Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications

Abhinav Bhatelé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
bhatele@illinois.edu

Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
kale@illinois.edu

Sameer Kumar
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598
sameerk@us.ibm.com

ABSTRACT

Molecular Dynamics applications enhance our understanding of biological phenomena through bio-molecular simulations. Large-scale parallelization of MD simulations is challenging because of the small number of atoms and small time scales involved. Load balancing in parallel MD programs is crucial for good performance on large parallel machines. This paper discusses load balancing algorithms deployed in a MD code called NAMD. It focuses on new schemes deployed in the load balancers and provides an analysis of the performance benefits achieved. Specifically, the paper presents the technique of topology-aware mapping on 3D mesh and torus architectures, used to improve scalability and performance. These techniques have a wide applicability for latency intolerant applications.

Categories and Subject Descriptors

C.4 [**Performance of Systems—Performance attributes**]; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—network topology

General Terms

Algorithms, Performance

Keywords

load balancing, mapping, topology, torus, mesh

1. INTRODUCTION

Accurate understanding of biological phenomena is aided by bio-molecular simulations using Molecular Dynamics (MD) programs [1, 2, 3]. MD programs involve simulation of millions of time steps, where each time step is typically 1 femtosecond (10^{-15} secs.) Simulating a millisecond in the life of a biomolecule can take years on a single processor. Hence it becomes imperative to parallelize MD programs. Parallelization of MD programs involves atom, spatial or force decomposition [4]. Atom and force decomposition are not scalable in the sense of their isoefficiency [1]. Spatial decomposition is scalable but creates load balancing issues because of relatively non-uniform distribution of atoms (especially the variation in density between water and protein.) It becomes a challenging problem because atoms migrate as the simulation proceeds and because the force computation has heterogeneous components.

In this paper, we describe load balancing algorithms deployed in a highly scalable MD code called NAMD [5, 6]. NAMD is written using the CHARM++runtime framework [7] and benefits from its load balancing support. Load-balancing in NAMD is measurement-based and dynamic [8]. A few hundred time steps are instrumented to record the time spent by each object doing useful work. This instrumented data enables the load balancer to make decisions for the next phase. Two load balancers are used in NAMD: a *comprehensive* load balancer which is used once at startup and moves lots of objects around, and a *refinement* load balancer which is used in the subsequent steps and refines the load balancing decisions from the previous phases and tries to bring the maximum load on any processor closer to the average.

This paper describes the existing load balancing infrastructure and performance metrics to assess its effectiveness. Then, we present new techniques which have been deployed in the load balancers to improve load balance and performance. Specifically, this refers to adding architecture awareness to the load balancers. Many modern supercomputers today are three-dimensional (3D) meshes or tori with n-way SMP nodes. The application (and in our case, the load balancer) can utilize this information to map communicating tasks on processors which are physically close-by. This can reduce the distance traveled by messages on the network, thereby decreasing contention and message latencies and improving performance. Techniques outlined in this paper can be used by applications with similar communication patterns and by other latency intolerant applications also.

Traditional molecular dynamics codes which used atom or spatial decomposition divided the atoms or space to achieve load balance. The use of force or hybrid decomposition makes load balancing more difficult if the number of particles for which a processor does the force calculations is variable. NAMD has used the benefits of the CHARM++'s dynamic load balancing framework for a long time. In contrast codes such as Desmond and Blue Matter do an intelligent static initial distribution of the work and ignore load variations which might occur later. NWChem is a compu-

