Achieving Strong Scaling with NAMD on Blue Gene/L

Sameer Kumar[1], Chao Huang[2], Gheorghe Almasi[1], Laxmikant V. Kalé[2]

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

[2]University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{chuang10, kale}@cs.uiuc.edu

Abstract

NAMD is a scalable molecular dynamics application, which has demonstrated its performance on several parallel computer architectures. Strong scaling is necessary for molecular dynamics as problem size is fixed, and a large number of iterations need to be executed to understand interesting biological phenomenon. The Blue Gene/L machine is a massive source of compute power. It consists of tens of thousands of embedded Power PC 440 processors. In this paper, we present several techniques to scale NAMD to 8192 processors of Blue Gene/L. These include topology specific optimizations, new messaging protocols, load-balancing, and overlap of computation and communication. We were able to achieve 1.2 TF of peak performance for cutoff simulations and 0.99 TF with PME.

1 Introduction

Molecular dynamics simulations of biomolecules, based on classical mechanics, are extremely useful in understanding the function of assemblages of biomolecules such as proteins, DNA, cell membranes, and water molecules. Typical simulations involve 10,000 atoms to a few hundred thousand atoms, with a few exceptional situations (such as simulations of ribosome or an entire virus coat) requiring over a million atoms. With the number of particles to simulate relatively small, the granularity available for parallelization is rather small. For example, a single time-step of a 92,000 atom simulation of apolipo-protein A1 requires only about 10.4 seconds on single processor of IBM's Blue Gene/L machine, based on the embedded 440 core. Since millions of time-steps are needed in a simulation, parallel computing is needed to make large studies in practical amounts of time. But since only a few seconds must be parallelized over thousands of processors, the problem is quite challenging. This challenge is compounded by the fact that the number of atoms in a given protein is fixed, and so the simulation size does not significantly increase over the years. With faster and larger computers, the challenge is to parallelize the same computation at an ever finer granularity. In contrast, simulations of continuum models, such as those involved in weather modeling or structural dynamics, can often be solved at a finer resolution (using finer grids, for example) so one can keep the amount of work per processor constant as larger machines are deployed.

IBM's Blue Gene/L machine [7] represents a new design point among parallel machines. The entire machine consists of an unprecedented number of processors : 64k nodes each with two cores. Even smaller partitions that are practically deployed in various centers around the world may have thousands of processors. Each processor has a modest amount of memory (512MB for a dual-core node). The machine is also unique in its power consumption characteristics, and achieves high scalability as several processors can be packed in a relatively small region.

The challenge we explore in this paper is that of scaling the performance of NAMD, a molecular dynamics program widely used by biophysicists, to 8k processors on the Blue Gene/L machine.

NAMD is a C++ based parallel program, implemented using the Charm++ [8] parallel programming system. It uses object based decomposition and measurement based dynamic load balancing to achieve its high performance. As explained in Section 2, it uses a combination of spatial decomposition and force decomposition to generate a high degree of parallelism. NAMD is one of the fastest and most scalable program for biomolecular simulations that is routinely used in published simulations. In 2002, a paper describing its performance on the 750 node (3,000 processor) Lemieux machine at Pittsburgh supercomputing center, shared the Gordon Bell award [14].

Although NAMD demonstrated scalability by scaling a 320,000 atom simulation to 3000 processors with more than 1TF of peak performance [11] for cutoff simulations, it was clear that the code as it was did not scale beyond that. Further, on the now-standard 92,000 atom benchmark, it was able to scale performance only up to about 1024 processors before reaching saturation. Since then, its performance has been improved to some extent on other machines.

Molecular dynamics, however, has several advantages on the Blue Gene/L machine [7]. Despite the low processing power of the 440 embedded processor, it has a relatively large 4MB L3 cache. The torus network [1] has a relatively large bandwidth of 175MB/s on each of the six links. The network also has good bandwidth for messages a few KB of size [2], which are typical in NAMD. Moreover, the Blue Gene/L native operating system does not run any operating system *daemons* on the Compute Node Kernel [13]. On the Pittsburgh Lemieux machine, performance is hindered [12] when synchronization is needed in the order of the operating system quanta which was 10ms for the Tru-64 operating system.

For the above mentioned reasons, we believed that even the ApoA1 system could be scaled to several thousand processors on Blue Gene/L. Thorough a combination of techniques that involve machine specific optimizations as well as NAMD restructuring to generate more parallelism and to limit the Amdahl bottlenecks, we are able to scale the 92,000 atom simulation to 8k processors in co-processor mode and 4k processors in virtual node mode, while scaling the 327,000 atom simulation to 8k processors in both modes on the Blue Gene/L machine. One of the major problems with strong scaling on Blue Gene/L is the inability of the second on-chip core to work as a co-processor, due to the lack of cache-coherence. We address this problem through a novel technique of interleaving computation and communication, that achieves the same effect even in virtual node mode.

