Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q

Sameer Kumar[1], Yanhua Sun[2], Laxmikant V. Kalé[2]

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

[2]University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{sun51, kale}@illinois.edu

***Abstract*—**

IBM Blue Gene/Q is the next generation Blue Gene machine that can scale to tens of Peta Flops with 16 cores and 64 hardware threads per node. However, significant efforts are required to fully exploit its capacity on various applications, spanning multiple programming models. In this paper, we focus on the asynchronous message driven parallel programming model – Charm++. Since its behavior (asynchronous) is substantially different from MPI, that presents a challenge in porting it efficiently to BG/Q. On the other hand, the significant synergy between BG/Q software and Charm++ creates opportunities for effective utilization of BG/Q resources. We describe various novel fine-grained threading techniques in Charm++ to exploit the hardware features of the BG/Q compute chip. These include the use of L2 atomics to implement lockless producer-consumer queues to accelerate communication between threads, fast memory allocators, hardware communication threads that are awakened via low overhead interrupts from the BG/Q wakeup unit. Burst of short messages is processed by using the ManytoMany interface to reduce runtime overhead. We also present techniques to optimize NAMD computation via Quad Processing Unit (QPX) vector instructions and the acceleration of message rate via communication threads to optimize the Particle Mesh Ewald (PME) computation. We demonstrate the benefits of our techniques via two benchmarks, 3D Fast Fourier Transform, and the molecular dynamics application NAMD. For the 92,000-atom ApoA1 molecule, we achieved $683\mu s/step$ with PME every 4 steps and $782\mu s/step$ with PME every step.

I. Introduction

The IBM Blue Gene/Q machine is a multi Peta Flop supercomputer that can scale to 100PF. The Sequoia machine at Lawrence Livermore is a 20PF BG/Q with over 1.6 million cores and 6.4 million threads. The BG/Q machine has a 5D torus network that increases the bisection throughput over the previous generation BG/P and BG/L machines. The BG/Q systems software provides a hybrid programming model with MPI across the nodes and OpenMP within the SMP node, as running a large number of processes on a BG/Q node partitions resources such as memory and network FIFOs. Hybrid programming models give applications access to more memory and network resources, enabling them to achieve the best performance. BG/Q also provides a low level messaging library PAMI [1] (Parallel Active Messaging Interface) and POSIX threads within the nodes.

Although MPI is a popular and widely accepted programming model, it forces the programmer to be fully aware of the underlying parallel architecture and map application computation directly to processors. In this paper we present challenges in optimizing and scaling the asynchronous message driven programming model – CHARM++ [2]. In CHARM++, application computation is mapped to C++ objects called chares and the load-balancer maps these objects to processors relieving the programmer of this burden. These objects communicate by asynchronous method invocations(aka messages). Due to its own features and requirements, porting CHARM++ to a new platform requires careful consideration. For example, we use the active message based PAMI libraries to enable CHARM++ to send and receive messages on the BG/Q network. As both PAMI and CHARM++ are active message paradigms, PAMI dispatch callbacks can directly call CHARM++ handlers resulting in significantly lower overheads than the MPI port of CHARM++.

The CHARM++ runtime can enable an SMP mode where a single process runs on multiple cores, allowing easy sharing of memory. As objects within a process exchange messages via pointer exchanges instead of sending messages over the network, communication overheads within the SMP node are minimal. This SMP mode is a good match for the BG/Q architecture, where each compute node has 64 cache coherent SMP threads. However, there were several challenges we had to overcome to enable this SMP mode to scale to all the 64 threads. Our design must permit several threads to concurrently send and receive messages at high rates, thus minimizing the use of mutexes and shared data-structures in the CHARM++ runtime. We use novel fine-grained threading techniques in the CHARM++ runtime that exploit the hardware features of the BG/Q compute chip. These include lockless producer-consumer queues to communicate between SMP threads, fast memory allocators and the use of multiple background communication threads to accelerate message processing.

We demonstrate the benefits of our techniques via two benchmarks, a 3D Fast Fourier Transform, as well as the

full-fledged molecular dynamics application NAMD [3]. Molecular Dynamics simulations require Particle Mesh Ewald (PME) computation to calculate the long range forces. PME has a pair of forward and backward 3D FFT operations. We have designed an optimized implementation of the CmiDirectManytomany interface in the CHARM++ software stack to optimize PME and 3D FFT. This interface enables applications to send several messages in a burst, and our implementation on BG/Q optimizes such patterns by parallelizing them across several communication threads. We make the following contributions in this paper.

- We designed and implemented a CHARM++ runtime on BG/Q. We studied three modes of CHARM++ – non SMP, SMP without communication thread, and SMP with communication threads.
- We proposed and implemented novel ideas to fully exploit 64 hardware threads on BG/Q. These techniques include lockless queues based on L2 atomic operations, scalable memory allocator for multiple threads and optimization of the Charm++ idle poll loop.
- We optimized CHARM++ message processing by enabling multiple hardware communication threads and an optimized implementation of the CmiDirectManytomany interface that optimizes a burst of short messages.
- We optimized NAMD compute loops by the use of BG/Q Quad Processing Unit (QPX) SIMD instructions and explored a new optimized PME that calls the CmiDirectManytomany interface.
- The above optimizations provided scalable performance results for the molecular dynamics application – NAMD with the STMV 20 million and 100 million atom benchmarks up to 16384 nodes. We also achieved $682\mu s/step$ for the 92,000-atom molecular system with PME every 4 steps and $782\mu s/step$ with PME every step.

We begin with an overview of the Blue Gene/Q architecture.

