Scalable Molecular Dynamics for Large Biomolecular Systems

Robert K. Brunner
Theoretical Biophysics Group
University of Illinois at Urbana-Champaign
rbrunner@uiuc.edu

James C. Phillips
Theoretical Biophysics Group
University of Illinois at Urbana-Champaign
jcphill@uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu

Abstract

We present an optimized parallelization scheme for molecular dynamics simulations of large biomolecular systems, implemented in the production-quality molecular dynamics program NAMD. With an object-based hybrid force and spatial decomposition scheme, and an aggressive measurement-based predictive load balancing framework, we have attained speeds and speedups that are much higher than any reported in literature so far.

The paper first summarizes the broad methodology we are pursuing, and the basic parallelization scheme we used. It then describes the optimizations that were instrumental in increasing performance, and presents performance results on benchmark simulations.

1 Introduction

Understanding the structure and function of biomolecules such as proteins and DNA is crucial to our ability to understand the mechanisms of diseases, drugs, and normal life processes. With the experimental determination of structures for an increasing set of proteins it has become possible to employ molecular dynamics (MD) simulations in such studies. Recent success in sequencing the entire human genome has given a significant incentive to increased use of MD, and the announcement by IBM of a project aimed at creating a million processor computer for MD simulations underscores the importance of this area. With these advances, significant breakthroughs can be expected during the next decade, leading to better treatment of diseases, for example. One necessary factor for this to occur is the ability to carry out MD simulations in a scalable fashion on a large number of processors.

In classical molecular dynamics, a biomolecular system (proteins, lipids, and/or nucleic acids, often surrounded by water) is modeled as a collection of atoms connected by bonds, and interacting via electrostatic and van der Waal's forces. Due to high frequency bond vibrations, the Newtonian equations of motion must be integrated in time-steps of (typically) one femtosecond while many phenomena of interest occur on time scales of nanoseconds or longer.

Although the total amount of computation required is large, it is divided into a very large number of time steps. The size of the molecular system to be modeled is typically only a few tens of thousands of atoms. Thus, each time step, which constitutes the basic unit available for parallelization, is relatively small: a few seconds to tens of seconds on sequential computers. These factors couple with the irregular and dynamic nature of the computation to generate one of the most difficult problems to parallelize in a scalable manner.

To be sure, some popular mainstream MD programs have already been parallelized, with good speedups over tens of processors, and useful, but lower speedups for up to 256 processors. Scaling them beyond this number has proved to be difficult. We describe a theoretically and practically scalable parallelization scheme for cutoff-based MD, and an aggressive parallel implementation based on data driven objects, automatic load balancing via object re-mapping, and specific performance optimizations. These have been embodied in NAMD,[1] a production-quality MD program used in a variety of biomolecular simulations published in scientific journals.

NAMD has attained, on a benchmark 206,617 atom protein-lipid-water assembly, a speedup of 1252 on 2048 processors, in comparison with speedups of 140 and 170 reported for other programs recently [5], and our own previous result of 180 on 256 processors. The actual speed of the program, which is written in C++, is comparable or better than other production-quality programs written in Fortran, on one processor.[2] We have identified a few more aggressive strategies that will allow us to increase the speedup even beyond the current high, as well as demonstrate speedups on larger (and faster) machine configurations.

Occasionally, computer science literature treats the MD problem as an N-body problem. Several excellent algorithms, and parallel implementations, are available for the N-body problem. However, the full MD problem involves forces due to bonds, which complicate the parallelization significantly. Further, use of a cut-off radius and Newton's third law are essential for efficient computation, which further complicate efficient parallelization. Lastly, the relatively small number of atoms involved makes it a harder problem to parallelize compared with multi-million particle N-body simulations used in astrophysics codes.

Note that even when full, long-range electrostatic interactions are included in a simulation, these forces may be calculated via an efficient combination of global grid-based and cutoff atom-based components. The results in this paper are directly applicable to the atom-based components of such methods. The remaining grid-based calculations consume a small fraction of the total computation time, particularly when combined with multiple timestepping methods, but their contribution to scalability must still be addressed. The parallelization of these methods is the subject of ongoing research by ourselves and others [14, 15, 16].

