Optimizing Fine-grained Communication in a Biomolecular Simulation Application on Cray XK6

Yanhua Sun, Gengbin Zheng, Chao Mei, Eric J. Bohm, James C. Phillips, Laxmikant V. Kalé
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{sun51, gzheng, chaomei2, ebohm, jcphill, kale}@illinois.edu
Terry R. Jones
Oak Ridge National Lab, Oak Ridge, TN 37830, USA
trjones@ornl.gov

***Abstract*— Achieving good scaling for fine-grained communication intensive applications on modern supercomputers remains challenging. In our previous work, we have shown that such an application — NAMD — scales well on the full Jaguar XT5 without long-range interactions; Yet, with them, the speedup falters beyond 64K cores. Although the new Gemini interconnect on Cray XK6 has improved network performance, the challenges remain, and are likely to remain for other such networks as well. We analyze communication bottlenecks in NAMD and its CHARM++ runtime, using the *Projections* performance analysis tool. Based on the analysis, we optimize the runtime, built on the uGNI library for Gemini. We present several techniques to improve the fine-grained communication. Consequently, the performance of running 92224-atom Apoa1 with GPUs on TitanDev is improved by 36%. For 100-million-atom STMV, we improve upon the prior Jaguar XT5 result of 26 ms/step to 13 ms/step using 298,992 cores on Jaguar XK6.**

I. Introduction

Biomolecular simulations are critical in understanding the functioning of biological machinery: the proteins, cell membranes, DNA molecules, etc. Atom-by-atom simulation of such systems allows us to determine the relationship between structure of proteins and their functions, understand various biological processes, and facilitate rational drug design.

Molecular dynamics simulations for this domain are challenging for one main reason: the significant difference between the timescales at which interesting biological phenomena occurs, and the time-steps at which the simulation must be carried out to maintain accuracy. The vibrational modes of many covalent bonds involved require the time-step to be around one femto-second. In contrast, hundreds of nanoseconds (and often several microseconds) of simulation are necessary to observe interesting behavior. A goal of $10\,ns$ per day requires finishing each time step in $8.6\,ms$, while a rate of hundred nanoseconds per day requires a time step in $860\,\mu s$.

The number of atoms involved in such simulations is relatively small: most simulations involve between hundred thousand to 10 million atoms, with a few interesting outliers with around hundred million atoms. Correspondingly, the computation involved in each time step is also relatively small: a single time step of a 1 million atom simulation requires about 20 seconds on one core of a modern processor. This paper is focused on challenges involved in scaling such simulations to the point where it may take a few milliseconds per time step.

NAMD [18] is a molecular dynamics application that was developed in the mid-1990's. Unlike its contemporaries at that time, NAMD was designed from scratch to be a parallel program. Its basic parallel structure, based on the CHARM++ programming system [11], has withstood the test of time. However, exploiting extreme strong scaling in the petascale era requires addressing unprecedented challenges. NAMD, running a 100 Million atom molecular system, was selected by NSF as an acceptance test for the Sustained Petascale Blue Waters platform. Cray's Gemini architecture underlies both the XK6 and XE6/XK7 platforms of Titan and Blue Waters, which further motivated this work in that optimizations for Titan will naturally extend to Blue Waters. In our earlier work, we have shown how NAMD scales to 64K cores of BG/P, and 224k cores on Jaguar XT5. Table I presents the performance of NAMD running a 100-million-atom system on Jaguar XT5 in 2011 [12]. It can be seen that the speedup starts to falter beyond 64K cores. A 1-million-atom simulation could not be scaled beyond 4K cores, where it achieved $8.66\,ms$/step. This paper addresses this poor scalability caused by the fine-grained parallelization, and describes the techniques we developed to attack the challenges so that the application can be scaled to new heights.

Cores (100 million STMV)	1680	26880	53760	107520	224076
Timestep	1343.9	94.38	54.25	37.10	26.28
Cores (1 million STMV)	125	500	2000	4000	8000
Timestep	162.33	44.72	15.35	8.66	11.8

TABLE I
TWO BENCHMARK TIMESTEP(MS/STEP) ON JAGUAR XT5

Although the new Gemini network helps performance, the fundamental challenge in the fine-grained parallelization still remains and motivates new software techniques described in this paper. NAMD, like most biomolecular simulation applications, uses particle-mesh-Ewald (PME) method to calculate the long-range forces. As our analysis in the subsequent section shows, this method becomes the primary performance bottleneck for strong scaling. A series of techniques are then presented to eliminate the bottleneck. With these, we demonstrate that a small molecular system can run in only a few hundred microseconds per step, and a 100-million-atom system scales to almost 300,000 cores with an average of 13 ms/step.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

In the era when the HPC community moves towards exascale computing, we believe that the lessons learned from this application shed light on other applications that target at better scalability with fine-grained parallelization. Here is the summary of our **contributions**:

- We present an SMP (multithreaded) execution mode based on Gemini's low-level interface (uGNI) for the message-driven programming model, which enables various low-level optimizations for fine-grained communication.
- We extend *Projections*, a performance analysis tool, with new features specifically for uGNI to facilitate the detection of communication bottlenecks.
- Several optimization techniques are developed to improve fine-grained communication, including supporting out-of-band messages, assigning higher priority to messages on the critical path, decomposing work in more efficient way and speeding up intra-node computation.
- GPU implementation in NAMD is optimized, which helps improve performance as much as 36% for very fine-grained decomposition on TitanDev.
- We have achieved $13\,ms$/step and 64% parallel efficiency for the 100M-atom simulation on $298,992$ cores of Jaguar XK6, compared with $26\,ms$/step and 38% efficiency on the full Jaguar XT5 machine.

In the rest of the paper, Section II describes the background of the Cray XK6 system, the CHARM++ runtime system and NAMD. Section III presents the performance analysis tool used to identify performance bottlenecks in NAMD. Communication optimization techniques are presented in Section IV. Strong scaling performance on GPUs and CPUs is reported in Section V. Related work and conclusions are discussed in Section VI. and Section VII.

II. BACKGROUND

In this section, we briefly describe the target Jaguar Cray XK6 machine and Gemini interconnect. Next, we present the CHARM++ programming model as well as the associated runtime system, biomolecular simulation program NAMD and its GPU design.

A. Titan - Cray XK6 with Gemini Interconnect

Our results stem from measurements taken on Oak Ridge National Laboratory's flagship computer - the Jaguar supercomputer. During calendar year 2012, the Jaguar supercomputer is undergoing a multi-phased upgrade that will result in a much more capable machine with a mostly hybrid architecture of central processing units as well as application accelerators called graphics processing units (GPUs). So far the upgrade has converted the Cray XT5 system – an all-CPU model with an older, smaller core-count version of the AMD processor – into a Cray XK6 system with Interlagos CPUs, twice the memory of its predecessor, and a more capable network named Gemini. After cabinets have been equipped with NVIDIA's newest Kepler GPUs by the end of 2012, the peak performance will be at least 20 petaflops, and Jaguar's name will be changed to Titan. At the time of our experiments, ten of Jaguar's two-hundred cabinets were upgraded to the target CPU-GPU hybrid to create a testbed system called TitanDev. Throughout the rest of the paper, we will refer to the entire machine (nodes with and without GPU accelerators) as Jaguar XK6, and the subset of the machine with GPUs as TitanDev.