The next section describes the basic parallel structure of NAMD, including the parallelization strategy used. In section 3, we describe the suite techniques along with the improvements they generated. Final benchmark performance data and analysis are shown in Section 4. We illustrate the performance improvements throughout with visualizations obtained via *Projections*, the Charm++ performance analysis tool. We hope that this case study will also be useful for other applications aiming to attain high performance on Blue Gene/L.

2 NAMD Parallelization Strategy

The dynamic components of NAMD are implemented in the Charm++[10] parallel language. Charm++ implements an object-based message-driven execution model. In Charm++ applications, there are collections of C++ objects, which communicate by remotely invoking methods on other objects by messages. Compared with conventional programming models such as message passing, shared memory or data parallel programming, Charm++ has several advantages in improving locality, parallelism and load balance [9, 3]. The flexibility provided by Charm++ is a key to the high performance achieved by NAMD on thousands of processors.

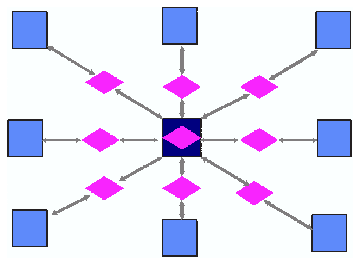


Figure 1. NAMD: Patches and Computes

In Charm++ applications, users decompose the problem into objects, and since they decide the granularity of the objects, it is easier for them to control the degree of parallelism. As described below, NAMD uses a novel way of decomposition that easily generates the large amount of parallelism needed to occupy thousands of processors.

Charm++'s object-based decomposition also help users to improve data locality. Objects encapsulate states, and Charm++ objects are only allowed to directly access their own local memory. Access to other data is only possible via asynchronous method invocation to other objects. Charm++'s parallel objects and data-driven execution adaptively overlaps communication and computation and hide communication latency: when an object is waiting for some incoming data, entry functions of other objects with all data ready are free to execute. In Charm++, objects may even migrate from processor to processor at runtime. Object migration is typically controlled by the Charm++ load balancer, described in Section 3.2.

NAMD 1 is parallelized via a form of spatial decomposition using cubes whose dimensions are slightly larger than the cutoff radius. Thus, atoms in one cube need to interact only with their 26 neighboring cubes. However, one problem with this spatial decomposition is that the number of cubes is limited by the simulation space. Even on a relatively large molecular system, such as the 92K atom ApoA1 benchmark, we only have 144 ($6 \times 6 \times 4$) cubes. Further, as density of the system varies across space, one may encounter strong load imbalance.

NAMD 2 addresses this problem with a novel combination of force [15] and spatial decomposition. For each pair of neighboring cubes, we assign a non-bonded force computation object, which can be independently mapped to any processor. The number of such objects is therefore 14 times (26/2 + 1 self-interaction) the number of cubes. To further increase the number and reduce the granularity of these compute objects, they are split into subsets of interactions, each of roughly equal work.

Figure 1 shows the different objects in NAMD. The spatially decomposed cubes, shown by *solid squares* are called *home patches*. Each home patch is responsible for distributing coordinate data, retrieving forces, and integrating the equations of motion for all of the atoms in the cube of space owned by the patch. The forces used by the patches are computed by a variety of *compute objects*, shown as *diamonds* in the gure. There are several varieties of compute objects, responsible for computing the different types of forces (bond, electrostatic, constraint, etc.). Some compute objects require data from one patch, and only calculate interactions between atoms within that single patch. Other compute objects are responsible for interactions between atoms distributed among neighboring patches.

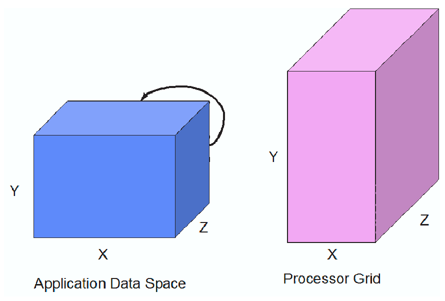
When running in parallel, some compute objects require data from patches not on the compute object's processor. In this case, a *proxy patch* takes the place of the home patch on the compute object's processor. During each time step, the home patch requests new forces from local compute objects, and sends its atom positions to all its proxy patches. Each proxy patch informs the compute objects on the proxy patch's processor that new forces must be calculated. When the compute objects provide the forces to the proxy, the proxy returns the data to the home patch, which combines all incoming forces before computing velocities and energies on each atom. The new atom coordinates are then computed and sent back to the proxies and the entire time step is repeated again. Thus, all computation and communication is scheduled based on priority and the availability of required data.