II. BLUE GENE/Q ARCHITECTURE

The Blue Gene/Q machine shown in Figure 1 uses low power embedded Power PC cores to scale to the 100 PF configuration. Each BG/Q node has 18 Power ISA A2 64-bit embedded PowerPC cores running at 1.6 GHz. One core is dedicated for operating system processing and one core is a spare core, leaving 16 cores for application processing. Each core has four hardware threads that have their own register files but share other resources such as the L1 and L2 caches, compute units and load/store resources. The A2 core can execute two concurrent instructions per cycle, one fixed and one floating point, but each thread can issue only one instruction per cycle. So, to fully saturate the core's resources, at least two threads per core must be used. The L1 total cache size is 32KB with the instruction and data caches of 16KB each. The L2 cache size is 32MB that is

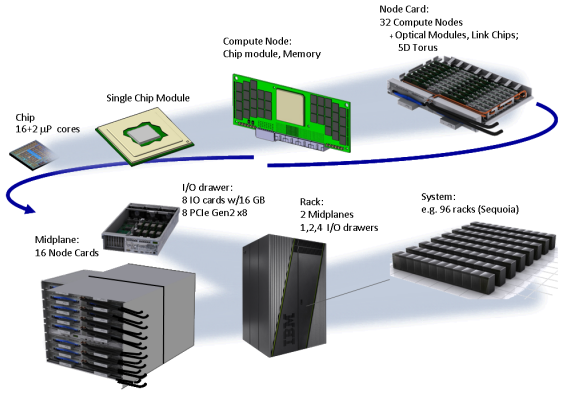


Figure 1. Blue Gene/Q architecture

divided into sixteen slices and interconnected to the A2 cores by a crossbar switch.

Scalable Atomic support in L2: On BG/Q the L2 cache has integer adders to implement L2 atomic operations such as load-increment, store-add, store-or and store-xor for 64 bit words in memory. L2 atomics have significantly lower overheads than traditional mutexes. L2 atomics can be used to design lockless queues and messaging counters that are used to track communication progress.

Wakeup unit: On PowerPC architectures, threads can go into a wait state by calling the wait instruction, where the threads do not consume any core resources such as pipeline slots. The wakeup unit on the BG/Q node can send a low overhead interrupt to awaken a hardware thread that is in a wait state. The wakeup unit can be programmed to track a range of memory addresses and network activity such as packet arrivals. Systems software can use the wakeup unit to enable communication threads that sleep in the absence of messaging work and are quickly awakened when there are incoming packets on the network.

A. Blue Gene/Q Network Architecture

Blue Gene/Q has a data network [4] with a 5D torus topology, where each link is capable of simultaneously sending and receiving at 2GB/s. Due to packet header overheads the maximum achievable throughput is 1.8GB/s. The 5D torus results in lower latency to furthest nodes and higher bisection throughput as compared with a 3D torus on BG/L and BG/P. On BG/Q the point-to-point network, the collective and barrier networks all share the same torus network. The BG/Q messaging unit (MU) is responsible for moving data between the memory and the 5D torus network. It supports three different point-to-point packet types: memory FIFO, RDMA read, and RDMA write. FIFO packets are delivered into a MU reception FIFO and RDMA packets are directly written into the memory address included in the packet. BG/Q architecture provides an extensive array of 544 MU injection FIFOs 272 MU reception FIFOs enabling several

threads to simultaneously inject and receive messages on different FIFOs.

B. Blue Gene/Q Systems Software

The BG/Q systems software stack provides optimized MPI and Parallel Active Messaging Interface (PAMI) [1] messaging libraries. BG/Q provides optimized MPICH libraries that internally call PAMI library calls. The PAMI messaging library API has calls for active message send, one-sided RDMA get and put and non-blocking collective operations. Runtime systems can create multiple PAMI *context* objects to enable fine-grained parallelism in applications. Multiple threads can concurrently call different contexts without acquiring and releasing mutexes.

API calls *PAMI_Send_immediate* and *PAMI_Send* can send short and long active messages. *PAMI_Send_immediate* copies the payload and metadata to an internal buffer and posts a single MU descriptor resulting in minimal overheads for short messages. *PAMI_Send* posts two MU descriptors, one each for the metadata and the payload. Active messages result in dispatch callbacks called on the destination nodes that must allocate buffers where the message payload is received. To advance the network and process newly arrived packets applications must call the *PAMI_Context_advance*. *PAMI_Rget* and *PAMI_Rput* enable one-sided RDMA read and write operations respectively. PAMI libraries on BG/Q also enable asynchronous progress via dedicated communication threads. Multiple communication threads can be enabled to asynchronously advance multiple contexts. Communication threads arm the wakeup unit and then call the wait instruction. While in the wait state communication threads do not consume any core resources. When a work object is queued or a network packet arrives, the communication threads are awakened by the wakeup unit to advance the context.

III. Design and Optimization of Charm++ on Blue Gene/Q

The Charm++ software stack includes a translator, message scheduler, priority queues, load balancers, the Converse adaptive runtime system, and the low level machine layer communication library. The performance of the machine layer has a direct impact on the Charm++ application performance. Therefore, it requires significant effort to optimize the machine layer interface. Co-authors have implemented and optimized Charm++ on several parallel platforms, including Cray XT/XE/XK [5], [6], [7], commodity InfiniBand clusters and IBM BG/L and BG/P machines. In this paper we focus on optimizing Charm++ on IBM BG/Q.

Charm++ supports two execution modes, non SMP and SMP. In non SMP, only one processing element (PE) executes within a process, which is typically assigned to a single core or a hardware thread. It carries out both computation and communication. This mode is easy to implement and has traditionally achieved reasonably good performance. However, in SMP mode, multiple threads run in the same process. Local communication within the process is via pointer exchange as compared with serializing and moving the message payload bytes on the network. The Charm++ load balancer can place objects that communicate with each other on the same SMP node to minimize communication overheads in applications. In addition, SMP mode also allows processing elements to access memory up to 16GB available on the BG/Q compute node. For example during the startup of some applications, PE zero may require more memory to read input files and broadcast input data structures to other processing elements. We explore SMP mode both with and without dedicated communication threads. Applications that are compute intensive can use all threads for computation and the worker threads can do message processing. Communication intensive applications can dedicate threads to accelerate communication processing. In Section V, we compare these two modes and analyze when to choose different modes to achieve the best performance.

