Load Balancing in Parallel Molecular Dynamics*

L. V. Kalé, Milind Bhandarkar and Robert Brunner

Dept. of Computer Science, and
Theoretical Biophysics Group, Beckman Institute,
University of Illinois,
Urbana Illinois 61801
{kale,milind,brunner}@ks.uiuc.edu

Abstract. Implementing a parallel molecular dynamics as a parallel application presents some unique load balancing challenges. Non-uniform distribution of atoms in space, along with the need to avoid symmetric redundant computations, produces a highly irregular computational load. Scalability and efficiency considerations produce further irregularity. Also, as the simulation evolves, the movement of atoms causes changes in the load distributions. This paper describes the use of an object-based, measurement-based load balancing strategy for a parallel molecular dynamics application, and its impact on performance.

1 Introduction

Computational molecular dynamics is aimed at studying the properties of biomolecular systems, and their dynamic interactions. As human understanding of biomolecules progresses, such computational simulations become increasingly important. In addition to their use in understanding basic biological processes, such simulations are used in rational drug design. As researchers have begun studying larger molecular systems, consisting of tens of thousands of atoms, (in contrast to much smaller systems of hundreds to a few thousands of atoms a few years ago), the computational complexity of such simulations has dramatically increased.

Although typical simulations may run for weeks, they consist of a large number of relatively small-grained steps. Each simulations step typically simulates the behavior of the molecular system for a few femtoseconds, so millions of such steps are required to generate several nanoseconds of simulation data required for understanding the underlying phenomena. As the computation involved in each timestep is relatively small, effective parallelization is correspondingly more difficult.

Although the bonds between atoms, and the forces due to them, have the greatest influence on the evolving structure of the molecular system, these forces do not constitute the largest computational component. The non-bonded forces,

* This work was supported in part by National Institute of Health (NIH PHS 5 P41 RR05969-04 and NIH HL 16059) and National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827EQ).

the van der Waals and electrostatic (Coulomb) forces between charged atoms consume a significant fraction of the computation time. As the non-bonded forces decrease as the square of interaction distance, a common approach taken in biomolecular simulations is to restrict the calculation of electrostatic forces within a certain radius around each atom (*cutoff* radius). Cutoff simulation efficiency is important even when full-range electrostatic forces are used, since it is common to perform several integration steps using only cutoff forces between each full-range integration step. This paper therefore focuses on cutoff simulations performance.

In this paper, we describe the load balancing problems that arise in parallelization of such applications in context of a production quantity molecular dynamics program, NAMD 2 [5], which is being developed in a collaboratory research effort. The performance of the load balancing strategies employed by this program are evaluated for a real biomolecular system. The load balancing strategy employed is based on the use of migratable objects, which are supported in Charm++ [7], a C++ based parallel programming system. It relies on actual measurement of time spent by each object, instead of predictions of their computational load, to achieve an efficient load distribution.

2 NAMD: A Molecular Dynamics Program

NAMD [8] is the production-quality molecular dynamics program we are developing in the Theoretical Biophysics group at the University of Illinois. From inception, it has been designed to be a scalable parallel program. The latest version, NAMD 2 implements a new load balancing scheme to achieve our performance goals.

NAMD simulates the motions of large molecules by computing the forces acting on each atom of the molecule by other atoms, and integrating the equations of motions repeatedly over time. The forces acting on atoms include bond forces, which are spring-like forces approximating the chemical bonds between specific atoms, and non-bonded forces, simulating the electrostatic and van der Waals forces between all pairs of atoms. NAMD, like most other molecular dynamics program, uses a cutoff radius for computing non-bonded forces (although it also allows the user to perform full-range electrostatics simulations computation using DPMTA library[9]).

Two decomposition schemes are usually employed to parallelize molecular dynamics simulations: force decomposition and spatial decompositon. In the more commonly used scheme, force decomposition, a list of atom pairs is distributed evenly among the processors, and each processor computes forces for its assigned pairs. The advantage of this scheme is that perfect load balance is obtained trivially by giving each processor the same number of forces to evaluate. In practice, this method is not very scalable because of the large amount of communication that results from the distribution of atom-pairs without regard for locality. NAMD uses a spatial decomposition method. The simulation space is divided into cubical regions called *patches*. The dimensions of the patches are chosen to be slightly larger than the cutoff radius, so that each patch only needs the coordinates from the 26 neighboring patches to compute the non-bonded forces. The disadvantage of spatial decomposition is that the load represented by a patch varies considerably because of the variable density of atoms in space, so a smarter load balancing strategy is necessary.