Some compute objects are permanently placed on processors at the start of the simulation, but others are moved during periodic load balancing phases. Ideally, all compute objects would be able to be moved around at any time. However, where calculations must be performed for atoms in several patches, it is more ef cient to assume that some compute objects will not move during the course of the simulation. In general, the bulk of the computational load is represented by the non-bonded (electrostatic and van der Waals) interactions, and certain types of bonds. These objects are designed to be able to migrate during the simulation to optimize parallel ef ciency. The non-migratable objects, including computations for bonds spanning multiple patches, represent only a small fraction of the work, so good load balance can be achieved without making them migratable.

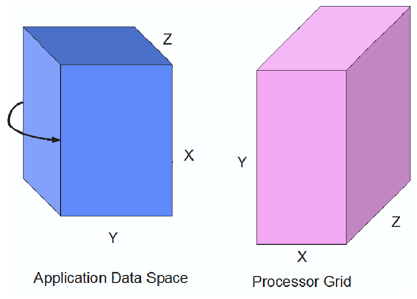
3 Blue Gene/L Optimizations

The Blue Gene/L machine at the Lawrence-Livermore national laboratory has 65536 dual core processor-chips connected by a 3D-torus interconnect [1]. However, these are slow embedded processors based on the Power-PC 440 core. They rely on the application to have massive scaling to achieve good performance on a large number of processors. Each core has a 32KB L1 cache and a shared 4MB on-chip L3 cache [7]. The large L3 cache is ideally suited for an application like NAMD as it has a small memory foot print. In fact, on a few hundred nodes, NAMD will mainly run from the L3 cache. The Blue Gene/L torus network also has good throughput for relatively small messages and the six outgoing links are ideal for the atom coordinate multicast.

However, we had to overcome several problems to get good performance with NAMD. Due the absence of DMA or a network interface controller, message passing has to be performed by the cores. Ideally one of the cores on the Blue Gene chip could have been used as a communication co-processor. But, due to the lack of cache coherence, the second core cannot effectively function as a communication co-processor. Strong scaling is easier to achieve with the overlap of computation and communication. Moreover, the bisection bandwidth of torus networks grows as $O(N^{2/3})$, and hence messages going over several hops are bound to have more bandwidth contention. So, applications

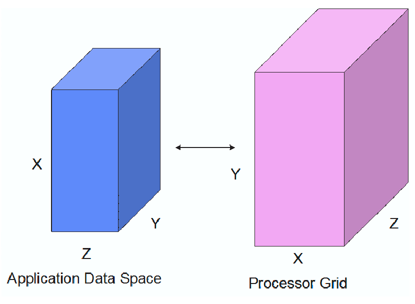


(a) Initial application and processor grids

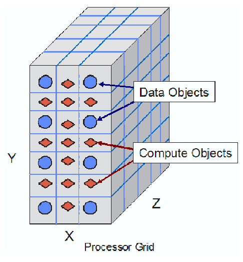


(b) Application grid after Z rotation

Figure 2. Patch allocation to processors



(a) Aligned application and processor grids



(b) ORB Allocation

Figure 3. Patch and compute allocation to processors

must localize communication to nearby processors in order to achieve good performance.

We also found it quite hard to get good sequential performance for NAMD on Blue Gene/L. The current performance numbers are based on compiling with just 440 instructions, thus not utilizing the dual FPU unit (double hummer) [4]. Even this performance was achieved after several weeks of hard work with the compiler team at IBM Toronto, to effectively software pipeline the inner loops of NAMD. We are still working on making use of the double hummer to further increase the single processor performance of NAMD. Even this sequential performance was quite hard to scale to 8k processors. We now present the techniques we used to overcome the above mentioned challenges and achieve good parallel performance for NAMD.

3.1 Problem Mapping on Blue Gene/L

To optimize communication performance on Blue Gene/L, the application messages should be localized on the torus. So, the mapping of patches to processors is critical. Before patches are allocated to processors, the axis of the application grid have to be mapped with the axis of the Blue Gene torus. This would allow the application communication load to be balanced along the three axis of the torus. We first sort the axis of the patch grid and the torus and then map the largest axis with each other. The application grid is then rotated to match the Blue Gene torus using the above *axis-map*.

We use an *orthogonal recursive bisection* (ORB) scheme to map patches to processors in NAMD. The ORB scheme first splits the patch grid into two partitions of similar load. The load of each patch is computed through a function that reflects the total computation and communication of that patch. The total computation of each patch is determined by the square of the number of atoms, while the communication is proportional to the number of atoms in that patch. The processor grid is then split in the same ratio as the two patch partitions along the corresponding axis, given by the *axis-map*.

This matching of patches to processors is shown by Figures 2(a), 2(b) and 3(a) (We only present the scenario where the processors are more numerous than patches.) The computation that is related to each patch is then placed on processors near that patch (Figure 3(b)). Typically, on Blue Gene, the Y and Z dimensions grow faster than the X dimension, while, for many NAMD problems, the X axis is the biggest. So such rotation is necessary. The axis mapping is stored in a persistent data structure and used for mapping computes and PME (See Section 3.3) objects.