The next section describes a general methodology that we are developing for effective parallelization of dynamic and irregular computations, and how it is supported by the Charm++ parallel programming system. Section 3 briefly explains the basic parallel structure and algorithm used in NAMD, along with the load balancing strategy used by NAMD. Section 4 then describes several specific performance optimizations, and shows the performance attained on parallel machines. Some of the lessons learned are discussed in the concluding section.

[1] NAMD is available at `http://www.ks.uiuc.edu/Research/namd/`.

[2] NAMD was found to be 25% faster than CHARMM (a popular modeling package written in FORTRAN) on a standard CHARMM benchmark (carboxy myoglobin in water) run independently by Surjit B. Dixit and Christophe Chipot, University Henry Poincare-Nancy I, France, on 1 CPU of the SGI Origin 2000 at the Center Charles Hermite, INRIA-Lorraine, Villers les Nancy, France.

2 Parallelizing dynamic computations

An increasing number of parallel applications are irregular and dynamic in their structure. "Irregular" structure implies that different components of the application are not uniform: a simple iteration over an array is an example of regular structure, whereas a computation distributed over an oct-tree, or a finite element unstructured grid are relatively irregular. In physical simulations, complex geometries and/or uneven distribution of physical entities causes such irregularities. "Dynamic" problem structure results when the components of a program change their behavior over time. Such changes tend to affect performance negatively. Especially on a large number of processors, these two factors combine to pose a substantial hurdle to effective parallelization. Not only does the performance of such applications tend to be poor, the amount of effort required in developing them is also inordinate. The broad approach we have been pursuing for effective parallelization of such computations is based on the principle of persistence, described below.

2.1 Multi-partition decomposition and the principle of persistence

A broad approach for effective parallelization of such applications that we are developing can be summarized as follows: the application programmer decomposes the problem into a large number of "chunks". The number of chunks is chosen independently of the number of processors, and is typically much larger than the number of processors. (We call this "multi-partition decomposition" for brevity). From the programmer's point of view, all the communication is between the chunks, and not the processors. The runtime system is then free to map and re-map these chunks, implemented as data-driven objects, across processors. The system may do so in response to internal or external [3] load imbalances, or due to requirements of a time shared parallel system. This approach tries to automate what "runtime systems" can do best (e.g. work scheduling and load balancing), while leaving those tasks best done by humans (deciding what to do in parallel), to the programmer.

In scientific and engineering computations, a runtime system has one more factor that helps its task: such computations tend to be iterative in nature, and so the computation times and the communication patterns exhibited by its objects (chunks) tend to persist over time. We call this "the principle of persistence", on par with "the principle of locality" exhibited by sequential programs. The apparent contradiction of this principle with our objective of aiming at dynamic applications is resolved when we note that dynamic CSE applications tend to either change their pattern abruptly but infrequently (as in adaptive refinement) or continuously but slowly. The relatively rare case of continuous and large changes can still be handled by our paradigm, using more dynamic and localized remapping strategies. However, for the common case, a runtime system can employ a measurement-based approach: it can measure the object computation and communication patterns over a period of time, and base its object remapping decisions on these measurements. We have shown that such measurement-based load balancing leads to accurate load predictions, and coupled with good object remapping strategies, to high-performance for such applications [2, 4].

This approach shifts a substantial burden of parallel programming to the runtime system. The Charm++ and Converse parallel programming system provides the components that support this approach, and are described next.

2.2 Runtime support for dynamic computation

NAMD is implemented using the Converse [8] runtime system, and the major components of NAMD are written in Charm++ [10], a parallel version of C++. Converse supplies several important capabilities for parallel applications. It provides machine-independent interfaces to all popular parallel computers as well as workstation clusters. Converse is an interoperable runtime system, allowing programs to be composed of modules written using a variety of different parallel languages and libraries. NAMD takes advantage of this by including components written in Charm++, MPI, PVM, and a low-overhead send-receive message library. Finally, Converse implements a data-driven execution model, allowing parallel languages such as Charm++ to support the dynamic behavior of our chunk-based applications.