The original version of NAMD balanced the load by distributing patches among processors and giving each patch responsibility for obtaining forces exerted upon its constituent atoms. (In practice, Newton's third law insures that the force exerted by atom i due to atom j is the same as that exerted on j by i. Such pair interactions are computed by one of the owning patches, and the resulting force sent to the other patch, halving the amount of computation at the expense of extra communication.) We discovered that this method did not scale efficiently to larger numbers of processors, since out of a few hundred patches, only a few dozen containing the densest part of a biomolecular system accounted for the majority of the computation. The latest version of NAMD adds another abstraction, *compute* objects (see figure 1). Each compute object is responsible for computing the forces between one pair of neighboring patches. These compute objects may be assigned to any processor, regardless of where the associated patches are assigned. Since there is a much larger number of compute objects than patches, the program achieves much finer control over load balancing than previously possible. In fact, we found that the program could reach a good load balance just by moving compute objects, so we also refer to them as *migratable* objects. Migratable objects represent work that is not bound to a specific processor.

Most communication in NAMD is handled by a third type of object, called *proxy patches*. A proxy patch is an object responsible for caching patch data on remote processors for compute objects. Whenever data from a patch X is needed on a remote node, a proxy PX for patch X is created on that node. All compute objects on that remote node then access the data of patch X through proxy patch PX, so that the patch data is sent to a particular processor only once per simulation step. The main implication of this for load balancing is that once a proxy patch is required on a particular processor, placing additional compute objects on that processor does not result in increased communication cost.

3 Static Load Balancing

NAMD uses a predictive computational load model for initial load balancing. The computational load has two components. One component, proportional to the number of atoms, accounts for communication costs, integration, and bond force computation. The second component resulting from the non-bonded force computation, is based on the number of atom pairs in neighboring patches. Profiling indicates that this component can consume as much as eighty percent of typical simulations, so good overall load balance depends on good distribution of this work.

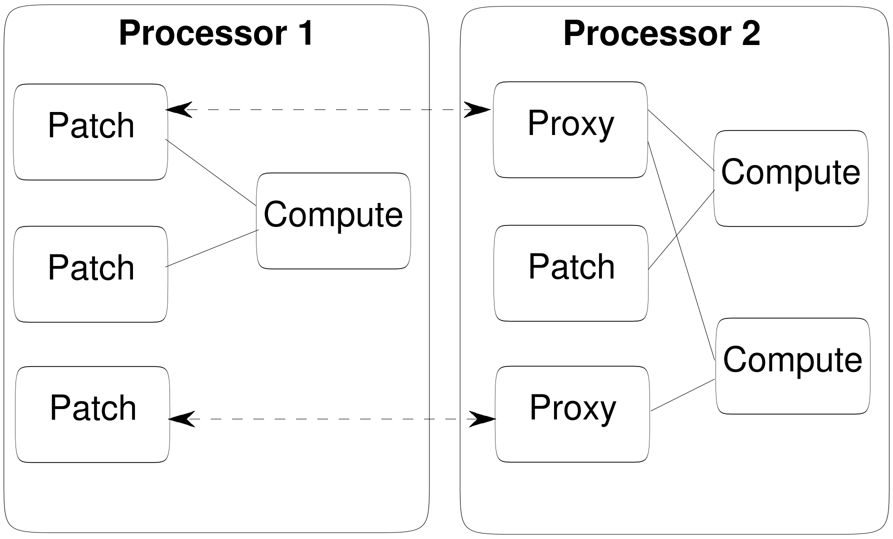

Fig. 1. This is a diagram showing object interactions in NAMD 2. Compute objects use data held by either patches or proxies. Each proxy receives and buffers position data from its owner patch on another processor, and returns forces calculated by compute objects to the owner patch. In order to move a Compute object to another processor, it is only necessary to create proxies on the new processor for all patches from which the compute receives position data.

The first stage of load balancing uses a recursive-bisection algorithm to distribute the patches among the processors so that the number of atoms on each processor is approximately equal. The recursive bisection algorithm ensures that adjacent patches are usually assigned to the same processor.