3.1.1 Two-Away Computation

Patches created by the cutoff metric are quite large with several hundreds of atoms. This results in the integration of forces and computation of energies typically requiring 5-10ms to finish on the 440 processor. Moreover, the coordinate multicast from patches would be composed of large messages with tens of kilo-bytes of data. This may restrict the scalability of NAMD to a large number of processors.

To make patches more fine-grained, NAMD optionally supports *two-away* computation. Figure 4 illustrates splitting of patches along the X axis. The split results in more computes for the interactions between *neighbors* of the x-neighbors of each patch. The integrates are now twice smaller. The two-away computes are smaller than one-away computes as fewer atoms will now be in the cutoff. With objects of different sizes, we need a good load-balancer to allocate computes to processors (Section 3.2).

Two-away splitting of patches also makes the coordinate multicast messages smaller, but possibly to more destinations. We have observed that sending smaller messages to more destinations has higher bandwidth due to adaptive routing on the torus. The Blue Gene torus network also achieves a high bandwidth for fairly small messages. In

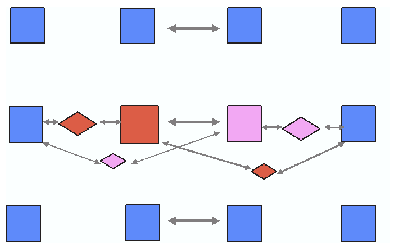


Figure 4. Two-Away splitting along the X-Axis

fact, with the APoA1 benchmark on 8k processors we had to split patches along all three dimensions X,Y and Z to achieve the best performance.

3.2 Load-balancing

NAMD uses a measurement-based load balancer, employing the Charm++ load balancing framework. When a simulation begins, patches are distributed using the ORB scheme introduced in Section 3.1. The framework measures the execution time of each compute object (the object loads), and records other (non-migratable) patch work as "background load." After the simulation runs for several time-steps (typically several seconds), the program suspends the simulation to trigger the initial load balancing. NAMD retrieves the object times and background load from the framework, computes an improved load distribution, and redistributes the migratable compute objects.

The initial load balancer is aggressive, starting from the set of required proxies and assigning compute objects in order from larger to smaller, avoiding creating new proxies unless necessary. To optimize performance on Blue Gene, we place an initial set of required proxies on the neighboring processors of each patch processor. These neighboring processors can compute the interaction of each patch with its neighboring patches. When two away computation is enabled, the initial proxy set also has proxies placed on the midpoints of a few neighboring two away patches.

The aggressive load balancer uses a greedy heuristic which tires to optimize the following metrics: 1) computation load, the total amount of work allocated to each processor, 2) number of proxies, this determines the connectivity of the atom coordinate multicast, and 3) communication hop-bytes.

Our *greedy* heuristic first allocates the heavy computes to processors. The compute is allocated to the lightest processor within *k* hops of the midpoint of the two patch processors corresponding to this compute. If the compute object overloads the above light processor more than an *overload_cutoff* of the average load, then the nearest acceptable processor is chosen. Preference is also given to processors which already have proxies for either or both home patches for that compute object.

After this initial balancing step, only small refinements are made, attempting to transfer single compute objects off of overloaded processors without increasing communication. Two additional cycles of load balancing refines try to reduce the *overload_cutoff*, by moving work away from the heavy processors. After these three phases, a refine is called every few thousand time steps to load-balance dynamic variance in processor loads from atom migrations.

3.3 Particle Mesh Ewald

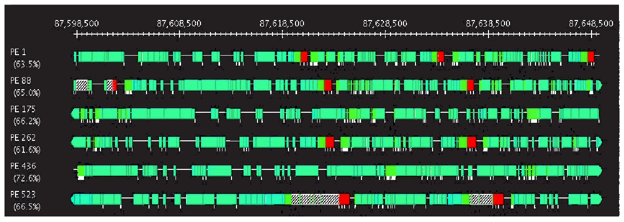
The cut-off computation presented so far does not account for the long range interactions between atoms. The Particle Mesh Ewald method [5] is used to compute the long-range electrostatic forces in NAMD. The parallelization of PME was first achieved in NAMD 2.2, and it has been further optimized to its current version in NAMD 2.6. PME requires two 3D *Fast-Fourier-Transform* operations. The FFTW library is currently used to do the serial work of the FFTs. The 3D-FFT operations are parallelized through a plane decomposition, where first a 2D FFT is computed on a plane of the grid. Then there is a global transpose after which the FFT is computed along the third dimension.

PME computation in NAMD involves five computation phases and four communication phases. The message driven NAMD/Charm++ framework can interleave these phases with force computation. The phases involve computing the PME charge grid in the patches. This is followed by communication of this grid to the PME objects, which then perform a forward 3D-FFT and a backward 3D-IFFT with three computation phases and two all-to-all transpose operations. Long-range forces are then sent back to the patches [14].