The PAMI port of Charm++ calls PAMI active message sending call for short and medium sized messages that are processed via dispatch callbacks implemented in the Converse machine layer. These callbacks allocate buffers for the message payload and when the message is fully received it is enqueued into the Charm++ runtime's scheduler queue. For large messages, we explored a rendezvous protocol where a short header packet with the address of the source buffer and a memory region that stores the translation is first sent from the source to the destination. At the destination the dispatch callback calls PAMI_Rget to do an RDMA read from the source node. When the RDMA read completes, the destination node sends an acknowledgment packet to the sender to free the source buffer.

To optimize the Charm++ programming model, we explored the following schemes in the machine layer software.

A. lockless Queues

To send a message to an intra-node peer, the Converse machine layer enqueues the message to the peer's message queue. Typically this queue is implemented via a producer consumer queue that is guarded by a mutex. However, this mutex can be a bottleneck when several peers simultaneously send messages to the same rank. To optimize this scenario, we explored lockless queues using L2 atomic operations on BG/Q, that allow several threads on an SMP node to simultaneously send messages to a peer thread on the same node. The lockless queues take advantage of the bounded load increment operation implemented in L2 cache on BG/Q. Here, a load on a counter with a special address results in an atomic increment on the counter. The L2 cache can handle several load increment requests concurrently. The adjacent memory location to the counter specifies the bound, which is the maximum value the counter can be incremented to.

When the counter reaches the bound the future increcement operations will fail.

Each L2 atomic queue allocates a pair of L2 counters in consecutive memory locations. The first counter is the producer counter while the second is the bound. The L2 atomic queue itself is a vector of slots for message pointers. The producer thread calls an L2 bounded increment instruction that atomically increments the producer counter and returns the old value of the producer counter. The identifier of the allocated slot in the queue is computed as $producer\_counter\%queue\_size$. When the consumer dequeues the message, it increments the bound to enable producers to enqueue more messages. The L2 atomic queue is full when the producer counter is the same as the bound. This will cause the bounded atomic increment operation to fail and return an error code. We use a mutex-protected overflow queue to enable producers to insert messages even when the L2 atomic queue is full. When the overflow queue has messages, the consumer worker thread will first pull messages out of the L2 atomic queue and then the overflow queue. Figure 2(a) illustrates two L2 atomic queues on threads PE0 and PE1. The L2 queue for PE0 has three slots available, while the L2 queue for PE1 is full and there is one message in the overflow queue. Figure 2(b) shows the state of the producer and bound counters after one message is enqueued on PE0. When the PE0 dequeues the message the resulting state is illustrated in Figure 2(c).

A similar approach is used in the PAMI messaging libraries to implement lockless work queues where message and summing work requests can be submitted. These work queues are typically executed by communication threads. As MPI has a match ordering requirement, lockless queues in PAMI must lock the overflow queue and check if the overflow queue has messages before incrementing the bound resulting in higher overheads. However, the CHARM++ programming model has no message ordering requirements enabling us to design lockless queues more efficiently. The overflow queue in Charm++ is only accessed by the consumer thread when the lockless queue is empty.

B. Scalable Memory Allocation

To send a message the CHARM++ application calls the GNU memory allocator to allocate a buffer, where the application data is copied. This buffer is then passed to the CHARM++ runtime to be sent to the destination chare. On BG/Q, the memory allocator in system software uses the GNU arena allocator where allocate calls try to find an arena that is not being used by another thread. However, the free call must acquire a mutex for the arena from which the buffer was allocated to complete the free operation. We observed mutex contention overheads when several threads tried to free buffers to the same arena. This case typically happened when multiple threads received messages from the same source. To eliminate this lock contention on the free

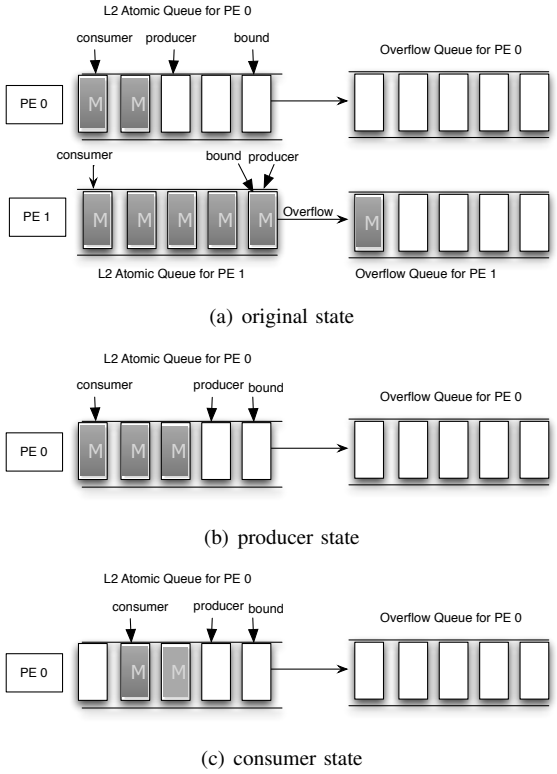


Figure 2. Lockless queue implementation

call, we enabled an L2 atomic queue for each thread to store a pool of temporary buffers. Free calls can do a lockless enqueue to the L2 atomic queue belonging to the thread that created the buffer. There is a threshold for the memory pools after which buffers are freed to the memory heap. Future malloc calls directly dequeue from the thread's L2 atomic pool via a lockless dequeue.

C. Communication Threads

To accelerate the message rate and communication processing we enabled communication threads in the PAMI library. These threads take advantage of the wakeup unit and to eliminate overheads when the communication thread is idle. Typically, a communication thread is enabled for four worker threads. Multiple communication threads can accelerate messages from several worker threads, enabling applications to be run efficiently with only one process per BG/Q node. The communication load from each worker thread is evenly distributed across all the communication threads. This benefits applications where some PEs send more messages than the others on the same node. For example, in NAMD application, PEs with atom coordinates send more messages than other PEs on the same node. Multiple communication threads can be used to accelerate these messages. Section V presents performance results with the NAMD application to show the benefit of communication

threads.