The second stage is the distribution of compute objects that carry out the non-bonded force computations. First, the compute objects responsible for self-interactions (interactions between a pair of atoms where both atoms are owned by the same patch) are assigned to the processor where the patch has been assigned, and the load for the processor incremented by the N^2_{atoms}. Then the compute objects for each pair of neighboring patches are considered. If the patches reside on the same processor, the compute object is assigned to that processor; otherwise the compute object is assigned to the least-loaded of the two processors. Then the load-balancer increments the load for that processor by $weight \times N_{atoms_1} \times N_{atoms_2}$. The weights take into account the geometric relationship between the two patches. There are three weights corresponding to whether the patches touch at a corner, edge or face, since patches which have an entire face in common contribute more pairs which actually must be computed than patches which share only a corner.

This method yields better load balance than earlier implementations, which only balanced the number of atoms. However, it does not take advantage of the freedom to place compute objects on processors not associated with either patch. It is possible to build a more sophisticated static load balancing algorithm that uses that degree of freedom, and even tries to account for some communication related processor-load. However, the geometric distributions of atoms within a patch impacts the load considerably, making it harder to predict by static methods. So, we decided to supplement this scheme with a dynamic load balancer.

4 Dynamic Load Balancing

Dynamic load balancing has been implemented in NAMD using a distributed object, the *load balance coordinator*. This object has one branch on each processor which is responsible for gathering load data and implementing the decisions of the load balancing strategy.

4.1 The Load Balancing Mechanism

NAMD was originally designed to allow periodic load rebalancing to account for imbalance as the atom distribution changes over the course of the simulation. We soon observed that users typically break multi-week simulations into a number of shorter runs, and that rebalancing at the start of each of these runs is sufficient to handle changes in atom distribution. However, difficulties with getting a good static load balance suggested that the periodic rebalancing could be used to implement a measurement-based load balancer.

If the initial patch assignment makes reasonable allowances for the load represented by each patch, compute object migration alone provides a good final load balance. During the simulation, each migratable object informs the load balance coordinator when it begins and ends execution, and the coordinator accumulates the total time consumed by each migratable object. Furthermore, the Converse runtime system [6] provides callbacks from its central message-driven scheduler, which allows the coordinator to compute idle time for each processor during the same period. All other computation, including the bond-force computations and integration, is considered background load. The time consumed by the background load is computed by subtracting the idle time and the migratable object times from the total time.

After simulating several time steps, the load balance coordinator takes this data and passes it to the selected load balancing strategy object (see section 4.2). The strategy object returns new processor assignments for each migratable object. The coordinator analyzes this list and determines where new proxy patches are required. The coordinator creates these new proxy patches, moves the selected migratable objects, and then resumes the simulation.

The first rebalancing results in many migratable object reassignments. The large number of reassignments usually results in changes to the background load, due to (difficult to model) changes in communication patterns. Therefore, after a few more steps of timing, a second load balancing step is performed, using an algorithm designed to minimize changes in assignments (and therefore in communication load). The second balancing pass produces a small number of additional changes, which do not change the background load significantly, but result in an improved final load distribution.

4.2 The Load Balancing Strategy

The load balancing strategy object receives the following pieces of information from the load balance coordinator (all times/loads are measured for several recent timesteps):

- The background (non-migratable) load on each processor.
- The idle time on each processor.
- The list of migratable objects on each processor, along with the computation load each contributes.
- For each migratable object, a list of patches it depends on.
- For each patch, its home processor, as well as the list of all processors on which a proxy must exist for non-migratable work.

Based on this information, the strategy must create a new mapping of migratable objects to processors, so as to minimize execution time while not increasing the communication overhead significantly. We implemented several load balancing strategies to experiment with this scenario. Two of the most successful ones are briefly described below.

It is worth noting that a simple greedy strategy is adequate for this problem if balancing computation were the sole criterion. In a standard greedy strategy, all the migratable objects are sorted in order of decreasing load. The processors are organized in a *heap* (i.e. a prioritized queue), so that the least loaded processor is at the top of the heap. Then, in each pass, the heaviest unassigned migratable object is assigned to the least loaded processor and the heap is reordered to account for the affected processor's load.