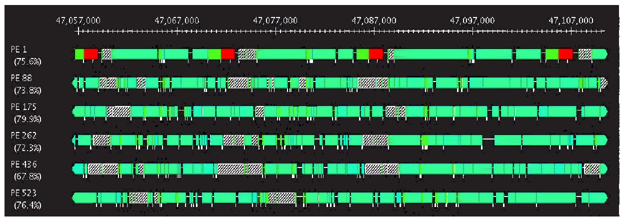
The plane decomposition restricts PME scaling to a few hundred processors (108 processors of the APoA1 system and 192 processors for the F1-ATPase benchmark). Fortunately, the total PME computation is a small fraction of the force computation. Moreover, with *multiple time stepping*, PME only has to be computed every few time-steps. We were still able to successfully scale NAMD with PME to 8k processors on Blue Gene/L, by **mapping** PME objects close to the patches with which they communicate.

3.4 Overlap of Computation and Communication

Overlap of communication with computation is an established technique to achieve strong scaling in applications. However, on the Blue Gene/L machine, overlap of computation and communication is hard to achieve because of the absence of cache coherence between the two cores



(a) No Overlap



(b) Overlap of communication with computation

Figure 5. NAMD Timeline

on a chip. Ideally, one of the cores could have been used as a communication co-processor to achieve this overlap. The co-processor mode [2] can achieve overlap for large messages, by flushing the L1 caches of the main CPU and the co-processor during messaging. But, such cache-flushes can hurt NAMD performance, resulting in no net gain.

However, we can take advantage of the fact that each torus FIFO has 4 packet buffers. At full link bandwidth of 175 MB/s, it takes about 4096 processor cycles to fill these buffers. A network poll of the six FIFOs only takes about 200 processor cycles with the new messaging layer developed by the authors. So we can interleave computation while the packets travel on the network. Moreover, with the complex communication pattern of NAMD, we can only achieve a total bandwidth of about two links, making each link less busy overall.

```
void CmiNetworkProgress() {
  new_time=rts_get_timebase();
  if(new_time<lastProgress+PERIOD)
    return;
  lastProgress = new_time;
  AdvanceCommunications();
}
//NAMD force compute loop
for (i=0;i<i_upper;++i){
  CmiNetworkProgress();
  const CompAtom &p_i=p_0[i];
  //Compute Pairlists
  for (k=0; k<npairi; ++k)
    //Calculate forces ........
}
```

To enable progress every few thousand cycles, *CmiNetworkProgress()* is called from the outer compute loop in NAMD. When PERIOD clock cycles have elapsed *CmiNetworkProgress()* calls *AdvanceCommunication()* which makes flushes the packets from the torus. The performance gains from overlap of communication and computation on 4096 processors is shown in Figures 5(a) and 5(b).

In the no-overlap case the application waits for packets to arrive, which results in the black regions in the timeline (Figure 5(a)). When communication is overlapped with computation, the packets stream in while the processor computes, minimizing the black regions (Figure 5(b)). Observe that the average utilization of the processors goes up from about 66% to 74%.

3.5 Communication Optimizations

Blue Gene/L nodes are organized into 3-dimensional torus network for point-to-point communication, with a bandwidth of 175MB/s. It is important for NAMD to have highly efficient communication operations to take full advantage of the high network bandwidth. For instance, at the end of the integration phase in NAMD, each patch sends out the updated atom information to all the computes whose computation depends on it (See Figure 1). Correspondingly, there is a subsequent reduction with which the computes contribute the force data back to the patch. These communication operations could cause excessive amounts of overhead both in the processor and in the network. We have experimented with a variety of communication optimizations, such as improved *FIFO mapping* schemes and new messaging protocols to solve these problems. We discuss some of the optimizations in this section.

FIFO mapping schemes: each Blue Gene/L node has several outgoing and incoming FIFOs [1]. A FIFO pinning function maps an outgoing packet with destination displacement $\{dx, dy, dz\}$ onto one of the 6 sending FIFOs. Our goal is to improve the overall utilization of the FIFOs with a better FIFO mapping scheme. Figure 6 is a simplified 2-D illustration of our pinning schemes. The Naive Scheme, which prefers X-FIFOs, makes a dramatically unequal division of the space and is incapable of producing a balanced mapping. The Cone Scheme divides the space into 6 equal portions and assigns each to a FIFO. However, this scheme can still suffer from load imbalance in the neighborhood multicast in NAMD, where it is not uncommon for the neighborhood to have an elongated shape along one dimension, for example, when the two-away splitting is enable (See Section 3.1.1 and Figure 4). To mend this situation,