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

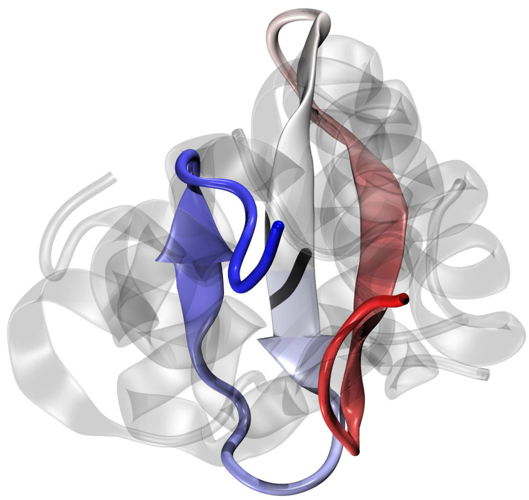

Figure 1: VMD visualization of the $30,591$-atom WW simulation using Namd

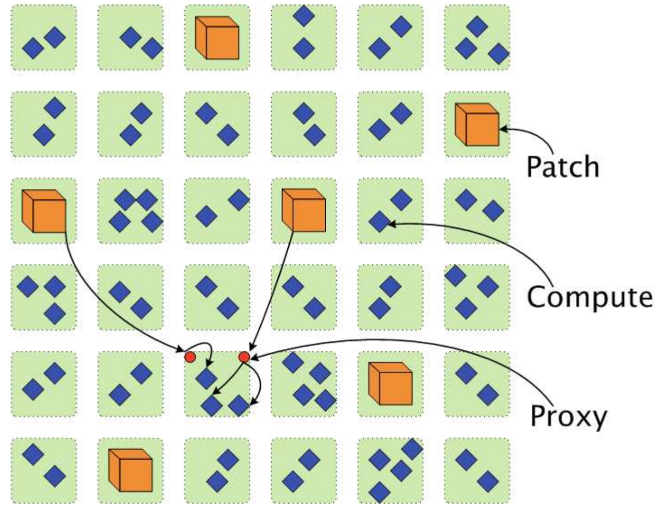

Figure 2: Placement of patches, computes and proxies on a 2D mesh of processors

tational chemistry software suite which developed sophisticated load balancing techniques to handle periodic atomic reassignments [9].

Significant work was done on developing general techniques for topology-aware mapping in the 80s [10, 11]. With the emergence of large machines with significant number of hops, researchers have started using such mapping techniques in their specific applications [12, 13, 14]. This paper presents topology aware techniques for molecular dynamics applications and points towards the need to develop more general techniques.

We present results on IBM Blue Gene/P (Intrepid) at Argonne National Laboratory (ANL) and Cray XT3 (BigBen) at Pittsburgh Supercomputing Center (PSC) to substantiate the schemes proposed above. Load balancing algorithms discussed in this paper are widely applicable and can be used in many scenarios, especially those involving multiple multicast sources and targets. Topology-aware schemes discussed here are beneficial for applications running on large supercomputers. They can help in minimizing message latencies and improving performance and scaling.

2. SCIENCE SIMULATIONS USING NAMD

Molecular dynamics (MD) simulations of protein folding offer insight that neither structural studies nor computational prediction of static structures can provide, as both structural and dynamic information on the folding process can be obtained. reach the timescales necessary for protein folding using all-atom, explicit solvent simulations. However, through a combination of advances in computing and discovery of new fast-folding mutants, it has recently become feasible to study the complete folding process of proteins through all-atom MD simulations [15] allowing direct comparison of simulation results with actual experiments. One early target for such simulations has been the WW domain (Figure 1), which was simulated for 10 microseconds (over twice the experimental folding time) using Namd. The protein failed to fold in the simulations, and follow-up efforts are currently underway to determine whether this failure is related to insufficient sampling or inaccuracies in the force field used. Similar simulations of another small protein, villin headpiece, using the name modified version of Namd have lead to several successful folding trajectories (unpublished data).

Complete folding simulations of these small proteins require trajectories of 5-10 μs in duration, at a time when most molecular dynamics papers report simulations of tens or hundreds of nanoseconds. When the WW domain simulations commenced, simulation rates of 42 ns/day (4.1 ms/step) were obtained on 184 processors of NCSA's Abe cluster. Improvements to the serial performance of Namd, including tuning of frequently used loops, allowed the simulations to reach 56 ns/day (3.3 ms/step). Improvements to the Namd load balancers for this 8-way SMP cluster and the particular atom simulation were made to allow efficient scaling to 329 Abe nodes, allowing the performance to reach an average of 101 ns/day (1.7 ms/step), enabling completion of a 10 μs simulation in less than four months. This points towards the criticality of optimizing performance of the load balancers and the direct impact it has on the science which is simulated using Namd.

3. LOAD BALANCING IN NAMD

Parallelization of Namd involves a hybrid of spatial and force decomposition. The 3D simulation space is divided into cells called "patches" and the force calculation between every pair of patches is assigned to a different "compute" object. *Patches* are assigned statically to processors during program start-up. On the other hand, *computes*, can be moved around to balance load across processors. If a patch communicates with more than one compute on a processor, a proxy is placed on this processor for the patch. The proxy receives the message from the patch and then forwards it to the computes internally (Figure 2). This avoids adding new communication paths when new computes for the same patch are added on a processor.

