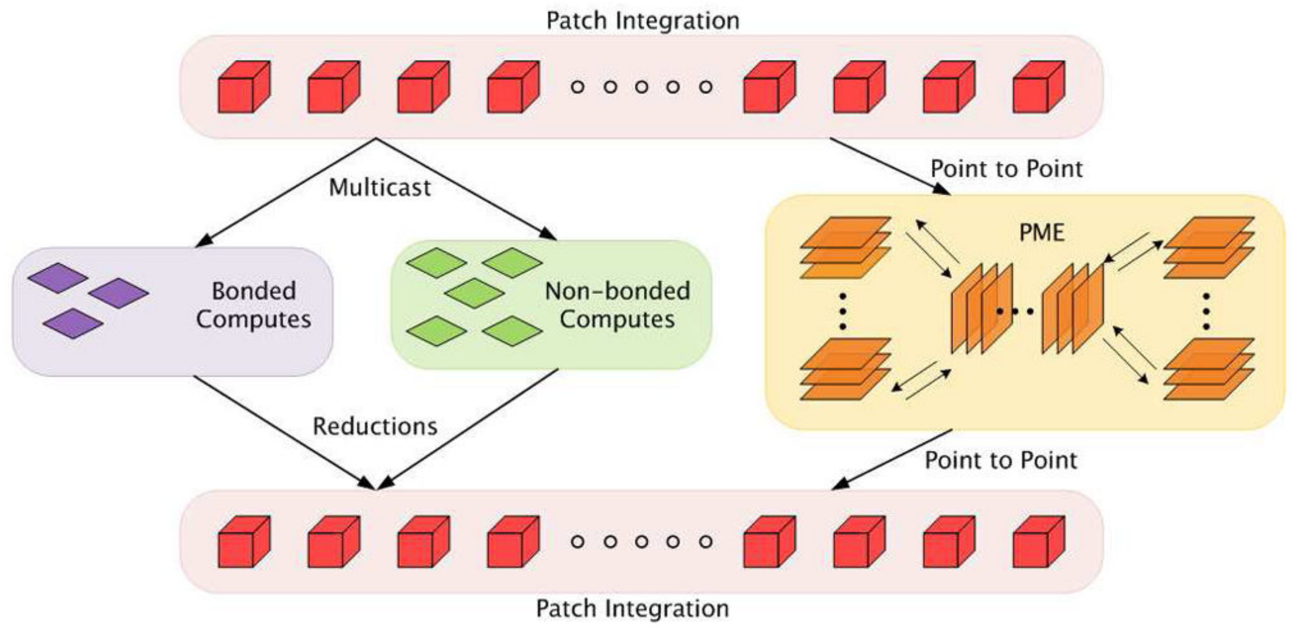


Figure 5.
Phases of the NAMD computation [6]

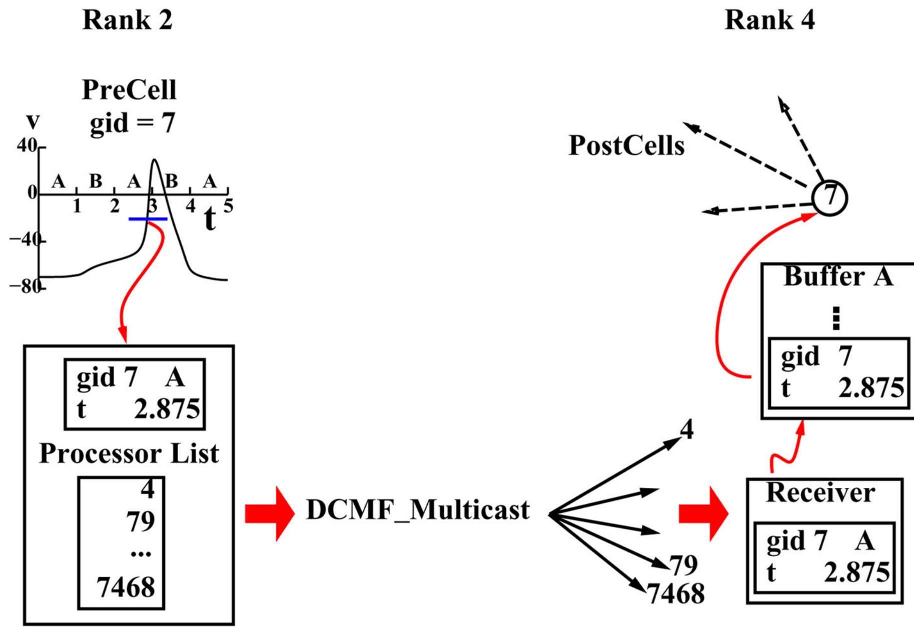
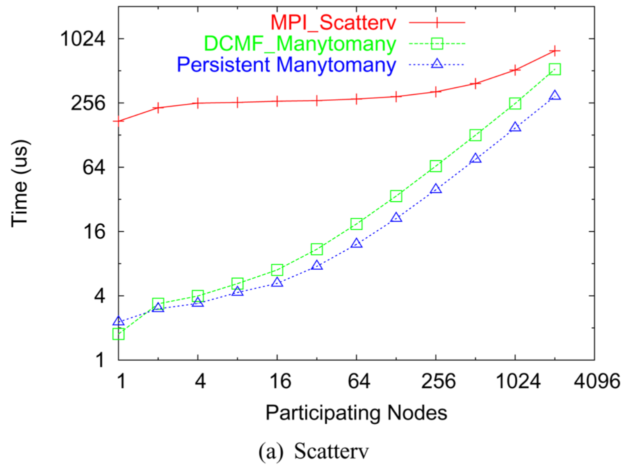