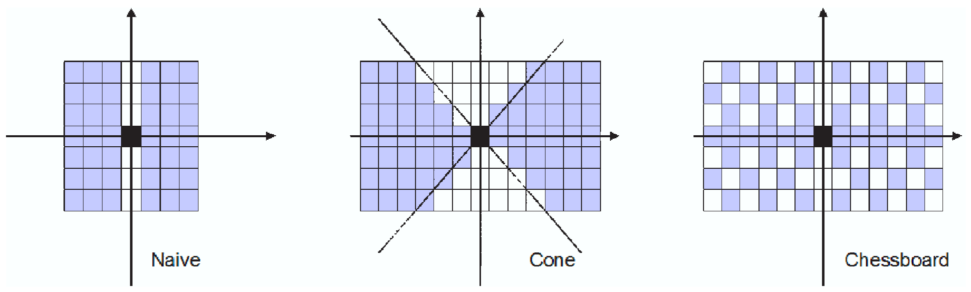


Figure 6. FIFO Mapping Schemes

we applied the Chessboard Scheme which ensures balanced mapping even in the presence of a disproportional neighborhood. A destination coordinate $\{dx, dy, dz\}$ is simply mapped to FIFO $(dx + dy + dz)\%6$. Finally, on top of the above static mapping schemes, a fully adaptive FIFO pinning function, which picks an available FIFO dynamically, is implemented and offers the biggest performance improvement. We call this scheme **dynamic FIFO** mapping.

Messaging protocols: on the Blue Gene/L torus Network, messages can have different routing protocols to suit various communication patterns. For example, short messages from reductions use eager protocol, while long data messages use rendezvous protocol. However, there is a rendezvous round-trip setup overhead. We designed a new message protocol called **active-put** to avoid the rendezvous overhead by putting messages into a persistent buffer. In traditional one-sided communication put, the receiver has to verify the completion of the put either by polling on a flag or by a callback at sender side. Either of these incurs an unnecessary overhead. We avoid this overhead by having the put carry the message handler along, so that the handler can process the message locally on the destination. After using active-put in multicast operations, we noticed a performance improvement on large runs of several thousand processors.

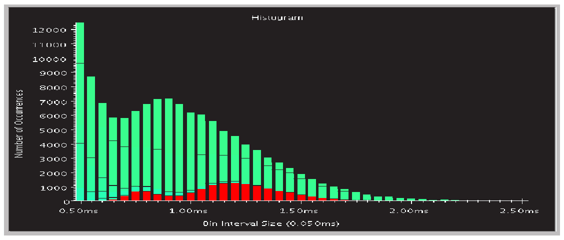
Topology optimizations: in both multicast and reductions, the root node sends or receives a large number of messages. Especially in large runs on several thousand processors, a multicast typically involves sending out 64 to 100 messages. A communication bottleneck occurs and it can hold up the node for unnecessarily long period of time. Also, switching back and forth between computation and the network progress engine could result in undue overhead. We addressed this issue by organizing the neighbors into a spanning tree so that the communication hot-spots are offloaded. In addition, we designed a topology-based spanning tree for the 3-D torus network to minimize the total hop-byte counts.

Congestion control: the Blue Gene/L MPI enforces congestion control through a message pacing mechanism that allows only a fixed sized window of outstanding packets from any message. For example, assuming the window size is 16, when we have 32 large messages to send, MPI will send 16 packets from each of the 32 messages and then wait for their acknowledgments. This works fine with MPI messages, yet it could hurt the efficiency of NAMD. This is because Charm++ uses *active messages*, where each message has some computation associated with it on the receiver. Thus, ensuring the prompt arrival and processing of full messages is more critical than packet-level congestion control. In Charm++, we devised a tunable parameter for the maximum number of outstanding send requests. Only packets from messages in the window will be sent out and the window advances when a full message has been dispatched. This message level congestion control improves efficiency as well as fairness of communication in NAMD.

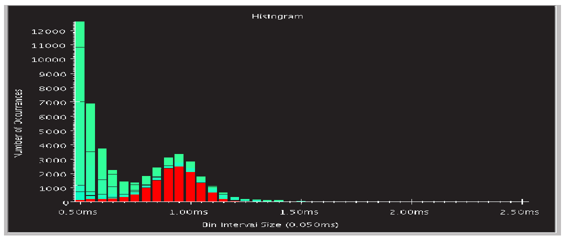
3.6 Grain-size Analysis

Grain-size is critical to NAMD scaling, as a course grain can restrict scaling and performance. The minimum step time is at least the size of the largest object. For good load balancing, mean computation of each object should be five to ten times smaller than the step-time. Having a very fine grain on the other hand, will generate too many objects, resulting in higher object context switching and runtime overheads. So, we use a grain size which is typically a few times less than the target time-step, tunable at runtime.

Figure 7 shows the histograms of the NAMD APoA1 benchmark objects with more than 0.5ms computation, from two 4096 processor runs with different grain sizes. Observe that in Figure 7(a) there are quite a few large objects with more than 1ms of computation. However in Figure 7(b), there are very few such objects.Infact, the latter grain size is responsible for the 4.3 ms time-step presented in Section 4.