The number of computes on a processor and their individual computational loads determines its computational load and the number of proxies on a processor indicates its communication load. Load balancing in Namd is measurement-based. This assumes that load patterns tend to persist over time and even if they change, the change is gradual (referred to as the *principle of persistence*). The load balancing framework records information about object (compute) loads for some time steps. It also records the communication graph between the patches and proxies. This information is collected on one processor and based on the instrumentation data, a load balancing phase is executed. Decisions are then sent to all processors. The current strategy is centralized and we shall later discuss future work to make it fully distributed and scalable.

It should be noted that communication in NAMD is a special case of a general scenario. In NAMD, every patch multicasts its atom information to many computes. However, each compute (or target of the multicast) receives data from only two patches (the sources). The general case is where each target can receive from more than two sources and we shall see in Section 6 how the strategies deployed in NAMD can be extended to other cases.

Patches in NAMD are statically mapped in the beginning and computes are moved around by the load balancers to achieve load balance. Two load balancers are used in NAMD. An initial *comprehensive* load balancer invoked in the beginning places the computes evenly on all processors. A *refinement* load balancer is invoked multiple times during a run and it moves a small number of computes to rebalance the load. Both load balancers follow a greedy approach to distribute load evenly among the processors.

3.1 The Algorithms

The decision to place patches statically and load balance the computes is based on the typical number of patches and computes for a system. For a standard MD benchmark, 92,227-atom ApoLipoprotein-A1 (ApoA1), the number of patches and computes in a typical run on 512 cores is 312 and 22212 respectively. Also, atoms in a patch move slowly and relative density of atoms per patch does not change much as there is no vacuum inside patches – unlike large density variations we see in out cosmology applications, for example. Hence, we do not need to load balance the patches. Atoms are migrated from one patch to another on after every 20 time steps.

Static Placement of Patches: The 3D simulation space is divided into patches using a geometric decomposition to have roughly equal number of atoms in each patch. These patches are then assigned to a subset of the processors in a simple round-robin or strided fashion. In a typical highly parallel run, the number of patches is significantly smaller than the number of processors.

Comprehensive Strategy: This algorithm iterates over the list of all computes in decreasing order of their computational loads and finds a "suitable" processor for each one, while minimizing load imbalance. A compute is placed on a processor only if the new load of the processor remains below a threshold value (set to be some factor of the average load on all processors). It also tries to minimize communication by avoiding the creation of new proxies (additional proxies require new communication paths from a particular patch to the processor on which a new proxy is being placed). Keeping in mind that each compute communicates with two patches, following are the steps in the search of a "suitable" processor for a compute:

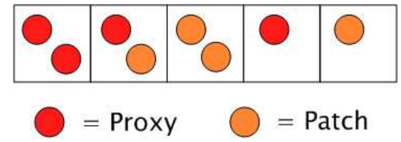

Figure 3: Preference table for the placement of a compute

Step I: Place the compute on an underloaded processor which hosts both the patches or proxies for both of them - this does not add any new communication paths to the graph.
Step II: If Step I fails, place the compute on an underloaded processor which hosts at least one patch or proxy for one of the patches - this requires adding one path for the other patch.
Step III: If both Step I and Step II fail, find the first underloaded available processor from the list of underloaded processors which can accept this compute.

To summarize the strategy, only underloaded processors are considered for placing a compute and among them, processors with available patches or proxies are given preference to minimize communication. This is implemented using a preference table which stores the least underloaded processors for different categories (Figure 3.) The first three cells in the table correspond to Step I and the last two correspond to Step II. The highest preference is given to a processor with proxies for both patches (cell 1), then to one with one of the patches and a proxy for the other (cell 2) and then to a processor with both patches on it (cell 3). If Step I fails, preference is first given to a processor with a proxy (cell 4) and then to one with the patch (cell 5). We give preference to placing computes on a processor with proxies compared to the patches themselves because it was observed that performance is better if the processors with patches are not heavily loaded.