The whole system is connected using the Gemini Interconnect. In comparison to the preceding SeaStar2+ interconnect, Gemini improves latency from $5\,\mu s$ to $1\,\mu s$ and bandwidth from $2\,GBytes/s$ to $8\,GBytes/s$ as well as provides better hardware support for one-sided communication. One Gemini ASIC serves two nodes by connecting each node to one network interface controller (NIC) over a non-coherent HyperTransport(TM) 3 interface. The NIC provides two hardware components for network communication: the *Fast Memory Access* (FMA) unit and the *Block Transfer Engine* (BTE) unit. It is important for developers to properly utilize both of them to achieve maximum communication performance.

User-level Generic Network Interface (uGNI) is a set of APIs for developers to interact with Gemini hardware. It defines a collection of functions to transfer data using the FMA and BTE units. Besides FMA/BTE post transactions, GNI *Short Message* (SMSG) is available for developers to transfer small messages. *Completion Queues* (CQ) are provided as a light-weight event notification mechanism to track the progress of local and remote FMA/BTE transactions. The rich set of functions provided by uGNI APIs makes it challenging to implement a general high-performance runtime system since many aspects need to be considered in the design process.

B. Charm++ Runtime System

CHARM++ is a parallel programming model which implements message-driven parallel objects. CHARM++'s portable adaptive runtime system automatically maps these objects and associated computation to processors evenly using the load balancing and communication optimizations.

Although CHARM++ runs on top of any MPI communication library including Cray MPI, to achieve the best performance, it is ported directly to the uGNI interface [19]. In this paper, we extended the work to support the SMP (multi-threaded) execution mode [13], [12], which is tuned for running on multicore-based parallel machines.

In SMP mode, there are multiple flows of control per process, called "worker threads", which are typically implemented via pthreads. Worker threads share their parent process's address space, but contain their own event schedulers and appear semantically as independent ranks with a persistent mapping of objects. Each thread is typically mapped to one core and persists for the life of the application. A communication thread handles the network communication for all the worker threads in the process, and dispatches messages to the event schedulers of its worker threads. In non-SMP mode, in contrast, each process embodies only one control flow, which handles both event scheduling and network communication. Several benefits of SMP mode result from the shared memory space, such as reduced overall memory footprint, less memory bandwidth consumption, faster application launch, and more efficient intra-node communication [12]. Section IV-A will describe the implementation of the SMP mode on uGNI interface.

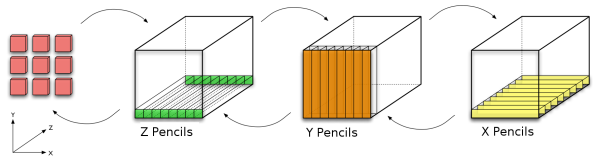


Fig. 1. Pencil PME communication of data

C. NAMD

The parallel structure of NAMD is based on a unique object-based hybrid decomposition, parallelized using the CHARM++ programming model. Atomic data is decomposed into spatial domains (called "patches") based on the short-range interaction cutoff distance such that in each dimension only atoms in one-away or, when necessary to increase concurrency, one-away and two-away neighboring domains will interact directly. These equally-sized domains are distributed as evenly as possible across the machine and are responsible for accumulating forces and integrating the equations of motion asynchronously via per-domain user-level threads. Patches are represented on other cores by proxies and all position and force data sent between cores passes via these proxy patches.

The calculation of short-range interactions is orthogonally decomposed into "compute objects" representing interactions between atoms within a single domain, between pairs of domains, or for groups of neighboring domains for terms representing multi-body covalent bonds. Compute objects are scheduled by local prioritized CHARM++ messages when updated position data is received for all required patches. Longer-running domains are further subdivided by partitioning their outer interaction loops to achieve a grain-size that enables both load balancing and interleaving of high-priority PME or remote-atom work with lower-priority work that does not require off-node communication.

1) NAMD Long-Range Interaction (PME): The long-range interaction in NAMD is implemented via the Fast Fourier Transform (FFT) based particle-mesh Ewald method (PME). PME calculation, due to the data transposes required in three dimensional Fourier transforms, is highly communication intensive [22], and therefore very challenging to scale. While NAMD supports both slab (one dimensional decomposition) and pencil (two dimensional decomposition) PME, this paper addresses only the pencil form due to its superior scaling characteristics [6].

The communication required is illustrated in Figure 1, which highlights the critical path of communication issues constraining performance. Constructing the grid and extracting the result from it is shown at left and the 3-D FFT forward and backward at right. Pencil based distributed parallel implementations of 3-D FFT have communication requirements that are well studied in the literature [6], so we present a minimal summary of the critical issues for completeness. Furthermore, the communication process from reciprocal space to real space is the reverse of the real to reciprocal process, therefore only the forward path will be considered in detail.

Given a PME grid of $N_x \times N_y \times N_z$ points, it is decomposed into pencils using n_x, n_y, n_z points per dimension. Each of the $\frac{N_x}{n_x} \times \frac{N_y}{n_y}$ ZPencils will therefore have $n_x \times n_y \times N_z$ points and will construct $\frac{N_z}{n_z}$ messages of size $n_x \times n_y \times n_z \times 4$ bytes to each intersecting YPencil. Each of the $\frac{N_x}{n_x} \times \frac{N_z}{n_z}$ YPencils of $n_x \times N_y \times n_z$ points will construct $\frac{N_y}{n_y}$ messages of size $n_x \times n_y \times n_z \times 4$ bytes to each intersecting XPencil. Points per pencil, n_x, n_y, n_z, are free parameters chosen at run time to produce the desired quantity of pencils. NAMD uses one dimensional FFTW3 [5] plans for the actual FFT operation at each step, communication is handled by CHARM++ messaging which may then be subjected to various optimizations as described in later sections of this paper.