(a) bad coarse grain-size



(b) improved finer grain-size

Figure 7. Histogram of object computation with NAMD APoA1 on 4096 processors

4 Performance Results

Before presenting the parallel performance of NAMD on a large number of processors, we first quantify the gains of many of the optimizations presented in the previous section. Table 1 shows the performance improvements as our techniques are implemented step by step. However, the performance improvement of each technique may depend on the order in which optimizations were used. Here, we present an arbitrary order of the optimization techniques.

The performance improvement of NAMD version-2.6 over version-2.5 mainly comes from a serial speedup of

NAMD v2.5	v2.6 Blocking	Fine grained
40	25.2	24.3
Congestion control	New LDB	Chessboard
20.5	14.0	13.6
Dynamic Mapping	Fast memcpy	Non-blocking
13.5	13.3	11.9

Table 1. Improving Cutoff Performance (ms) of the ApoA1 on 1024 processors

#Procs	MPI	Msglayer	
		Blocking Send	Non-Blocking
512	27.8	26.7	23.7
1024	17.3	14.4	13.8
2048	10.2	9.7	8.6
4096	7.3	6.8	6.2

Table 2. PME Performance (ms) of ApoA1

#Procs	Time/step(ms)	Speedup	GFLOPS
4	2200	4	1.31
512	21.7	406	133
1024	11.9	739	242
2048	7.3	1205	394
4096	4.3	2047	695
8192	3.5	2514	871

Table 3. ApoA1 Cutoff Performance

#Procs	Time/step(ms)	Speedup	GFLOPS
4	2611	4	1.23
512	23.7	441	135
1024	13.8	757	233
2048	8.6	1214	373
4096	6.2	1685	536
8192	5.2	2008	651

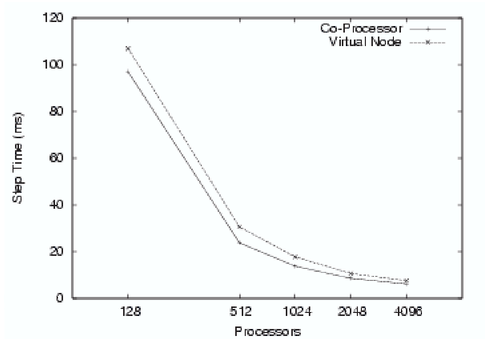
Table 4. ApoA1 PME Performance

#Procs	Time/step(ms)	Speedup	GFLOPS
32	1120	32	7
512	73.9	485	113
1024	39.2	914	219
2048	21.8	1644	393
4096	11.8	3037	726
8192	7.1	5048	1210

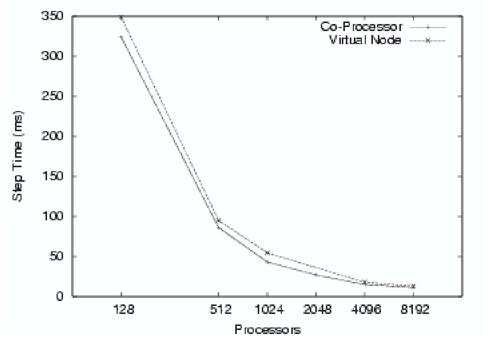
Table 5. ATPase Cutoff Performance

#Procs	Time/step(ms)	Speedup	GFLOPS
32	1228	32	7
512	86.2	456	105
1024	43.2	910	224
2048	26.9	1461	360
4096	15.1	2602	641
8192	9.8	4010	991

Table 6. ATPase PME Performance



(a) ApoA1 Benchmark



(b) ATPase Benchmark

Figure 8. Virtual Node Mode Performance

about 40%, due to a higher degree of software pipelining and the elimination aliasing problems in the inner compute loops. In Table 1, the new topology aware load balancer has the biggest parallel performance gains.

Table 2 presents a performance comparison of NAMD using MPI and NAMD on top of a native message layer developed by the authors. Currently MPI only supports a static FIFO mapping scheme for point-to-point messages. Moreover, non blocking communication in the MPI version of NAMD has poor performance as it uses expensive MPI_test and MPI_Iprobe calls while calling progress. So the MPI version uses blocking communication. The table compares MPI, with the native blocking and non-blocking versions respectively. On 4096 processors, the native non-blocking version does better than MPI by about 18%.

Tables 3, 4, 5 and 6 show the time per step, speedup and GFLOPS reached on scaling runs, for both the 92K atom ApoA1 benchmark and the 327K atom ATPase benchmark, using the non-blocking message layer. Compared with the previous results presented [14], the new results show major improvement. For the ApoA1 benchmark, the time per step is further reduced to 3.5 ms at 871 GFLOPS for cutoff and 5.2 ms at 651 GFLOPS with PME. The larger simulation of the ATPase benchmark has better scaling to 8192 processors. It achieves 1.2 TFLOPS with cutoff and 0.99 TF with PME. The limited parallelism of the plane decomposition based PME appears to have become an Amdahl bottleneck.