However, such a greedy strategy totally ignores communication costs. Each patch is involved in the electrostatic force computations with 26 neighboring patches in space. In the worst-case, the greedy strategy may end up requiring each patch to send 26 messages. In contrast, even static assignments, using reasonable heuristics, can lead to at most six or seven messages per patch. In general, since more than one patch resides on each processor, message-combining and multicast mechanisms can further reduce the number of messages per patch if locality is considered. Since the communication costs (including not just the cost of sending and receiving messages, but also the cost of managing various data structures related to proxies) constitute a significant fraction of the overall execution time in each timestep, it is essential that the load balancing algorithm also considers these costs.

One of the strategies we implemented modifies the greedy algorithm to take communication into account. Processors are still organized as a heap, and the migratable objects sorted in order of decreasing load. At each step, the "heaviest" unassigned migratable object is considered. The algorithm iterates through all the processors, and selects three candidate processors: (1) the least loaded processor where both patches (or proxies) the migratable object requires already exist (so no new communication is added), (2) the least loaded processor on which one of the two patches exists (so one additional proxy is incurred), and (3) the least loaded processor overall. The algorithm assigns the migratable object the best of these three candidates, considering both the increase in communication and the load imbalance. This step is biased somewhat towards minimizing communication, so the load balance obtained may not be best possible.

After all the migratable objects are tentatively assigned, the algorithm uses a refinement procedure to reduce the remaining load imbalance. During this step, all the overloaded processors (whose computed load exceeds the average by a certain amount) are arranged in a heap, as are all the under-loaded processors. The algorithm repeatedly picks a migratable object from the highest loaded processor, and assigns it to a suitable under-loaded processor.[2]

The result of implementing these new object assignments, in addition to anticipated load changes, creates new communication load shifts. Also, a part of the background work depends on the number (and size) of proxy patches on each processor, and increases or decrease as a result of the new assignment. As a result, the new load balance is not as good as the load balance strategy object expected. Rather than devising more complex heuristics to account for this load shift, we chose a simpler option. As described in section 4.1, a second load balancing pass is performed to correct for communication-induced load imbalance. This pass uses the refinement procedure only. Since the load is already fairly well balanced, only a few migratable objects are moved, improving load balance further without significant change in communication load. We find two passes sufficient to produce good load balance.

5 Performance

Performance measurements of various runs of NAMD were done using Projections, a performance tracing and visualization tool developed by our research group. Projections consists of an API for tracing system as well as user events and a graphical tool to present this information to the user in addition to modules to analyze the performance. To suit our needs for this particular task, Projections was extended with capabilities such as generating histograms, displaying user-defined markers, and tracing thread-events.

Our performance results are from an actual molecular system (ER-GRE, an estrogen receptor system) being studied by the Theoretical Biophysics Group. The system is composed of protein and DNA fragments in a sphere of water. Approximately 37,000 atoms are simulated using an 8.5 Å cutoff. The simulations are run on 16 processors of CRAY T3E.

[2] The techniques used to select the appropriate pair of candidates is somewhat involved and have been omitted.