Refinement Strategy: This is algorithmically similar to the comprehensive strategy. The difference is that it does not place all the computes all over again. This algorithm builds a max heap of over-loaded processors and *moves* computes from them to under-loaded processors. Once it has reduced the load on all overloaded processors to below a certain value, it stops. The process of choosing an underloaded processor on which to *move* a compute, is similar to that in the comprehensive strategy. The three steps outlined above for the search of a suitable processor are followed in order in this case also. For a detailed and historical perspective to the NAMD load balancers, read Kalé *et al.* 1998.

3.2 Metrics for Success

Optimal load balancing of objects to processors is NP-hard, so in practice, the best one can do is to try different heuristic strategies to minimize load imbalance. A combination of several metrics decides the success of a load balancer and we will discuss them now before we compare different load balancing strategies:

Computational Load: The most important metric which decides the success of a load balancer is the distribution of computational load across all processors. A quantity which can be used to quantify this is the ratio of the maximum to average load across the set of processors. A high *max-to-average* ratio points towards load imbalance.

Communication Volume: As we balance computational load, we should also aim at minimizing inter-processor communication. This can be achieved by using proxies, as described earlier, to avoid duplicate communication paths from a patch to a processor. Additionally, we need to minimize the number of proxies by avoiding the addition of new proxies.

Communication Traffic: Another optimization possible on non-flat topologies is to reduce the total amount of traffic on the network at any given time. This can be done by reducing the number of hops each message has to travel and thus reducing the sharing of links between messages. Number of hops can be reduced by placing communicating objects on nearby processors. This reduces communication contention and hence, the latency of messages. This will be the main focus of this paper. Communication traffic is quantified by the *hop-bytes* metric which is the weighted sum of the messages sizes where the weights are the number of hops traveled by the respective messages

4. TOPOLOGY-AWARE TECHNIQUES

Recent years have seen the emergence of large parallel machines with a 3D mesh or torus interconnect topology. Performance improvements can be achieved by taking the topology of the machine into account to optimize communication. Co-locating communicating objects on nearby processors reduces contention on the network and message latencies, which improves performance [16, 14]. Let us now see the deployment of topology-aware techniques in the static placement of patches and the load balancers.

Topology placement of patches: Since patches form a geometric decomposition of the simulation space, they constitute a 3D group of objects which can be mapped nicely onto the 3D torus of machines. An ORB of the torus is used to obtain partitions equal in number to the patches and then a one-to-one mapping of the patches to the processor partitions is done. This is described in detail in [17]. This idea can be used in other applications with a geometric decomposition such as cosmological and meshing applications.

Topology-aware Load Balancers: Once patches have been statically assigned onto the processor torus, computes which interact with these patches should be placed around them. We will now discuss modifications to the load balancing algorithm that try to achieve this heuristically. The three steps for choosing a suitable processor to place a compute on (for both the comprehensive as well as refinement load balancer) are modified as follows:

Step I: If the compute gets placed on a processor with both the patches, then no heuristic can do better than that because both messages are local to the processor (and no new communication paths are added). However, if we are searching for a processor with proxies for both patches, we can give topological preference to some processors. Consider Figure 4

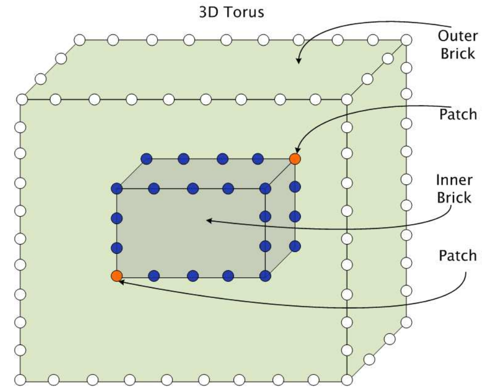

Figure 4: Topological placement of a compute on a 3D torus/mesh of processors

which shows the entire 3D torus on which the job is running. When placing a compute, it should be placed topologically close to the two processors that house the patches it interacts with. The two patches define a smaller brick within the 3D torus (shown in dark grey in the figure). The sum of distances from any processor within this brick to the two patches is minimum. Hence, if we find two processors with proxies for both patches, we give preference to the processor which is within this inner brick defined by the patches.

Step II: Likewise, in this case too, we give preference to a processor with one proxy or patch which is within the brick defined by the two patches that interact with the compute.

Step III: If Step I and II fail, we are supposed to look for any underloaded processor to place the compute on. Under the modified scheme of things, we first try to find an underloaded processor within the brick and if there is no suitable processor, we spiral around the brick to find the first underloaded one.