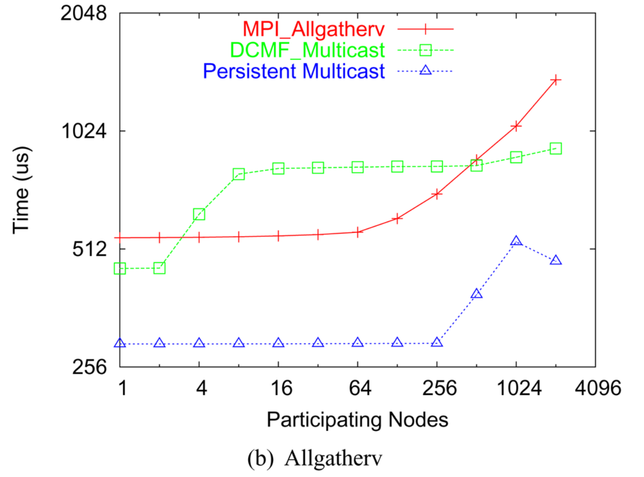
Figure 6.
The multisend method of spike exchange via DCMF_Multicast

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript



(a) Scatterv



(b) Allgatherv

Figure 7.

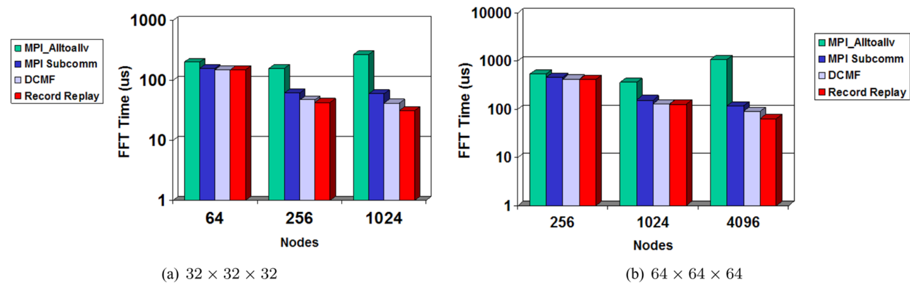
Collective operations to a subset of nodes with 16 byte per destination on 2048 nodes

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript



(a) $32 \times 32 \times 32$

(b) $64 \times 64 \times 64$

Figure 8.

Step time for 3D-FFT operations on BG/P

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

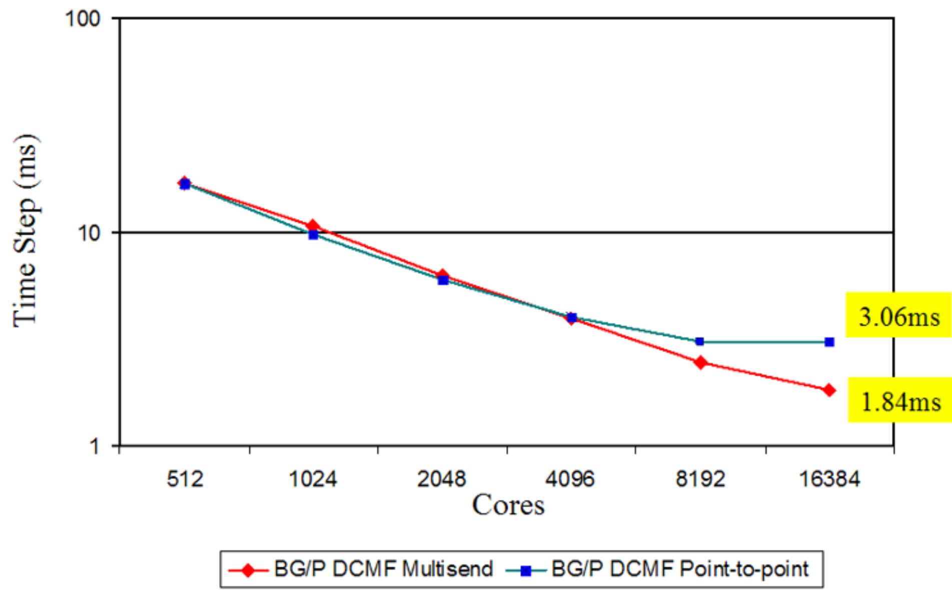


Figure 9.
NAMD: APoA1 Benchmark timestep time (ms) with PME every step in Quad mode

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Table I

4D Near neighbor benchmark performance (μs) on 512 nodes in Quad mode

	MPI Eager			Many-to-many		
Problem Size(N) N^4 sites/proc.	**run**	**comp**	**wire**	**run**	**comp**	**wire**
8	6594	6264	526	6583	6264	526
4	511	392	66	460	392	66
2	104	24.5	8.2	55.7	24.5	8.2

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Table II

Performance of NEURON (seconds) on BG/P in Quad mode

Cores	Cells	Conn.	MPI_Allgather		DCMF_Multicast		Record Replay	
			run	comp	run	comp	run	comp
8192	256K	1k	2.09	0.695	2.06	0.785	1.89	0.684
16384	256K	1k	1.76	0.353	1.25	0.397	0.979	0.347
32768	256K	1k	2.17	0.191	0.834	0.217	0.633	0.187
8192	256K	10k	11.1	6.14	14.9	6.64	14.3	6.04
16384	256K	10k	6.87	3.19	10	3.56	8.88	3.19
32768	256K	10k	4.83	1.61	6.75	1.82	5.87	1.59