Molecule	Atoms	Cutoff($\AA$)	Simulation Box
DHFR	23558	9	62x62x62
Apoa1	92224	12	108x108x77
1M STMV	1066628	12	216x216x216
100M STMV	106662800	12	1084x1084x867

TABLE II
PARAMETERS FOR FOUR MOLECULAR SYSTEMS

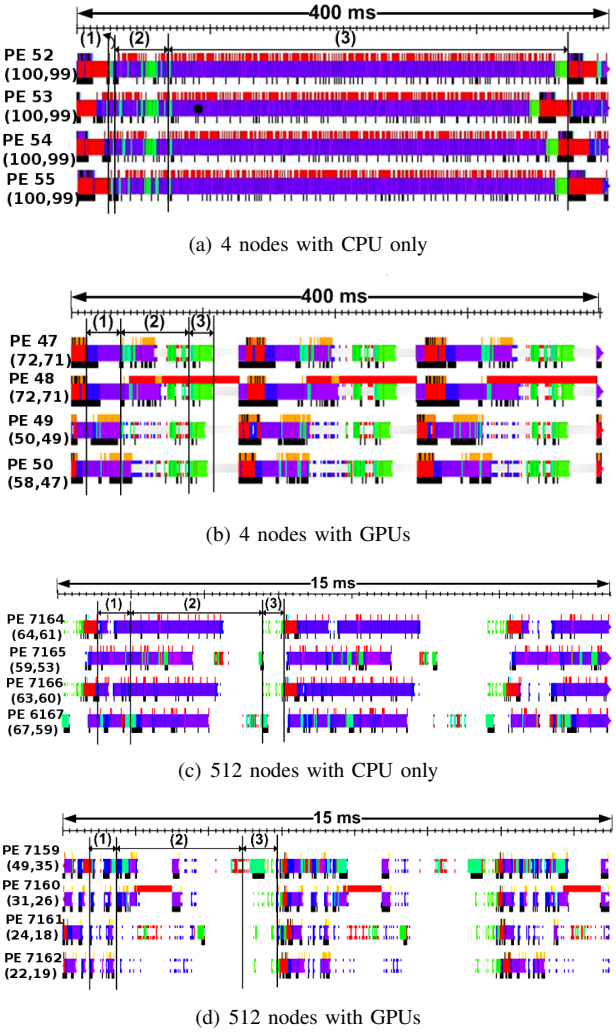
2) NAMD GPU Design: NAMD offloads only short-range non-bonded calculations to the GPU as these are the most expensive part of the calculation, are best suited to the GPU architecture, and do not require additional communication stages. Every non-bonded compute object assigned to a given GPU is mapped to one or two GPU multiprocessor work units ("blocks" in CUDA). Although GPUs support very fine-grained parallel decomposition internally, the interface presented to the controlling CPU requires data and work to be aggregated for efficiency. When all required position data is received on the CPU, it is then copied to the GPU and two sets of work units are scheduled. The first set calculates forces for remote atoms, which require off-node communication, and the second set calculates only forces for local atoms. This allows some overlap of communication and computation [17].

With the increasing power of GPUs and increasing core counts of CPUs, the design of NAMD has shifted from each CPU core having its own independent context to a possibly shared GPU, to each GPU being managed by a single CPU process. This new arrangement greatly increases the amount of work available to the GPU to execute in parallel while eliminating the copying of redundant atom coordinates to the GPU. However, CHARM++ SMP execution mode is now necessary to allow multiple CPU threads to share the overhead of servicing the GPU, presenting a challenge to parallel scaling as the network must be driven by a single communication thread per GPU rather than multiple processes per node. The increase in CHARM++ SMP communication performance required by GPU-accelerated NAMD has driven many optimizations described below.

3) Experimental Setup: In this paper, four different sized molecular systems (see Table II) are used for performance evaluation. All simulations are of biomolecules in explicit solvent employing the CHARMM all-atom force field and periodic boundary conditions. For all experimental results in Section IV and V, PME is performed every 4 steps.

III. PERFORMANCE ANALYSIS

In order to optimize the fine-grained communication performance in NAMD, it is necessary to understand the details of



(a) 4 nodes with CPU only

(b) 4 nodes with GPUs

(c) 512 nodes with CPU only

(d) 512 nodes with GPUs

Fig. 2. Projection timeline views of NAMD simulating 1M STMV using 4 nodes and 512 nodes

application communication patterns message by message. In particular, the critical path and communication bottleneck has to be identified and analyzed. To realize this capability, in this section, we describe a trace-based performance analysis tool, called *Projections* [10], and extend it to support uGNI-based CHARM++'s SMP mode. The extension allows us to monitor uGNI low-level communication events and significantly facilitates optimization efforts on improving PME communication.

A. Trace-based Performance Analysis Tool

Projections is a comprehensive trace-based performance analysis tool associated with CHARM++. It consists of an automatic runtime instrumentation module that generates trace files for an application execution, and a stand-alone Java GUI program to visualize and analyze the performance data.

The runtime instrumentation module is embedded in the CHARM++ runtime system. It automatically records the performance metrics, including execution time of user functions invoked by messages, and events such as sending and receiving a message, idle time and etc. The stand-alone Java GUI program offers many tools to visualize and analyze performance , such as: displaying the timeline of events on a given set of processors, illustrating CPU utilization, detecting load imbalance, and finding performance outliers.

In order to trace the details of messaging behavior in the uGNI layer, we extended the runtime instrumentation module to trace uGNI events and completion events. In CHARM++ SMP mode, all inter-node communication is handled by the communication threads. The worker threads push the messages to the network queues, which are processed by the communication threads to deliver to the network. By instrumenting SMSG, RDMA operations (GET and PUT), and their completion time, we can record the activities on communication threads. For instance, to send a medium size message with a rendezvous protocol, the sender sends a control message to the receiver, and the receiver reserves registered memory and posts a GET RDMA transaction. Adding instrumentation to every stage of this protocol, we can easily understand the cause of message delay in much more detail. For example, a prolonged completion time for a GET transaction can suggest a delay in network hardware due to network contention.

B. PME Communication Issues

Figure 2 is generated by the timeline tool in *Projections*. It shows the execution details of running 1 million STMV simulation on TitanDev with 4 nodes and 512 nodes respectively. Each sub-figure depicts the activities on 4 chosen processor cores. Each timeline stands for the execution of a thread on one core. Different colors represent different functions, while the portion in white represents the idle time. The blocks in the main timeline mean the activities on CPU, while the red bar on top of the blocks represents the GPU execution, which overlaps with the CPU execution. Meanwhile, the two numbers in the bracket below the PE number indicate CPU utilization and the percentage of the useful work excluding the runtime overhead. In the figure, three phases in PME are separated by vertical lines in one timestep: (1) Patch-to-Pencil communication; (2) Forward/Backward FFT; (3) Pencil-to-Patch communication.