D. Optimize Idle Poll

The BG/Q compute chip has 16 cores for application computation and each core can run four threads to fully utilize its resources. However, in an iterative scientific application typically some threads finish before others. When worker threads are idle, they call into the CHARM++ runtime's idle poll loop. For best performance idle threads should not consume the core's pipeline slots, arithmetic units and load/store resources. In our optimized implementation on BG/Q, the idle poll loop spins on L2 atomic counters in the worker thread's message queue. So, when the thread is idle it stalls on L2 atomic counter load instructions take about 60 cycles to complete. Thus, minimizing the load on the core from the idle thread and enabling active threads maximal access to the core's resources.

E. Optimize Short Messages

Several Charm++ applications send short messages in the computation iteration. Computations such as Particle Mesh Ewald (PME) perform 3D Fast Fourier Transform operations that have all-to-all operations on subsets of nodes. We call such neighborhood collective communication operations as Manytomany operations [8]. The CmiDirectManytomany interface optimizes Manytomany communication in Charm++, enabling chares in Charm++ to send a burst of short messages to neighboring chares in a single optimized call. It is a persistent interface where messages with base addresses and offsets are setup ahead of time and registered with a handle. When the data is ready to be sent the application just calls start on the handle. Our implementation of CmiDirectManytomany on BG/Q generates a list of sends and receives and completes them by posting work on multiple communication threads. The work functions on the communication threads make calls to PAMI point-to-point send and send-immediate APIs that move data to neighboring nodes. When several communication threads inject and receive messages, the message rate is significantly accelerated resulting in higher application throughput as shown in Section V.

IV. BENCHMARK AND APPLICATION OVERVIEW

A. 3D Fast Fourier Transform

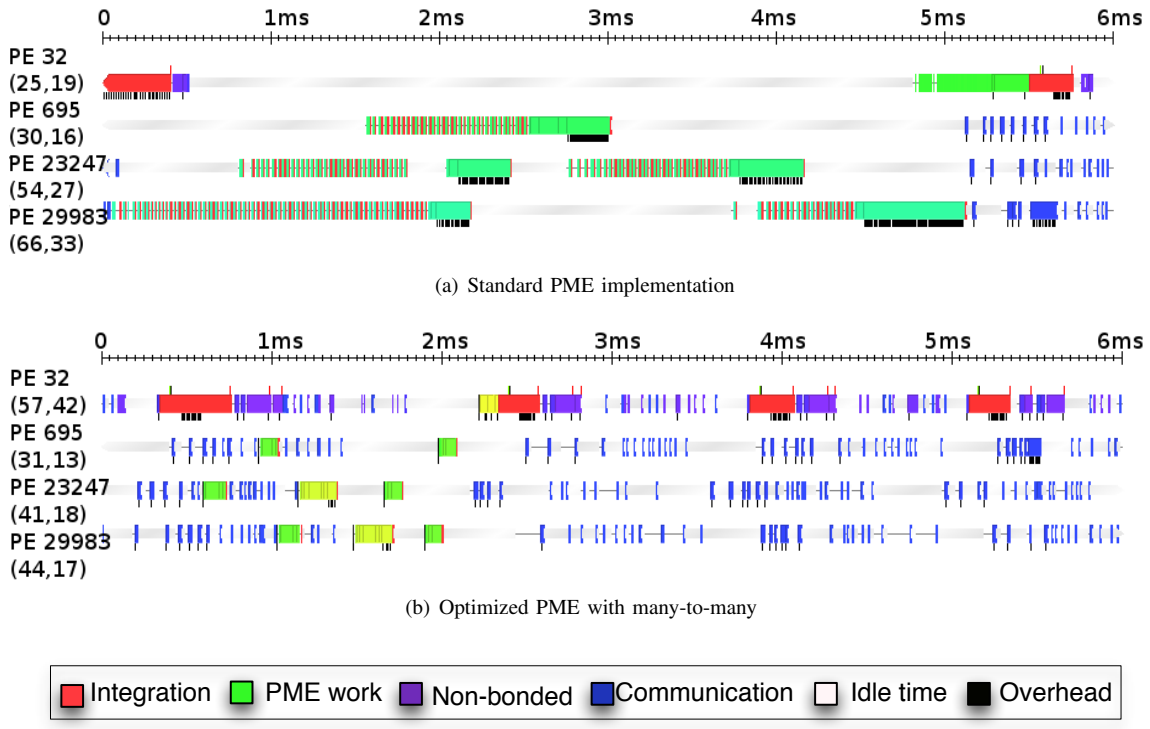
Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform and its inverse. In a three dimensional (3D) Fast Fourier Transform (3D-FFT) operation, FFT operations must be performed along the X,Y and Z dimensions. The 3D-FFT operation is used in many scientific applications, such as in molecular dynamics Particle Mesh Ewald computation, quantum chemistry [9], [10] and DNS3D turbulence codes. We parallelize 3D-FFT computation via a 2D pencil decomposition where each processor has a subset of the data along two dimensions and all input points in the 3rd dimension called a pencil. At scaling limits of an $N \times N \times N$ 3D FFT, each processor will have only one pencil in any of the three dimensions and send N messages that contain only one complex number. We explore 3D FFT via an internal FFT library in CHARM++, that communicates both via point to point messages and the CmiDirectManytomany interface.

B. NAMD

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. Molecular Dynamics (MD) simulation timestep is typically one femtosecond while biological phenomenon happen over hundreds of nanoseconds to a few microseconds of simulation time, which means hundreds of million to several billion timesteps. As problem sizes are fixed, scaling MD applications to tens and hundreds of thousands of processors can be challenging. Molecular dynamics simulators such as NAMD compute bonded (bond, angle and torsion) and non-bonded Coulomb/Leonard Jones interactions within a cutoff radius and long range electrostatics via the Particle Mesh Ewald (PME) technique. This reduces the computation to O(N + C*N*log(N)) from $O(N^2)$, where the constant C is relatively small. NAMD is developed using the CHARM++ programming language. We present optimizations for NAMD and show the benefits of the SMP optimizations presented in Section III via NAMD simulation performance improvements.