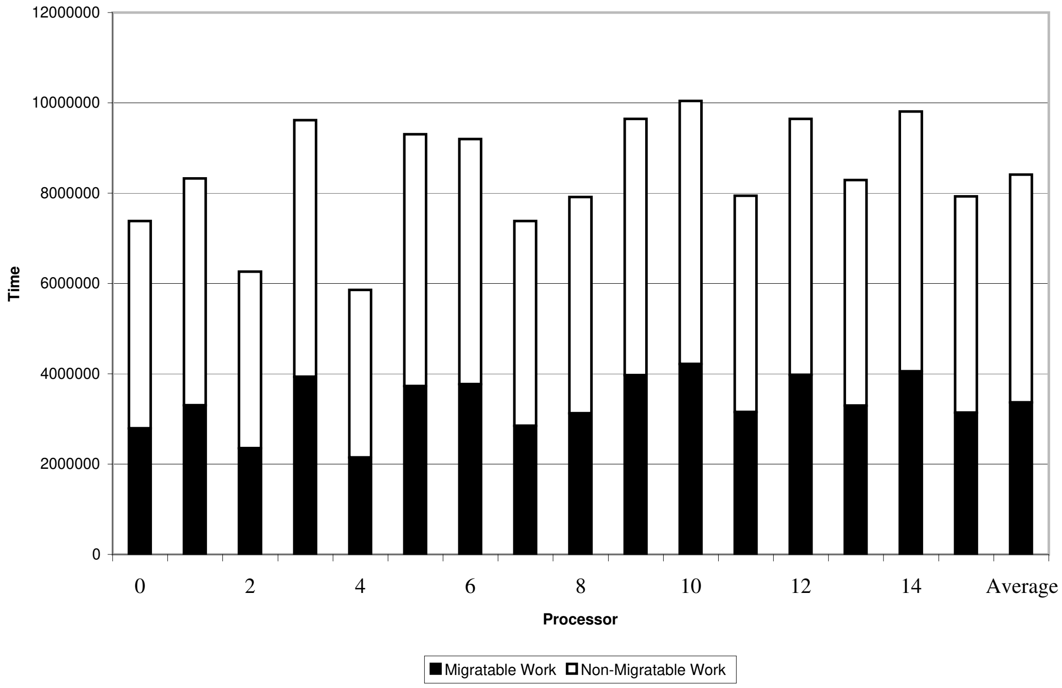

Fig. 2. Load with static load balancing

Figures 2 and 3 show a typical improvement obtained with our load balancer. Figure 2 shows measured execution times for the first eight steps of the simulation (where load balancing is based on a predictive static load balancing strategy described in section 3), and figure 3 shows times for the eight steps after load balancing. Since total execution time is determined by the time required by the slowest processor, reducing the maximum processor time by ten percent decreases execution time by the same amount. We have also observed that for larger numbers of processors, the static load balancer produces even more load variation. The measurement-based load balance does not produce such variation, producing greater performance improvement. Comparisons of the average times (the rightmost bar on each plot) shows that the new load distribution does not increase computational overhead to obtain better balance.

Figure 4 shows the speedup compared to one processor for the same simulation, using measurement-based load balancing. Since the number of computational objects is fixed, and total communication increases with increasing number of processors, load balancing grows more complex. Our load balancer exhibits good speedup even for larger numbers of processors.