In Figure 2(a) on the 4 nodes without GPUs, the communication time of PME phases totally overlaps with the bonded and non-bonded calculation. Therefore, almost 100% CPU utilization is obtained. In the case of using GPU (Figure 2(b)), the non-bonded calculation is moved to GPU, while bonded and PME calculation is still performed on CPUs. As a result, CPUs have less work and go idle while waiting for both GPU work and PME communication to finish. The utilization is around 70% for the first two cores and 50% for the other two cores. Still, the overall GPU performance is 2.7 times of that using only the CPUs. When scaling to 512 nodes shown in Figure 2(c) and Figure 2(d), the CPU utilization drops to 60% for CPUs only run and 40% for GPUs run. It is clear to see that the PME phases become the bottleneck. In the particular aspect of NAMD, this bottleneck is due to the lack of overlap between the non-bonded and the long range computation. Fundamentally this is caused by the ratio of atoms/core decreasing with strong scaling. Although the performance analysis in this section is carried out on small number of cores run, similar analysis and conclusion also

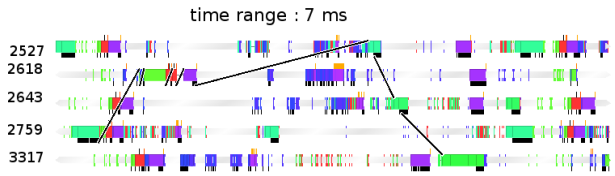


Fig. 3. Message tracing in Projections for NAMD simulation 1M STMV using 512 GPU nodes on TitanDev

applies to large number of cores run, maintaining the the same atoms/core ratio.

Knowing that PME is the bottleneck, to figure out where messages are delayed in the network, we developed a new feature in *Projections* to trace messages back to its sender and repeat until it finds the first sender. Using this feature, it is convenient to locate the occurrence of long message delay along the path. An example is shown in Figure 3. The message tracing of a PME message reveals a $2.1ms$ latency for a kilobyte message, while the timestep is only $7ms$. In a quiet network, the latency of such messages usually is only a few microseconds. Several factors may lead to such unexpected message delay. It could be due to the runtime system, where the communication thread is a bottleneck in processing messages, or it could be due to network contention or bandwidth limit. This motivates our work to optimize PME communication, which is discussed in the next section.

IV. Optimizing Communication

As described previously, the fine-grained communication in PME imposes a bottleneck to the strong scaling performance. In this section, we present several techniques to improve the communication by optimizing both application and the underlying runtime system.

A. Communication Progress Engine in SMP Mode

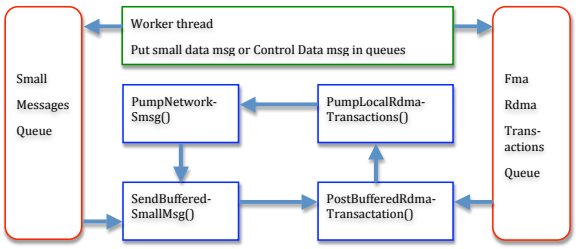


Fig. 4. Design of uGNI-based SMP communication engine

In [19], we have demonstrated the advantage of the uGNI-based Charm++ runtime over the MPI-based one. In this paper, we will continue focusing on the uGNI-based implementation, but extend it to support SMP execution mode. In this mode, one communication thread calls the progress engine to serve several worker threads, performing the tasks of both sending and receiving messages via uGNI interface. The tasks of a communication thread in the Gemini machine layer are illustrated in Figure 4. When a worker thread sends a message, depending on the size of the message, it is enqueued in the small message queue or the FMA/RDMA transactions queue as shown in the figure. The communication progress engine repeatedly performs the following four tasks in order:

PumpNetworkSmsg(): check incoming small messages from SMSG mailbox.

PumpLocalRdmaTransactions(): check the local FMA/RDMA completion queue (CQ) for the completion of GET/PUT transactions.

SendBufferedSmsg(): send buffered SMSG (small) messages.

PostBufferedRdmaTransaction(): post buffered GET/PUT RDMA transactions.

During bursty communication, the communication server can easily become a bottleneck when it is busy performing the above tasks sequentially. For example, a small message which arrived at the Gemini NIC can only be polled and delivered to the worker thread when the communication server calls *PumpNetworkSmsg*. The message can be stuck if the communication thread is busy performing other tasks. When a message is in a critical path, the unresponsiveness of the communication thread can greatly delay the message, and negatively impact the overall performance.

This subsection describes a few techniques to speedup the communication progress engine's turnaround time in processing its tasks.

Reducing small messages A large amount of small message traffic is due to the small ACK messages. For example, when a GET RDMA transaction finishes on a processor, a small ACK message is sent to the source processor to tell it to free the send message buffer. To eliminate the small ACK messages and reduce the burden of the communication server, we use a technique in uGNI that exploits the *remote events* generated on the remote processor (i.e. the source processor for GET and the destination processor for PUT operation). The remote event data can be set by the processor issuing the FMA/RDMA transaction. When the transaction is completed, the remote event data is obtained by the remote processor polling the completion queue (*GNI_CqGetEvent*). However, the remote events carry only 32 bit of data, which in our scenario is not enough to store a 64-bit memory address. This can be solved by storing the actual address of the message buffer in a table, and using its table index as the remote event data. In our experiments up to 298,992 cores, this scheme works well.

Removing scaling bottleneck Using our uGNI instrumentation and statistics collection of the progress engine, we found that function *SendBufferedSmallMsg()* takes a significant amount of time, which is much longer than the other 3 progress engine calls. In *SendBufferedSmallMsg()*, the communication server processes all buffered small messages. These messages are sorted in queues according to the destination. As the number of nodes increases, looping over all the queues can be expensive. By using the statistics counters in our tracing framework, we observed that the cost of communication server calling *SendBufferedSmallMsg* function increases linearly with the number of nodes. To overcome this scaling problem, instead of looping over all the queues, we maintain a linked list which contains only the non-empty queues. Although locks are needed when modifying the link list, the optimized *SendBufferedSmallMsg()* no longer has to loop over all cores. Moreover, the number of threads contending the lock does not

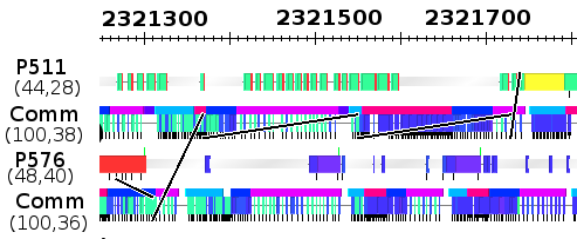
changes with the number of nodes. Therefore, it has no scaling issues.

Increasing responsiveness As shown in Figure 4, the communication thread alternates among four different tasks. If a communication thread spends extra long time on any one of the tasks, other tasks are delayed, which may result in prolonged message latency, especially for messages on critical paths. To prevent this, we put a limit on the number of messages each task can process, so that each tasks take no more than certain amount of time. This simple method improves the responsiveness of the communication thread. Later in section IV-B , we will show how this change combined with other techniques reduces the latency of the message on the critical path.