1) Optimizing NAMD compute loops: We use XL compiler intrinsic calls for the BG/Q four way QPX SIMD unit to optimize the inner loops of the NAMD application. The BG/Q processor core has a small L1 cache of 16KB that is shared by four threads, resulting in increased pressure on the L1 cache for the NAMD compute loop that has a large interpolation table. We worked with the XL compiler team to unroll loops and increase the load to use distance. This enabled other instructions to be executed while the interpolation table entries were loaded from the L1P cache. As the latency to the L1P cache is about 27 cycles, increasing the load to use distance to 27 instructions in NAMD had the best performance. The above optimizations improved the serial performance of NAMD by about 15.8% in the ApoA1 benchmark on a single node of BG/Q. We observed a speedup of 2.3x when using all four threads vs only one thread on a single core of BG/Q with the ApoA1 benchmark. The most effective way of using BG/Q core is to use all threads on the core for computation. At the scaling limits we use one or two threads for computation and communication thread on the BG/Q core.

2) PME Optimization: To compute long-range interactions via the PME technique, first the charge grid is computed and then sent to the PME processors that do a parallel 3D FFT operation via a 2D decomposition. The transformed grid is multiplied by the Ewald electrostatic kernel, and then



(a) Standard PME implementation

(b) Optimized PME with many-to-many

Figure 3. Timeline of ApoA1 simulation on 1024 nodes with regular PME and optimized PME using many-to-many

a backward 3D FFT is performed. The PME processors then send atom forces back to be integrated with cutoff forces. Both the communication of the charge grid from/to the PME processors and the FFT phases are so intensive that they can be challenging to scale.

The standard PME implementation in NAMD is feature rich, which adds to the complexity of its implementation. We developed a new optimized PME implementation in NAMD that has a static persistent communication pattern. Our new optimized implementation does not support all features of standard PME resulting in a simpler faster implementation. For example we only implement a single charge grid and there is limited support for stray charges etc. The new optimized PME creates CmiDirectManytomany handles and sets them up with all the communication operations in the different phases of PME. During the NAMD simulation it only calls $CmiDirectManytomany\_start()$ to trigger the message sends, resulting in significantly better performance over standard PME implementation in NAMD.

Figure 3 shows the timelines of four representative threads with PME work. In the figure, each line represents the execution activities of a thread on a core. Red stands for integrations to calculate atom velocities and positions. Purple color represents non-bonded compute work while green shows PME work and white color represents idle time. The numbers between the parenthesis represents the total CPU utilization and useful work utilization. Observe that with the traditional PME implementation over point-to-point messages(Figure 3(a)), each thread sends and receives 36 small messages in one FFT phase resulting in significant overheads. Figure 3(b) shows PME over many-to-many, where all messages are sent and received in a single call reducing Charm++ stack and multi-threading overheads. Therefore, with fine-grained decomposition and small messages (32 bytes in Figure 3(b)) many-to-many significantly improves performance.

V. Performance Results

We ran micro-benchmarks, 3D FFT, and the NAMD application on up to 16384 nodes of Blue Gene/Q.

The performance of Converse level ping-pong is presented in Figure 4. For messages smaller than 32 bytes, the non SMP version has the lowest latency of about $2.9\mu s$, while the latency of SMP version is about $3.7\mu s$ and $3.3\mu s$ with and without communication threads respectively. SMP with communication threads enabled provides best performance for messages between 32 bytes and $16K$ bytes. For messages larger than $16K$ bytes, the performance of the three modes is similar since the performance is dominated by network.

Figure 5 shows the Converse level latency within a BG/Q node in two modes, I) when messages are exchanged between threads that are on the same BG/Q node but in

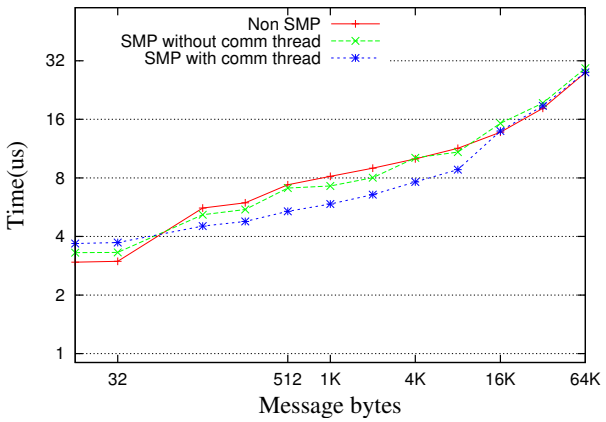


Figure 4. Pingpong one-way latency to neighboring node for different modes

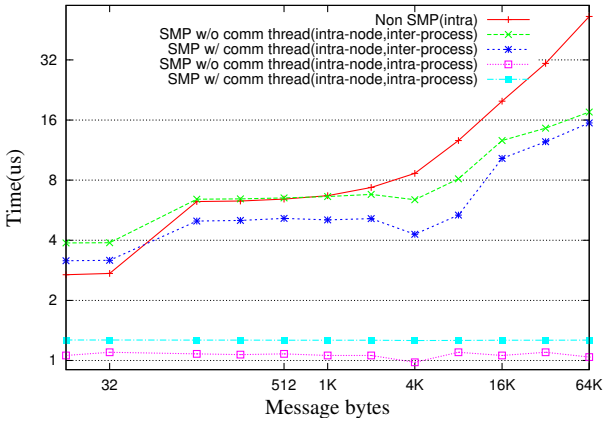


Figure 5. Pingpong one-way latency to another thread on same node for different modes

different processes and II) when messages are exchanged between threads in the same CHARM++ SMP node. Observe in case II the ping-pong latency is about $1.1\mu s$ without communication threads and $1.3\mu s$ with communication threads. This intra node latency does not change with message size as only pointers are exchanged, showing the true benefits of hybrid programming.

We ran a memory performance benchmark where all 64 threads on a BG/Q node simultaneously make calls to allocate and free memory. In this benchmark, each thread allocates 100 buffers and then frees each of the 100 buffers. Figure 6 compares the overheads with the lockless pool allocator and the direct calls to the GNU memory allocator. Observe that the lockless pool allocator has significantly lower overheads when compared with direct calls to GNU allocators. This is mainly because it avoids mutex contention overheads when multiple threads simultaneously call allocate and free as explained in Section III.