To implement these new topology-aware schemes in the existing load balancers, we build two preference tables (similar to Figure 3) instead of one. The first preference table contains processors which are topologically close to the patches in consideration (within the brick) and the second one contains the remaining processors (outside the brick). We look for underloaded processors in the two tables with preference in order to the following: number of proxies, hops from the compute and then the load on the processor.

4.1 Performance Improvements

Performance runs were done to validate the theoretical basis behind the topology-aware schemes discussed in the paper. Two supercomputers were used for this purpose: IBM Blue Gene/P (Intrepid) at ANL and Cray XT3 (BigBen) at PSC. The default job scheduler for XT3 does not guarantee a contiguous partition allocation and hence those runs were done with a special reservation on the whole machine.

Figure 5 shows the *hop-bytes* for all messages per iteration when running NAMD on Blue Gene/P on different sized partitions. A standard benchmark used in the MD community was used for the runs: $92,227$-atom ApoLipoprotein-A1 (ApoA1). All runs in this paper were done with the PME

No. of cores	512	1024	2048	4096	8192	16384
Topology Oblivious	13.93	7.96	5.40	5.31	-	-
TopoPlace Patches	13.85	7.87	4.57	3.07	2.33	1.74
TopoAware LDBs	13.57	7.79	4.47	2.88	2.03	1.25

Table 1: Performance of NAMD (ms/step) on IBM Blue Gene/P

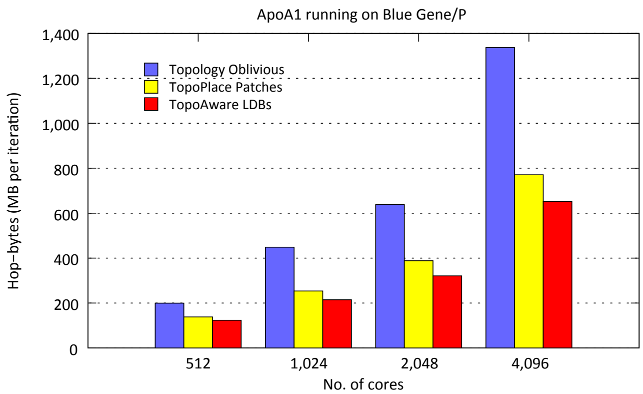

Figure 5: Hop-bytes for different schemes on IBM Blue Gene/P

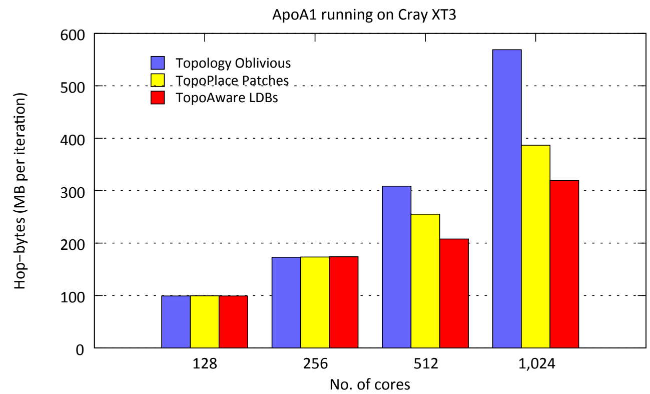

Figure 6: Hop-bytes for different schemes on Cray XT3

No. of cores	128	256	512	1024
Topology Oblivious	17.43	8.83	5.14	3.08
TopoPlace Patches	17.50	8.88	5.34	3.15
TopoAware LDBs	17.47	8.78	5.10	3.01

Table 2: Performance of NAMD (ms/step) on Cray XT3

No. of cores	512	1024	2048	4096
Topology Oblivious	4907	15241	22362	38421
TopoPlace Patches	4922	15100	22280	28981
TopoAware LDBs	4630	14092	20740	29572

Table 3: Reduction in total number of proxies on Blue Gene/P

computation turned off to isolate the load balancing issues of interest. As we would expect, hop-bytes consistently increase as we go from a smaller partition to a larger one. The three strategies compared are: topology oblivious mapping of patches and computes (Topology Oblivious), topology-aware static placement of patches (TopoPlace Patches) and topology-aware placement for both patches and load balancing for computes (TopoAware LDBs).