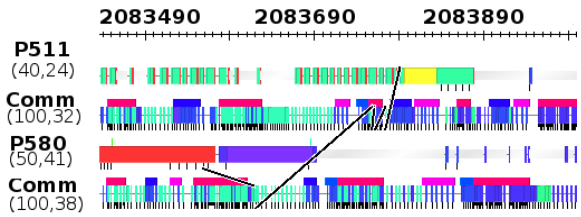
B. Priority Messages on Critical Path

In molecular dynamics simulation, typical computation includes bonded/non-bonded calculation, long-range force calculation by PME and integration of updating velocity and position. From the perspective of communication, the non-bonded and bonded calculation requires data from the home patches and a few neighbor processors while PME calculation is a three-dimensional FFT calculation. These different types of computation and communication pattern are driven by different types of messages in CHARM++. Some of these messages play a significant role in performance, while others do not. For example, if the message latency can be overlapped by computation, it does not affect the overall performance. In Section III, we see that the communication in NAMD PME phases is the performance bottleneck for strong scalability. When a time step starts, both cutoff messages and PME messages are sent to the network at the same time. This burst of messages incurs significant overhead on the communication thread and causes network contention. Therefore, it is highly desirable to associate PME messages with higher priority than non-bonded compute messages. In the optimized scheme, on the sender side, high priority messages are sent as soon as their data is ready. The communication server delivers these messages to the network even when there are other low priority messages waiting in the send buffer queue. The implementation of supporting such high priority messages on the sender side is the *out-of-band* sending. On the receiver side, the high priority messages are handled as soon as possible, even when there are other low priority messages in the scheduler queue. It is implemented by prioritized execution and expedited messages.

Out-of-band sending: High priority messages should be handled with much less turn-around time in messages queues. To realize it, besides the two queues for regular messages in Figure 4, two more special queues are added for the high priority messages. The corresponding sending functions are implemented and called by the communication thread more often than the four regular functions. For GET-based large message transfer, the control message is assigned with a special uGNI tag so that the receiver issues the FMA/RDMA transaction immediately instead of buffering it. Moreover, the transaction for this high priority message is associated with



(a) Without optimization



(b) With optimization

Fig. 5. The message tracing of patch-to-PME in Projections timeline for DHFR running on 1024 cores

a specialized Completion Queue (CQ), which is polled more often to ensure responsiveness.

Prioritized execution: CHARM++ messages can attach two kinds of priorities: integer priorities and bit vector priorities. In both cases, numerically lower priorities will be dequeued and processed before numerically greater priorities. When CHARM++ messages are delivered to the receiver core, typically they are first deposited in a low level FIFO network queue, and after they are processed, moved to a priority-based message queue for execution.

In NAMD, the above two techniques are applied to PME messages. To see the effect of the above optimizations (including the optimizations described in Section IV-A), we ran NAMD with the DHFR system on 1024 cores. Before optimization, as the Projections timeline view in Figure 5(a) shows, it takes $470us$ for a 3020-bytes patch-to-PME message to send from core 576 on node 82 to core 511 on node 73. Out of the total of $470us$, $100us$ is spent on sending the small control message to the communication thread on the destination node 73 after calling *PumpNetworkSmsg()*. After the control message is processed on node 73, a GET RDMA request is enqueued. When the communication thread calls *PostBufferedRdmaTRansaction()*, it posts the RDMA GET transaction. It takes another $178us$ for the GET transaction to finish and communication server calls *PumpLocalRdmaTransactions()* to check the event. As seen in the figure, calling *PostBufferedRdmaTRansaction()* is delayed because it has to wait until the ongoing task *PumpLocalRdmaTransaction()* (shown in the pink color) is finished, which takes $119us$. Also, delivery and processing this PME message can be delayed due to other non-PME messages ahead in the queue. After applying the optimizations described previously, including putting a limit on amount of work each task can do at a time, and out-of-band messages support, the time is reduced from $470us$ to $240us$ as shown in Figure 5(b).

Persistent communication for FFT: In NAMD, PME messages are typically a few kilobytes. Sending these medium

sized messages involves a small control message transfer and an FMA/RMDA GET transaction. This message passing does not match naturally to the one-sided communication uGNI provides, since the memory address on the remote processor is unknown at the time of sending a message.

For communication with fixed patterns, such as the PME communication, the communication can be expressed in the persistent communication API in CHARM++, using the persistent messages implemented on uGNI [19]. With persistent messages, a persistent channel between the sender and receiver is setup once at the beginning, so that the sender knows the memory address and handler of the receive buffer. When sending a persistent message, a PUT transaction is issued directly using the information in the persistent handler. Besides saving the time of sending the small control message, there is another benefit of using persistent messages for PME. Since PME objects do not migrate among processors, and the PME communication repeats in each PME step, this PME persistent channel can be used repeatedly to avoid memory allocation and registration. Similarly, the *remote events* idea discussed in section IV-A is also applied here to avoid the ACK message which was needed to notify the remote side the completion of a PUT transaction.

Using persistent messages for PME in NAMD, we observed about 10% overall performance improvement running the 100M STMV atom system on the full Jaguar XK6 machine.

C. PME Decomposition

The problem size of 3D FFT used for long-range force calculation in NAMD is fixed in strong scaling. Therefore, performance can be greatly influenced by how the FFT is decomposed. To better understand the effect of the PME decomposition, we describe the total time cost of one phase of the 3D FFT in Eq. 1.

$$T = T_{comm} + T_{comp} = \frac{D}{4 * B * \alpha} + \frac{NlogN}{P} \quad (1)$$

Here N is the FFT size in one dimension, P is the total number of cores, and D is the total amount of FFT data. Half of the data needs to be moved to the other half processors. Data movement in one direction is therefore $\frac{D}{4}$. B is the bisection bandwidth, and α is the sustained ratio for particular message sizes, which is always less than 1. In a quiet network, with the message size increasing, α approaches 1. Therefore, large messages, corresponding to bigger PME pencils, utilize network bandwidth more efficiently. However, having bigger PME pencils means fewer PME pencils so that fewer cores can be used for computation (less parallelism), which increases the time in the second portion of the equation, leading to worse performance. Another potential problem with fewer but larger PME pencils is that it may lead to less overlapping of computation and communication. Therefore, finding a tradeoff number of PME pencils is critical to the overall PME performance.

It is inefficient to have multiple PME objects per core due to per object communication overhead. It is desirable though to have at least one PME object on each physical node to maximize network capacity. This gives two extreme cases of decomposing FFT computation: (1) one PME object per core, or (2) one PME object per physical node. Assigning one PME object to each core maximizes parallelism, allowing better overlap of computation and communication. However, this finer-grained decomposition may result in too many small messages, which makes it challenging to fully utilize the network bandwidth.