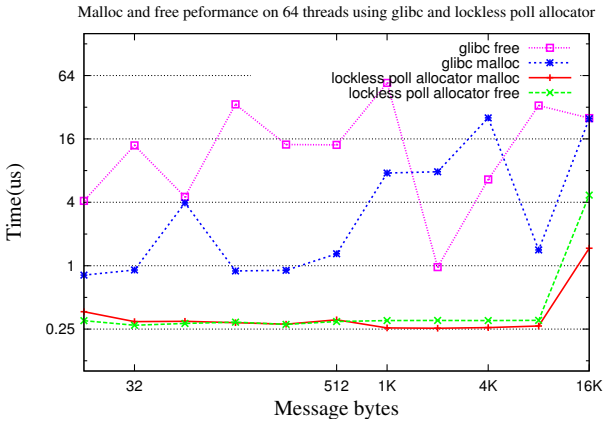


Figure 6. Malloc and free performance (us) on 64 threads w/ and w/o lockless queue and BG/Q allocator

A. 3D FFT Performance

nodes	128*128*128		64*64*64		32*32*32	
	p2p	m2m	p2p	m2m	p2p	m2m
64	3030	1826	787	507	457	142
128	2019	1426	731	459	398	127
256	1930	944	625	268	379	110
512	1785	677	625	229	376	93
1024	1560	583	621	208	377	74

Table I
COMPLEX-TO-COMPLEX FORWARD+BACKWARD 3D FFT TIME STEP (MICROSECONDS)

Table I shows the time in microseconds to execute forward and backward complex-to-complex 3D FFT computations with pencil decomposition for different problem sizes using CHARM++ point-to-point messages and the CmiDirectManytomany interface. We have two interesting points about this table. For each of the FFT problem sizes, significant performance improvements are seen with calls to CmiDirectManytomany interface in CHARM++ software stack. In addition, on the same number of nodes, CmiDirectManytomany helps performance even more with smaller sized problems. For example, on 64 nodes the speedup of m2m timestep over p2p is $\frac{3030}{1826} = 1.66$ for the $128 \times 128 \times 128$ problem while it is $\frac{457}{142} = 3.33$ for the $32 \times 32 \times 32$ problem. Moreover, with strong scaling of the same problem on different number of nodes, m2m helps even more on large node counts. It pushes the scalability limit even further, as overheads of using CmiDirectManytomany are much lower than sending and receiving fine-grained small messages.

B. NAMD results

In this section, we present the results of running NAMD on Blue Gene/Q. We use the 92,000 atom ApoA1 benchmark

with 1 femto second time step, 12 Ångstrom cutoff, constant volume and PME executed every four steps and the 20 million and 100 million atom STMV benchmarks with 1 femto second time steps, non-bonded computation every other step and PME every fourth step.

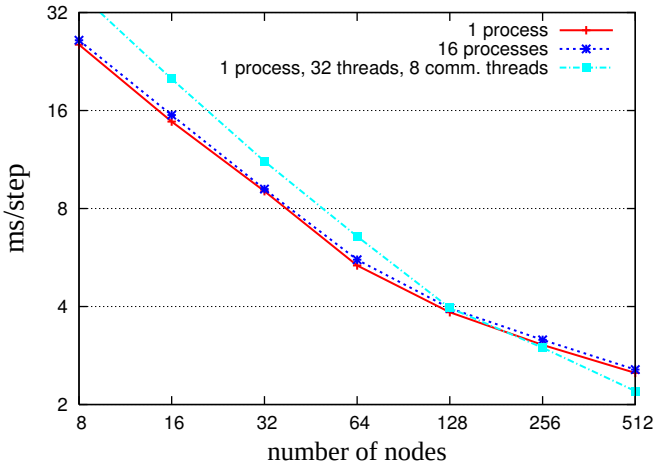


Figure 7. Comparison of performance with different configurations

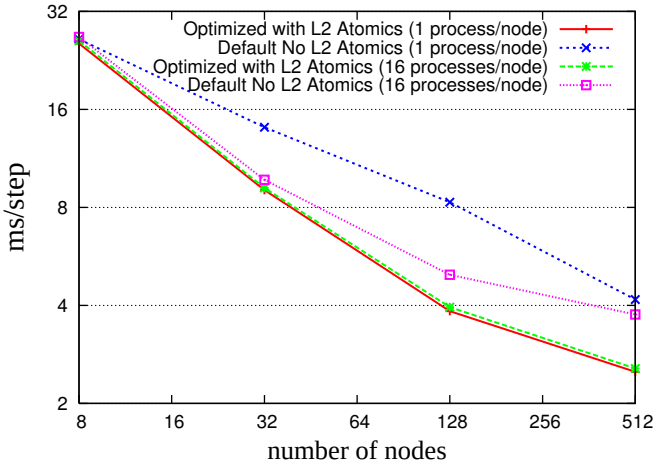
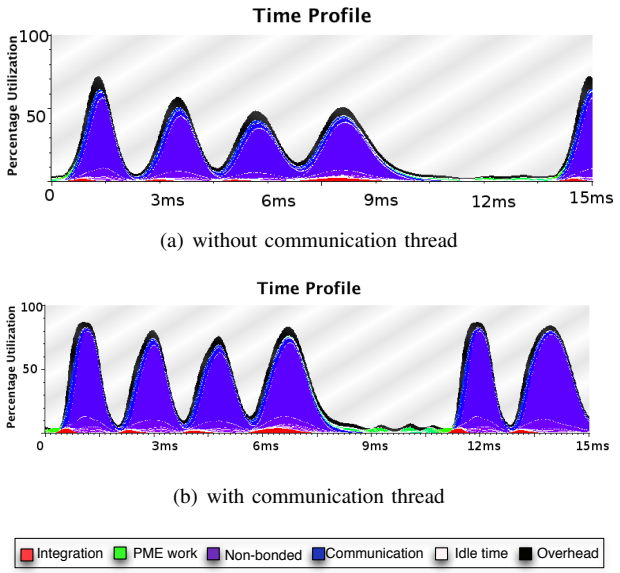


Figure 8. Performance with and without L2 atomics in two configurations

Figure 7 shows the performance of the ApoA1 benchmark with three process and thread configurations. When the application is compute bound, the case with 64 threads per node performs best. However, when the application is communication bound dedicated communication threads that optimize messaging overheads have the lowest timesteps. Figure 8 shows the benefit of using L2 atomics to optimize peer to peer communication and optimized memory allocation. Observe that at 512 nodes, L2 atomics speed up one process per node by 67%.