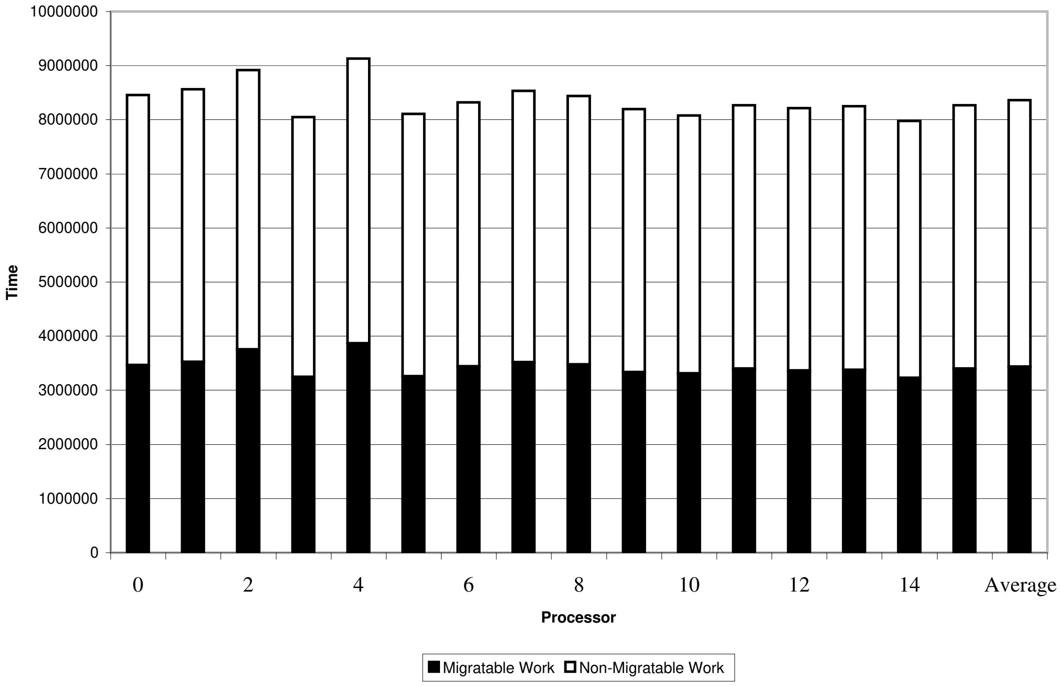

Fig. 3. Load after measurement-based load balancing

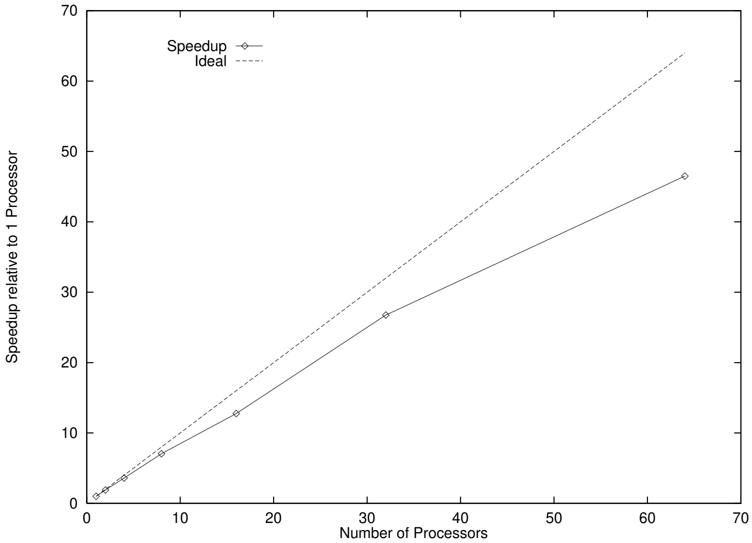

Fig. 4. Speedup for simulation steps after load balancing.

6 Conclusion

Molecular dynamics simulation presents complex load balancing challenges. The pieces of work which execute in parallel represent widely varied amounts of computation, and the algorithm exhibits an appreciable amount of communication. We found it difficult to devise good heuristics for a static load balancer, and therefore used measurement-based dynamic load balancing that was accurate and simpler to implement, especially when object migration is supported by the programming paradigm.

Future work will focus on improvements on the load balancing strategy. Smarter strategies may be able to reduce communication costs in addition to balancing the load, producing speed improvements beyond that obtained through equalizing load without great attention to communication. The current load balancer also does not address memory limitations; for large simulations it may be impossible to place objects on certain processors if memory on those processors is already consumed by simulation data. Although the measurement-based scheme is more capable of adjusting to changes in processor and communication speeds than other schemes, we will also study the algorithms behavior for other architectures, including workstation networks and shared memory machines. Also, the current implementation of load-balancing strategy is centralized on processor 0. We plan to provide a framework for parallel implementation for future load-balancing strategies.

References

1. Bernard R. Brooks, Robert E. Bruccoleri, Barry D. Olafson, David J. States, S. Swaminathan, and Martin Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.
2. Axel T. Brünger. *X-PLOR, A System for X-ray Crystallography and NMR.* Yale University Press, 1992.
3. Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgeway Scott. Parallelizing molecular dynamics using spatial decomposition. Technical report, Center for Research on Parallel Comutation, Rice University, P.O. Box 1892, Houston, TX 77251-1892, November 1993.
4. H. Heller, H. GrubMuller, and K. Schulten. Molecular dynamics simulation on a parallel computer. *Molecular Simulation*, 5, 1990.
5. L. V. Kalé, Milind Bhandarkar, Robert Brunner, Neal Krawetz, James Phillips, and Aritomo Shinozaki. A case study in multilingual parallel programming. In *10th International Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, Minnesota, June 1997.
6. L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.
7. L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

8. Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. NAMD— A parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.
9. W. Rankin and J. Board. A portable distributed implementation of the parallel multipole tree algorithm. *IEEE Symposium on High Performance Distributed Computing*, 1995. [Duke University Technical Report 95-002].
10. W. F. van Gunsteren and H. J. C. Berendsen. *GROMOS Manual*. BIOMOS b. v., Lab. of Phys. Chem., Univ. of Groningen, 1987.
11. P. K. Weiner and P. A. Kollman. AMBER: Assisted model building with energy refinement. a general program for modeling molecules and their interactions. *Journal of Computational Chemistry*, 2:287, 1981.