Number of Nodes	4K	8K	16K
multiple PME each node	49	27.5	16.9
one PME each physical node	47	26.3	16.2
one PME each Charm node	45	25.1	15.7

TABLE III
100M-ATOM SIMULATION BENCHMARK TIME (MS/STEP) USING DIFFERENT NUMBER OF PME OBJECTS

The other extreme case places one PME object on each physical node to better utilize the network bandwidth. This comes at the cost of reduced parallelism within each node, furthermore each PME object has to wait for more data to arrive before it can start computation, which may lead to idle time.

The tradeoff in between is that each SMP process has one PME object. On Cray XK6, experiments show that 2 processes (with multiple worker threads each) for one node gives best performance, while on Cray XE6, it is 4 processes on each node (XE6 has 32 cores/node). This approach tries to combine the benefits of overlapping computation and communication and reducing communication overhead. Table III shows how the performance is affected by different decompositions. It can be seen that the middle case of having one PME on each CHARM++ SMP node gives best performance. This is the configuration that we used for experiments in Section V.

One potential problem with this configuration is that only one of the worker threads in an SMP process owns the PME object and performs the expensive PME calculation, leading to load imbalance inside a node. Next we describe a scheme to parallelize the PME work among worker threads.

Exploiting Intra-node Parallelism: To parallelize the PME calculations, one option is to use OpenMP to parallelize the for loops in PME. However, currently it is not straightforward to use OpenMP directly in CHARM++. This is because OpenMP and CHARM++ manage threads on their own. The scheduling of OpenMP threads and CHARM++ worker threads is not coordinated, which may result in degraded performance. In addition, CHARM++ is an asynchronous message driven runtime, each worker thread schedules messages independently, while OpenMP assumes all threads are available when parallelizing a loop. Therefore, applying the same idea of OpenMP, we implemented a micro runtime utilizing the CHARM++ worker threads to parallelize the loops, with a dynamic scheduling scheme that considers the load of other worker threads.

In this scheduling scheme, when a loop is partitioned into small pieces of tasks, only the idle CHARM++ worker threads will execute these tasks. When a loop is parallelized, we create a loop descriptor to represent the loop parallel task. Inside this descriptor, in addition to basic information about the loop such as the range of loop iterations etc., we store the number of chunks that the loop is divided into and an "index" indicating the chunk that is currently being executed. After this loop

descriptor is broadcast to every PE on the same SMP node, the idle thread will retrieve the atomically incremented "index" value, and start executing the chunk of loop indicated by the "index". The idle threads continue doing this until there is no remaining chunks for execution. The thread that originates this parallel loop task will also participate this process, and do a busy-waiting at the end until all tasks are finished.

System	#Nodes	no Exploit	Exploit	Speedup(%)
DHFR	128	1.76	1.56	12.82
Apoa1	128	2.32	2.16	7.41
1M-STMV	1024	4.69	4.47	4.92

TABLE IV
PERFORMANCE (MS/STEP) OF EXPLOITING INTRA-NODE PARALLELISM (PME EVERY TIMESTEP) IN NAMD

Applying the intra-node parallelism into NAMD PME, we have obtained observable performance improvement, shown in table IV, when PME is on the performance scaling critical path. Since we have chosen to place at most one PME object per CHARM++ node, the total PME computation will be fixed on each node beyond a certain scale. This implies there is a theoretical upper limit in the reduction of PME computation time we can achieve per node. The fine-grained communication in PME will then begin to dominate the overall PME performance, which re-emphasizes the importance of communication optimization techniques described in section IV.

D. GPU Optimization

In order to allow pipelining of results from the GPU to the CPU at finer granularity than supported by the CUDA runtime facilities we have modified the NAMD GPU kernel to stream force data back to the GPU at the level of individual spatial domains. This is accomplished by combining system-level memory fence operations on the GPU to ensure that force data written to CPU memory is fully visible, with periodic CPU polling of an output queue. Receiving forces from the GPU incrementally allows the CPU cores to begin sending results and even integrating the equations of motion while the GPU is still calculating forces. One outstanding issue with this approach is how to order work on the GPU to have some results complete earlier while not extending total GPU runtime and delaying the final results. Table V presents the results of running Apoa1 with PME every 4 steps on GPUs with and without the pipelining optimization. On 32 nodes, we saw the biggest gain of 23%. However, the improvement becomes less when scaling to 64 nodes and 128 nodes, where PME communication starts to dominate performance. Therefore, all the communication optimizations discussed above also help GPUs performance. The results will be reported in section V.

Number of nodes	16	32	64	128
Initial SMP GPU	3.55	2.86	1.86	2.45
Pipelining SMP GPU	3.45	2.32	1.72	2.30

TABLE V
APOA1 GPU BENCHMARK TIMESTEP (MS/STEP) W/ AND W/O PIPELINING

V. PERFORMANCE RESULTS

In this section, we present the results of NAMD running different sizes of molecular simulation with PME every 4 steps using both GPUs and CPUs on TitanDev, and CPUs only on the Jaguar XK6 machine. Performance using GPU is analyzed and compared. We also compare the NAMD performance on new Jaguar XK6 with the old Jaguar XT5 for a 100M-atom benchmark up to the whole machines. The effect of optimizations presented above is demonstrated by comparing the uGNI-based NAMD results with the initial performance.

A. NAMD GPU Performance

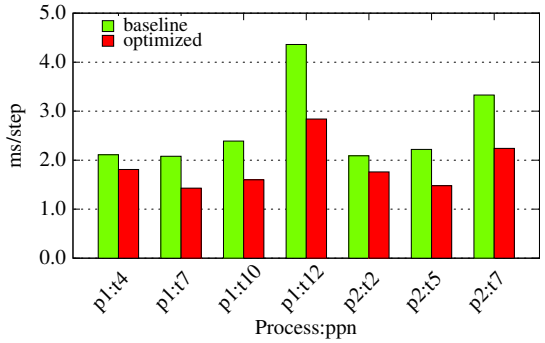


Fig. 6. NAMD Apoa1 GPU performance on 64 XK6 nodes (TitanDev) for varying numbers of processes (p) per node and worker threads(t) per process.

We first compare the performance of the initial version of NAMD running Apoa1 with the optimized version using different number of threads on one node. Figure 6 shows the time step (in milliseconds) of NAMD running on 64 GPU nodes with different configurations of CHARM++ processes and work threads. We can see that the performance varies as much as 2 times using different configurations. Using more cores does not necessarily provide better performance; the worst performance occurs when using 12 work threads. This is mainly due to the communication thread becoming bottleneck in serving 12 work threads. In the other aspects, optimized version outperforms the baseline ones by as much as 54%, which is the case with 12 threads in one process. Among all configurations, the best performance is $2.08\,ms$/step in the baseline version, while $1.43\,ms$/step in the optimized version.