Topology aware schemes for the placement of patches and the load balancer help in reducing the hop-bytes for all processor counts. Also, the decrease in hop-bytes becomes more significant as we go to larger-sized partitions. This is due to the fact that the average distance traveled by each message increases as we increase the partition size in the case of default mapping, but it gets controlled when we do a topology-aware mapping. Since the actual performance of the load balancers depends on several metrics, so the question remains that does the reduction in hop-bytes lead to an actual improvement in performance. As it turns out, we also see a reduction in the number of proxies and in the *max-to-average* ratio for topology-aware load balancers, which is reflected in the overall performance of NAMD on Blue Gene/P (Table 1). The topology oblivious scheme stops scaling around $4,096$ cores and hence we did not obtain numbers for it beyond that. We see an improvement of 28% at $16,384$ cores with the use of topology-aware load balancers.

Similar tests were done on Cray XT3 to assess if a faster interconnect can hide all message latencies and make topology-mapping unnecessary. Figure 6 shows the hop-bytes for all messages per iteration when running NAMD on Cray XT3 on different sized partitions. We could only run on up to 1024 nodes (1 core per node) on XT3 and as a result we do not see a huge benefit on the lower processor counts. However, if we compare the 512 processor runs on XT3 with 2048 processor (512 node) runs on Blue Gene/P, we see a similar reduction in hop-bytes. It is also reflected in a slight improvement in performance at this point (Table 2).

Improvement in performance indicates that computational load is balanced. Reduction in hop-bytes indicates a reduction in the communication traffic on the network. A reduction in communication volume can be inferred from the number of proxies during a simulation. Table 3 presents the number of proxies being used in a particular run with different topology schemes. It is clear from the table that topology aware schemes reduce the total number of proxies also apart from reducing hop-bytes.

4.2 Improving Communication Load Prediction

As a part of the ongoing efforts to improve load balance in NAMD, it was noticed that NAMD uses a simplified model for communication as only total communication load per processor is recorded. This was leading to non-optimal load balancing results. To address this issue, we modified the

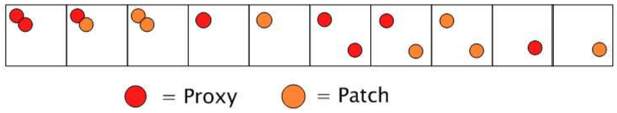

Figure 7: Adding SMP-awareness to the preference table for compute placement

load balancer to consider the addition/deletion of a proxy to/from a processor. Whenever we add a proxy to a processor, we increase the load of that processor by a small pre-determined value. When we remove a proxy from a processor because it is no longer needed, we decrease the processor load by similar amount. This optimization leads to performance improvements (up to 10%) when running at a small atoms-to-processor ratio on a large number of processors. The results in Table 1 are inclusive of the performance improvements from this technique.

5. FUTURE WORK

We present some research ideas which were out of the scope of this paper but will be used to effectively use the upcoming architectures and scale the code to a large number of processors.

5.1 SMP-aware Techniques

Most big cluster machines and supercomputers today consist of SMP nodes ranging from 4 to 16 cores per node. On such machines, communication can be further optimized by considering the fact that intra-node communication is faster than inter-node communication and that favoring intra-node communication reduces contention on the network. Load balancers in NAMD can utilize the fact that there are multiple cores per node.

Let us consider Blue Gene/P or XT4 and assume 4 cores per node for this analysis. However, the technique is general and applies to any number of cores on a node. We can make changes to the preference table while choosing a suitable processor, to give preference to processors which are on the same node at each step of the algorithm (I, II and III.) Thus, we can make two preference tables (Figure 7) to accommodate new scenarios. We first consider placing on the processor with a proxy or patch (first five cells in the table), then consider a processor whose neighbors on the same node contain a proxy or patch (next five cells) and then consider other external processors.

5.2 Hierarchical Load Balancers

The load balancers used in the production version of NAMD are centralized – this means that the instrumentation information is collected on one processor and the decisions are sent out from that processor. Although load balancing happens infrequently in a simulation run (once in every 5000 steps), it is more frequent in a benchmarking run and can take a large time on a very large number of processors.

To effectively scale NAMD to petascale machines such as Blue Waters, we need to modify the centralized scheme and use hierarchical ones [18]. In hierarchical load balancing, we divide all the processors into groups. Every group has a master rank which communicates with masters of other groups and takes the load balancing decisions for its own group. The load balancing strategy used within a group and across groups can be different and hence, these load balancers are also referred to as hybrid load balancers.