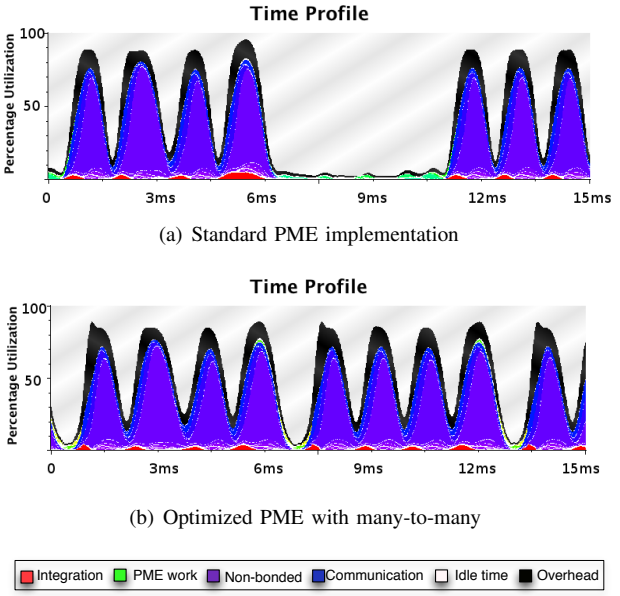
Figure 9 presents a time profile chart that shows the CPU utilization of running ApoA1 on 512 nodes with and without



(a) without communication thread

(b) with communication thread

Figure 9. ApoA1 simulation on 512 nodes w/ and w/o communication thread



(a) Standard PME implementation

(b) Optimized PME with many-to-many

Figure 10. Timeprofile of ApoA1 simulation on 1024 nodes with regular PME and optimized PME using many-to-many

communication threads. Simulation time is shown on the X axis while CPU utilization is shown on the Y axis. The colored regions show computation and communication while white represents idle time. Each peak represents a timestep. Note that every four time steps there is more idle time due to the PME computation. Also observe that utilization is greatly improved by the use of communication threads as there are more peaks in Figure 9(b). Figure 10 shows further

improvement in NAMD performance on 1024 nodes by the use of the CmiDirectManytomany to optimize PME communication in the ApoA1 benchmark. Here communication threads are enabled in both runs. The time ranges of all four runs are 15ms. Observe significantly smaller PME step when many-to-many is enabled, resulting in nine timesteps (Figure 10(b)) vs seven with standard PME (Figure 10(a)) in the 15ms window.

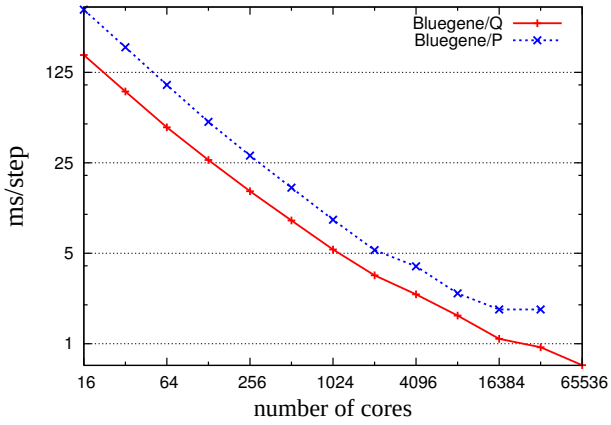


Figure 11. Comparison of ApoA1 performance with PME every 4 steps on BG/P and BG/Q

Figure 11 shows a performance comparison of the ApoA1 benchmark on BG/P and BG/Q. The best performance on BG/Q is with all 64 threads/node for computation uptill 128 nodes, 32 worker threads and 8 communication threads per node from 256 to 1024 nodes and 16 worker threads and 8 communication threads on 2048 and 4096 nodes. Optimized PME via CmiDirectManytomany is enabled at 128 nodes and larger node counts. As shown in Figure 11, we get the best timestep of $683\mu s$ on 4096 nodes of BG/Q with PME every four steps. We achieve a speedup of 2495 on 1024 nodes (16,384 cores) and 3981 on 4096 nodes (65,536 cores) respectively over a single core. These speedups are significant given that the ApoA1 is a relatively small system with only 92,000 atoms, less than 2 atoms per core at 4096 nodes. We also ran the ApoA1 benchmark with PME executed every step on BG/Q resulting in a timestep of $782\mu s$. These are the best known performance results for the ApoA1 benchmark.

Figure 12 shows the performance of NAMD with the STMV 20 million atom system. This benchmark has a large PME computation with a grid size of $216 \times 1080 \times 864$ that limits scaling of the standard NAMD PME computation. However, the CmiDirectManytomany enhancement to PME with eight communication threads to accelerate communication does scale to 16,384 nodes of BG/Q with a performance of 5.8ms/step, which is the best published result for the STMV 20 million atom system. Table II shows performance results of the 100 million atom STMV

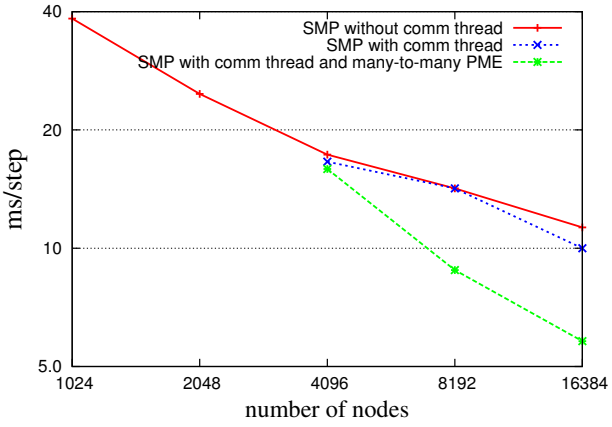


Figure 12. 20M STMV performance with PME every 4 steps

benchmark. In this table, the speedup is calculated based on the assumption that the parallel efficiency on 2048 nodes (32768 cores)is 1. This benchmark has an even larger PME computation of $1080 \times 1080 \times 864$. Optimized PME via CmiDirectManytomany results in a timestep of 17.9ms/step and a speedup of 180,864 on 16,384 nodes and 262,144 cores of BG/Q. Note, the best performance for both these systems was achieved with a single process on a BG/Q node and between 32 and 64 threads per node showing the benefits of the hybrid programming approach in the CHARM++ runtime.