The benchmark time of running Apoa1 simulation on different number of GPU nodes using different versions of NAMD is presented in Table VI. All results are obtained with NAMD on uGNI-based CHARM++. The first two rows are the results of running non-SMP and SMP versions of NAMD. SMP-built GPU version performs better than non-SMP NAMD with a speedup as much as 1.34 on 16 nodes. This is mainly due to the benefit of avoiding memory copy in the SMP-built NAMD. Due to the limitation of CUDA library on TitanDev, only one process can communicate with GPU, all data on the physical node have to be moved to that host process. In non-SMP mode, each core runs a NAMD process (vs. pthread in SMP version), data must be copied from guest processes to the host process via inter-process communication which is slow. In SMP mode, all NAMD threads share the same address space in the parent process so that the data copying is avoided. However, with the number of nodes increasing, the amount of data on each node decreases. This limits the benefits of using GPUs, and the CPU work starts to dominate the execution

time. Therefore, the advantage of using SMP mode reduces as the number of nodes increases. The last second row in Table VI presents the results after applying the optimization techniques described in Section IV. Comparing with the initial performance, the performance is improved by as much as 36%. The last row lists the results of running Apoa1 system without GPU. Similar timestep performance is achieved on 256 CPU nodes comparing with 64 GPU nodes. In the CPU only case, NAMD scales up to 512 nodes, while in the GPU case, NAMD does not scale on more than 128 GPU nodes. The missing data points in the Table VI could not be obtained for that configuration of Apoa1, which has only 144 patches, because the GPU version of the code only functions when there is at least one patch per GPU.

Number of nodes	16	32	64	128	256	512
Non-SMP GPU	4.67	3.05	2.25	2.5	-	-
Initial SMP GPU	3.55	2.86	1.86	2.45	-	-
Optimized SMP GPU	3.45	2.10	1.44	1.92	-	-
Optimized SMP CPU	9.16	4.85	2.82	1.75	1.36	1.13

TABLE VI
APOA1 XK6 BENCHMARK TIMESTEP (MS/STEP)

Table VII compares the NAMD GPU benchmark times running the 100M-atom system before and after the optimizations. In both cases, the performance scales well up to 512 nodes when the GPU computation still overlaps the PME communication. However, the scaling drops after 512 nodes. Using our runtime network counters, we found that some messages take much longer time than the usual case, possibly due to network contention. Indeed, the GPU allocation on TitanDev has the dimensions of 2 by 16 by 24, resulting in small bisection bandwidth. Even though with this limitation, the techniques presented in this paper improve the GPU performance by 25%. In the rest of this section, we will mainly focus on the results on CPU allocation, which is much bigger and has better bisection bandwidth.

Number of nodes	128	256	512	768	950
Initial GPU	427.7	236.5	132.5	98.1	85.9
Optimized GPU	355	189	117.7	77.5	70.8

TABLE VII
100M-ATOM SIMULATION XK6 PERFORMANCE (MS/STEP)

B. Scaling Results on Jaguar XK6 vs. XT5

In this section, we present NAMD performance using three different sized atom system benchmarks (increasing order) on the XK6, and compare some of the results with the Jaguar XT5 reported in [12]. The Jaguar XT5 system had 224,256 cores (two 2.6GHz hex-core AMD Opteron per node) and a 3D-torus SeaStar2+ interconnect. As mentioned before, Jaguar XK6 has 298,992 cores (one 2.1GHz sixteen-core AMD Interlagos per node) with Gemini interconnect. It also has 960 GPU nodes. Single core performance of Jaguar XK6 is slower than Jaguar XT5. However, the interconnect of XK6 has lower network latency and higher bandwidth.

Figure 7 shows the results of running DHFR (Section II-C3) with three versions of NAMD - MPI-based, uGNI baseline and uGNI optimized versions. On a smaller number of cores, the performance is similar. As the number of cores increases,

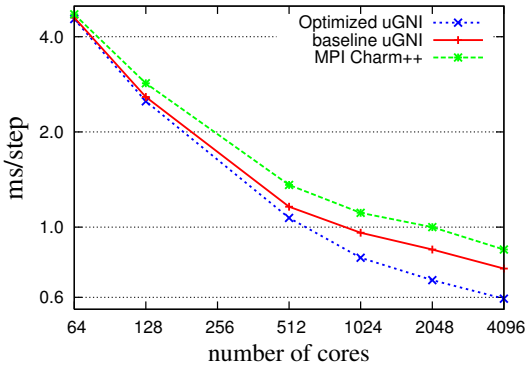


Fig. 7. NAMD DHFR CPU performance on TitanDev

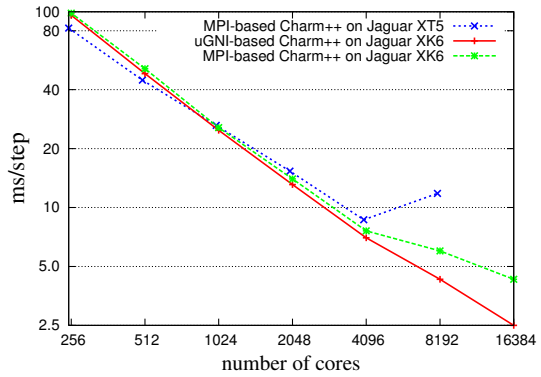
the uGNI version starts to outperform the MPI-based NAMD. Even the baseline version has the improvement by 18% over the MPI-based version. Furthermore, the optimization techniques for fine-grained communication improve the benchmark time farther to $0.65\,ms$ on 2048 cores, which is 54% improvement comparing with the MPI-based NAMD.

The performance of running 1M STMV system on Jaguar XK6 is shown in Figure 8(a), compared with the results on Jaguar XT5 [12]. It can be seen that on XT5, NAMD stops scaling beyond 4096 cores, while NAMD on XK6 keeps scaling to $16,384$ cores. Meanwhile, the performance is improved from $8.6\,ms$ per step to $2.5\,ms$/step, which is significant. The improvement from MPI-based NAMD on XT5 to MPI-based NAMD on XK6 is largely due to the upgrading to Gemini interconnect. The optimizations on the fine-grained communication in the uGNI-based CHARM++ also play a significant role.

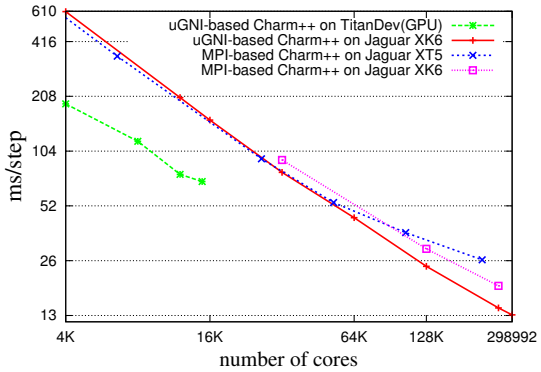
In Figure 8(b), we present the results of NAMD running the 100M-atom system on Jaguar XK6, and compared to Jaguar XT5. We first compared the performance of the MPI-based NAMD on XK6 and XT5. On smaller number of nodes, Jaguar XT5 has better performance. This is because XT5 has faster cores and program execution time is largely dominated by the computation time. When the number of cores increases to beyond 65K cores, XK6 starts to outperform XT5 by as much as 39% for same number of cores. Compared at full machine scale, MPI benchmark time on XK6 is $18.9\,ms$/step, which is significantly better than the $26\,ms$/step on XT5.