So far we have only presented NAMD performance in co-processor mode. When both cores on the BGL chip are used for computation, its termed as *virtual node mode*. However, now both cores have to share memory and network bandwidths. Figures 8(a) and 8(b) compare NAMD performance in the two modes for ApoA1 and ATPase respectively. Observe that virtual node mode only has a slightly lower performance than co-processor mode. In fact, most of the optimizations presented in Section 3 are applicable to both co-processor mode and virtual node mode.

5 Related Work

Blue Matter [6] is another application that has achieved strong scaling and good parallel performance with classical molecular dynamics on Blue Gene/L. This application is designed specifically for the Blue Gene/L machine. It also uses a spatial decomposition scheme like NAMD and makes very good use of hardware collective support from the BGL network architecture.

We have kept NAMD general and use Blue Gene specific optimizations through abstractions in Charm++ and the Blue Gene message layer. We also use different parallelization techniques and algorithms from Blue Matter. NAMD derives its scaling and performance mainly from efficient load-balancing and overlap of computation and communication on a variety of communication architectures.

Our performance results for the APoA1 system are comparable to Blue Matter (in some cases even better) till 8k nodes. We are still to experiment with 16k nodes. In this paper, we also present the performance scaling on a large problem like ATPase, which was possible in virtual node mode as NAMD is quite memory efficient.

6 Summary and Future Work

We presented the several performance optimization schemes that were used to achieve good parallel performance for NAMD on Blue Gene/L. We scaled NAMD with PME on 8192 processors for both APoA1 and ATPase. The floating point performance of ATPase on 8192 processors is 1.2TF for cutoff and 0.99TF for PME, with speedups of 5048 and 4090 respectively. This impressive performance can be attributed to the novel mechanism to achieve overlap of communication with computation, along with topology aware load balancing; while communication performance was optimized by the persistent active-put protocol, message based congestion control and dynamic FIFO mapping.

We still have several challenges ahead of us. The current floating point performance is less than expected. The inner compute loops do not utilize the double hummer. The inner loops compute on data structures that are of size 24 bytes, and hence hard to program with the double hummer which requires 16 byte alignment. Moreover, the PME and Energy compute loops have register spills due to the limited number of registers in the processor architecture. We need to devise techniques to address the above mentioned serial performance issues. We are investigating a pencil based PME scheme, which has three transposes and much more parallelism than the current plane based scheme. The third transpose however, may make the pencil based scheme unsuitable for smaller number of processors. But on a large number of processors its gains will be evident. We are also in the process of developing asynchronous non-blocking interfaces for the hardware collectives on Blue Gene/L. The full potential of this machine is not utilized without the use of its hardware collective support. For example, we intend to optimize the coordinate multicast in NAMD through row and column broadcasts.

Acknowledgments : We would like to acknowledge the various students and staff of the Parallel Programming Laboratory and the Theoretical and Computational Biophysics Group at the University of Illinois, for their help in developing NAMD. In particular, we would like to thank Gengbin Zheng and Jim Phillips. We are grateful for the funding provided by the National Institute of Health (NIH PHS 5 P41 RR05969-04). The NAMD performance runs were carried out at the BlueGene Watson machine. We would like to thank Fred Mintzer, Jose Castanos, Bob Walkup, Nathmuni Ramanujam and Dave Singer for providing us with computer time and technical support.

References

[1] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2/3):265–276, 2005.

[2] G. Almasi, C. Archer, J. G. Castanos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, 2005.

[3] R. K. Brunner and L. V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[4] S. Chatterjee, L. R. Bachega, P. Bergner, K. A. Dockser, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. Mendell, R. Nair, C. D. Wait, T. J. C. Ward, and P. Wu. Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L. *IBM Journal of Research and Development*, 49(2/3):377–391, 2005.

[5] T. Darden, D. York, and L. Pedersen. Particle mesh Ewald. An N·log(N) method for Ewald sumsin large systems. 98:10089–10092, 1993.

[6] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. C. Ward, M. Giampapa, Y. Zhestkov, M. C. Pitman, F. Suits, A. Grossfield, J. Pitera, W. Swope, R. Zhou, R. S. Germain, and S. Feller. Blue Matter: Strong Scaling of Molecular Dynamics on Blue Gene/L. *IBM Research Technical Report RC3688*, 2005.

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

[8] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[9] L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[10] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[11] 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.

[12] D. J. Kerbyson, F. Petrini, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Perform ance on the 8,192 Processors of ASCI Q, November 2003.

[13] J. E. Moreira, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castanos, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2/3):367–376, 2005.

[14] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[15] S. J. Plimpton and B. A. Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. *"J. Comp. Chem"*, 17(3):326–337, 1996.