VI. Related Work

MPI is the most popular programming paradigm on today's Peta scale machines. MPI and OpenMP together are gaining in popularity as a hybrid programming paradigm on BG/Q and other architectures. OpenMP relies on the compiler for optimized loop level parallelism support. The PGAS languages UPC [11] and Co-Array Fortran [12] also rely on compiler driven extensions and optimizations. Unlike compiler driven approaches the CHARM++ approach is based on dynamic runtime system optimizations. We present various optimizations to optimize hybrid CHARM++ applications on BG/Q. We take advantage of the features of CHARM++ such as no ordering requirements to further optimize performance. The techniques we present in this paper can be used to port and optimize other programming paradigms on BG/Q and on other architectures that support lock queues and communication threads in their messaging libraries.

Blue Matter [13] is a molecular dynamics application from IBM that was developed and optimized on Blue Gene/L. Desmond [14] is an MD software application from D. E. Shaw Research, while Anton [15], [16] is a specialized parallel machine for molecular dynamics. Our contribution is the exploration the hybrid programming in CHARM++ on BG/Q. The NAMD results we present are comparable to

nodes	cores	process per node	threads per process	timestep(ms)	speedup
2048	32768	1	48	98.8	32,768
4096	65536	1	48	55.4	58,438
8192	131072	1	48	30.3	106,847
16384	262144	1	32	17.9	180,864

Table II
100M STMV TIME STEP (MS) WITH PME EVERY 4 STEPS

Desmond. For ApoA1 we report the best known performance of $683\mu s$/step with PME every four steps and $782\mu s$/step with PME every step. Moreover, hybrid programming with one process per node enables simulation of very large molecular systems such as the 20 Million and 100 Million atom STMV benchmarks with scalable performance.

VII. SUMMARY AND FUTURE WORK

We presented our exploration of the CHARM++ programming paradigm on IBM Blue Gene/Q. We presented several new techniques to enable fine-grained threading in the CHARM++ runtime, such as lockless queues and scalable memory allocators by taking advantage of scalable atomics in the level-2 cache on BG/Q. We also enabled communication threads in the PAMI library to accelerate message processing. Performance results showed record performance for NAMD to simulate the ApoA1 benchmark on 4096 nodes, that is $782\mu s/step$ with PME every step. NAMD also scales up to 16,384 nodes for the STMV 20 million and 100 million atom benchmarks.

We have observed that at scaling limits we get the best performance with one or two worker threads per core and a communication thread. This is because running with a larger thread count increases communication and CHARM++ scheduling overheads that cancel the benefits from using additional threads on the BG/Q core. We plan to explore fine-grained schedulers to take advantage of all four threads on the BG/Q core even when step times are very small. As BG/Q has a 5D torus network, we get good scaling for most of our NAMD benchmark runs even with topologically oblivious placement. However on larger BG/Q configurations we expect topological placement will improve performance and we plan to explore that as well.

VIII. ACKNOWLEDGMENTS

We would like to thank Bob Walkup and Paul Coffman for help in porting and optimizing NAMD on Blue Gene/Q and Daniel Zabawa for technical support with the XL compiler. We would like to thank Ray Loy and Wei Jiang at the Argonne National Laboratory for their assistance in running NAMD at the ANL MIRA machine. The work presented in this paper was funded in part by the US Government contract number B554331. This work was supported in part by a NIH Grant PHS 5 P41 RR05969-04 for Molecular Dynamics.

REFERENCES

[1] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burow, "PAMI: A parallel active message interface for the BlueGene/Q supercomputer," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Shanghai, China), May 2012.

[2] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++* (G. V. Wilson and P. Lu, eds.), pp. 175–213, MIT Press, 1996.

[3] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Baltimore, MD), pp. 1–18, September 2002.

[4] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q interconnection network and message unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 26:1–26:10, ACM, 2011.

[5] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson, "A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Shanghai, China), May 2012.

[6] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[7] Y. Sun, G. Zheng, C. M. E. J. Bohm, T. Jones, L. V. Kalé, and J. C.Phillips, "Optimizing fine-grained communication in a biomolecular simulation application on cray xk6," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, (Salt Lake City, Utah), November 2012.

[8] S. Kumar, P. Heidelberger, D. Chen, and M. Hines, "Optimization of applications with non-blocking neighborhood collectives via multisends on the Blue Gene/P supercomputer," in *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Atlanta, GA), April 2010.

[9] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna, "Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers," *Journal of Comptational Chemistry*, vol. 25, pp. 2006–2022, Oct. 2004.

[10] M. E. Tuckerman, "Ab initio molecular dynamics: Basic concepts, current trends and novel applications," *J. Phys. Condensed Matter*, vol. 14, p. R1297, 2002.

[11] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the upc language," in *18th International Parallel and Distributed Processing Symposium*, p. 254, 2004.

[12] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey, "A multi-platform co-array fortran compiler," in *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, (Antibes Juan-les-Pins, France), October 2004.

[13] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, M. C. Pitman, and R. S. Germain, "Molecular dynamics—blue matter: approaching the limits of concurrency for classical molecular dynamics," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 87, ACM Press, 2006.

[14] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossvary, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan, and D. E. Shaw, "Scalable algorithms for molecular dynamics simulations on commodity clusters," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), ACM Press, 2006.

[15] C. Young, J. A. Bank, R. O. Dror, J. P. Grossman, J. K. Salmon, and D. E. Shaw, "A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–11, ACM, 2009.

[16] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles, "Millisecond-scale molecular dynamics simulations on Anton," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.