6. CONCLUSION

This paper discussed load balancing algorithms deployed in the highly scalable MD application, NAMD. It demonstrated the impact of topology-aware mapping techniques on application performance on 3D torus/mesh architectures. We presented results on two machines, ANL's Blue Gene/P, which is a 3D torus and PSC's XT3, which is a 3D mesh. In both cases, we saw a reduction in hop-bytes, which is a measure of available link bandwidth used by the application, and an improvement in performance up to 10%. We expect larger performance improvements on machines with larger torus sizes and on machines with faster processors.

The scenario which the application presents points to a more general situation where every object multicasts to some target objects and every target of the multicast receives from multiple sources. Schemes similar to those used in this paper can be deployed in such a scenario to restrict communication to a smaller portion of the torus. The best place for a multicast target is within the brick formed by enclosing all multicast sources for this target. Any processor within this brick gives the minimum number of hops from the sources to the target.

The paper also presented different factors which affect the success of load balancers. Heuristic-based load balancing involves a complex interplay of such metrics and a deeper understanding of these issues will be part of future work. Latency tolerant applications, including NAMD, also mitigate the effect of communication contention, typically at some cost in overhead due to fine-graining. The trade-off between grainsize and topology-aware balancers also needs further analysis.

Acknowledgements

This work was supported in part by a DOE Grant B341494 funded by Center for Simulation of Advanced Rockets, DOE Grant W-7405-ENG-48 for load balancing and by a NIH Grant PHS 5 P41 RR05969-04 for Molecular Dynamics. We thank Brian W. Johanson and Chad Vizino from PSC for help with system reservations and runs on BigBen (under TeraGrid allocation grant MCA93S028 supported by NSF). We also used running time on the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357. We thank Peter L. Freddolino for providing the text and figure for Section 2.

7. REFERENCES

[1] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[2] Kevin J. Bowers, Edmond Chow, Huafeng Xu, Ron O. Dror, Michael P. Eastwood, Brent A. Gregersen, John L. Klepeis, Istvan Kolossvary, Mark A. Moraes, Federico D. Sacerdoti, John K. Salmon, Yibing Shan, and David E. Shaw. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference*

on Supercomputing, New York, NY, USA, 2006. ACM Press.

[3] Blake G. Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, T. J. Christopher Ward, Mark Giampapa, and Michael 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] S. J. Plimpton and B. A. Hendrickson. A new parallel method for molecular-dynamics simulation of macromolecular systems. *J Comp Chem*, 17:326–337, 1996.

[5] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant 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.

[6] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[7] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[8] L. V. Kalé, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 251–261, 1998.

[9] T. P. Straatsma and J. A. McCammon. Load balancing of molecular dynamics simulation with NWChem. *IBM Syst. J.*, 40(2):328–341, 2001.

[10] Shahid H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[11] P. Sadayappan and F. Ercal. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Trans. Computers*, 36(12):1408–1424, 1987.

[12] Francois Gygi, Erik W. Draeger, Martin Schulz, Bronis R. De Supinski, John A. Gunnels, Vernon Austel, James C. Sexton, Franz Franchetti, Stefan Kral, Christoph Ueberhuber, and Juergen Lorenz. Large-Scale Electronic Structure Calculations of High-Z Metals on the Blue Gene/L Platform. In *Proceedings of the International Conference in Supercomputing*. ACM Press, 2006.

[13] Hideaki Kikuchi, Bijaya B. Karki, and Subhash Saini. Topology-aware parallel molecular dynamics simulation algorithm. In *PDPTA*, pages 1083–1088, 2006.

[14] Abhinav Bhatelé and Laxmikant V. Kalé. Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, 18(4):549–566, 2008.

[15] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. 14:437–449, 2006.

[16] Abhinav Bhatele and Laxmikant V. Kale. An Evaluation of the Effect of Interconnect Topologies on Message Latencies in Large Supercomputers. In *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS '09)*, May 2009.

[17] Sameer Kumar, Chao Huang, Gengbin Zheng, Eric Bohm, Abhinav Bhatele, James C. Phillips, Hao Yu, and Laxmikant V. Kalé. Scalable Molecular Dynamics with NAMD on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):177–187, 2008.

[18] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.