When using the native uGNI machine layer of CHARM++ for XK6 system, NAMD outperforms the MPI-based version for all the experiments we carried out. The best performance when running the 100M-atom system on the full Jaguar XK6 machine ($298,992$ cores) with the uGNI-based NAMD is $13\,ms$/step, which is a 32% improvement from the MPI-based version. Compared to the $26\,ms$/step best performance on Jaguar XT5, this is a significant improvement of 100%. In summary, the performance improvement of Jaguar XK6 v.s. XT5 is largely due to the hardware upgrade and the optimization techniques presented in this paper.

The performance of 100M atoms running on GPU is also illustrated in Figure 8(b). Good speedup is observed up to the full GPU allocation currently installed on the TitanDev machine. Compared with the CPU results on same number of nodes, using GPU boosts the performance by about 3.2



(a) 1M STMV



(b) 100M STMV

Fig. 8. Timestep (ms/step) of NAMD simulating 1M and 100M STMV on Jaguar XT5 and XK6

times. Because GPU allocation is of size 2 by 16 by 24, the bisection bandwidth is constrained by the narrow shape, which decreases the NAMD performance with GPUs.

VI. Related Work

Similar highly scalable MD codes, such as Blue Matter [3], Desmond [1], Amber [7] and GROMACS [9] have not yet demonstrated strong scaling results for molecule systems larger than $1x10^7$ on over $200,000$ cores as in this paper, though the latter two have reported good GPGPU results on smaller systems on smaller machines. Impressive work has recently been done [22] optimizing FFT performance in a similar context on Anton, however unlike that work ours is focused on commodity hardware.

MPI is the dominant paradigm for distributed parallel programming and provides an excellent foundation for optimization efforts. Compiler based approaches, such as OMPI [16], similarly Friedley and Lumsdaine describe a compiler approach, producing a 40% improvement, by exploiting one-sided communication via transformation of MPI calls [4]. Similar work exploits one-sided communication within the Partitioned Global Address Space (PGAS) languages like Chapel [2], UPC [20], Global Arrays [14], and Co-Array FORTRAN [15] with some preliminary Gemini work using the DMAPP API[21].

Unlike compiler approaches, our strategy is based on dynamic runtime system optimizations, most of which (excluding the persistent message scheme) require no changes to application source code. The Charm++ approach differs from the above PGAS languages in its migratable object-based virtualization strategy, which is leveraged for adaptive load balancing, critical path, and fault tolerance optimizations.

OpenMP/MPI is the dominant paradigm for hybrid programming and has been applied to MD codes, such as AMBER and GROMACS. However, loop based parallelization in the OpenMP model is fundamentally an SPMD bulk synchronous approach, in that it is rarely effective to have more than one type of task executing at the same time. In contrast, the Charm++ loop parallelization approach is designed to interleave seamlessly with other computations.

VII. Conclusion and Future Work

This paper focused on how to tune NAMD, a fine-grained communication intensive molecular dynamics simulation application, on Cray XK6 supercomputers. In particular, the paper addressed the scaling challenges in PME communication of NAMD. Our approach to optimizing NAMD on Cray XK6 explored multiple facets of this platform. We illustrated that significant performance gains can be achieved by utilizing the uGNI API to access multiple aspects of the Gemini interconnect. A performance analysis and visualization tool was extended to help identify the fine-grained communication bottleneck. Significant scaling improvements were demonstrated for the PME phase using several techniques. A new implementation of loop-level intra-node parallelism was applied to the PME phase, and was shown to improve performance when it is bounded by parallelizable communication overhead. We also demonstrated the streaming optimization technique for GPUs.

On the small portion of GPU allocation that is currently available on TitanDev, the performance of NAMD running Apoa1 benchmark is improved by 36% after applying the optimization techniques. On the full Jaguar XK6 machine (CPU-only partition), we were able to achieve 13 ms/step with parallel efficiency of 64%. It is twice as fast as the best performance of 26 ms/step we achieved on full Jaguar Cray XT5.

Future work will consider network topology-aware layout of PME tasks, as our work in that area had inconclusive results at the time of writing due to several challenges found on Cray XK6, including the fact that the job scheduler does not allocate contiguous nodes. Furthermore, the performance challenges of PME are theoretically circumventable via the adoption of alternative algorithms, such as multi-level summation [8].

Acknowledgments

This work was supported in part by a NIH Grant PHS 5 P41 RR05969-04 for Molecular Dynamics, by NSF grant OCI-0725070 for Blue Waters deployment, by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign and by DOE ASCR grant DE-SC0001845 for system software (HPC Colony). This research used resources of the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC05-00OR22725.

REFERENCES

[1] 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. Molecular dynamics—scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 84, New York, NY, USA, 2006. ACM Press.

[2] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[3] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, and M. C. Pitman. 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, 2006. ACM Press.

[4] A. Friedley and A. Lumsdaine. Communication optimization beyond mpi. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2018 –2021, may 2011.

[5] M. Frigo and S. Johnson. Fftw: an adaptive software architecture for the fft. *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, 3:1381–1384 vol.3, May 1998.

[6] H. Gahvari and W. Gropp. An introductory exascale feasibility study for ffts and multigrid. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–9, April 2010.

[7] A. W. Gtz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born. *Journal of Chemical Theory and Computation*, 0(0):null, 0.

[8] D. J. Hardy, J. E. Stone, and K. Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *Journal of Parallel Computing*, 35:164–177, 2009.

[9] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, 2008.

[10] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[11] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[12] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[13] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. In *TeraGrid'10*, number 10-13, Pittsburgh, PA, USA, August 2010.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996. 10.1007/BF00130708.

[15] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24:4–17, August 2005.

[16] H. Ogawa and S. Matsuoka. Ompi: Optimizing mpi programs using partial evaluation. *SC Conference*, 0:37, 1996.

[17] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[18] 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*, pages 1–18, Baltimore, MD, September 2002.

[19] 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.

[20] T. S. Tarek El-Ghazawi, William Carlson and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[21] A. Vishnu, M. ten Bruggencate, and R. Olson. Evaluating the potential of cray gemini interconnect for pgas communication runtime systems. In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 70 –77, aug. 2011.

[22] 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*, pages 1–11, New York, NY, USA, 2009. ACM.