The dynamic components of NAMD are implemented in the Charm++ parallel language. Charm++ applications are composed of collections of C++ objects, which communicate by remotely invoking methods on other objects. When multiple chunks are mapped to each processor, some form of local scheduling is necessary. On each processor, there is a collection of objects waiting for data. Method invocations (messages) are sent from object to object. All method invocations for objects on a processor are maintained in a prioritized scheduler queue. The scheduler repeatedly picks the next available message, and invokes the indicated method on the indicated object with the message parameters. In addition to supporting multi-partition decomposition, such data-driven execution also adaptively overlaps communication and computation.

Applications based on a multi-partition decomposition also call for a sophisticated load-balancing system. We have developed such a load balancing framework for Charm++ programs. The framework automatically instruments all Charm++ objects, collects their timing and communication data at runtime (in a "database"), and provides a standard interface to different load balancing *strategies* (the job of a strategy is to decide on a new mapping of objects to processors). The framework is general enough to apply beyond the Charm++ context, and it has been implemented in the Converse runtime layer, requiring only a small amount of language-specific code to support additional parallel languages. As a result, several other languages on top of Converse (including threaded MPI) can also use the load balancing functionality. The strategies themselves are independent of the framework and can be plugged in and out easily. Also, it is possible to write strategies that are somewhat specialized to individual application domains, when necessary.

Some of the strategies supported are centralized whereas others are distributed. All strategies interface to the database built by the framework on each processor separately. A centralized strategy may use a system-provided library to gather the object communication graph on one processor. The strategy processes this graph to make decisions about which processor each object should be mapped to subsequently. The framework then communicates this map to individual processors, where object managers belonging to language runtime systems (such as Charm++) migrate their objects as indicated by the map. A distributed strategy does not collect all information in one place; instead it may choose to communicate with neighboring processors, to exchange information and then to exchange objects. There is clearly a higher overhead for centralized strategies. However, in many applications, including molecular dynamics, the load balance does not change significantly for a long period of time. In such applications, it is possible to spend a considerable amount of time in a centralized strategy to come up with a good new mapping.

3 A scalable parallel algorithm for molecular dynamics

Many existing implementations of parallel molecular dynamics use atom replication or atom decomposition techniques [1, 17]. Although these techniques allow relatively easy porting of existing sequential codes, they can be shown to be theoretically non-scalable: as the number of processors increases, the communication to computation ratio also increases, even if the problem size is arbitrarily increased. More sophisticated strategies, which are variants of force decomposition ([6, 12])

are also non-scalable in this sense, although in practice they may lead to reasonable speedups on medium-size computers (up to 128 processors). Spatial decomposition schemes (used in [11, 13]), where atoms are distributed into cubes depending on the spatial locations, and migrated among the cubes as needed, are shown to be theoretically scalable. For a more detailed analysis of scalability, see [9].

The variant of spatial decomposition we propose uses cubes whose dimensions are slightly larger than the cutoff radius. Thus, atoms in one cube need to interact only with their neighboring cubes; there are 26 such neighboring cubes. However, a 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 ApoA-I system used as a benchmark in this paper (with 92,442 atoms), we only have 245 ($7 \times 7 \times 5$) cubes. Further, as density of the system varies across space, one may encounter strong load imbalances.

NAMD uses a novel combination of force 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.

Forces due to covalent bonds within biomolecules are represented via a sum of 2-body (bond), 3-body (angle), and 4-body (dihedral and improper) terms which follow the topology of the molecule. In order to minimize communication while avoiding redundant calculations, a force computation object is created for each cube and its *upstream* neighbors, these being the (at most) 7 neighboring cubes at equal or greater coordinates along all three axes. Bonded forces among sets of (2, 3, or 4) atoms are calculated by this object if and only if the base cube coordinates are equal to the minimum of the cube coordinates for all constituent atoms along each axis.

Non-bonded interactions are excluded or modified between atoms connected by one, two, or three bonds. These pairs must be detected as a part of the normal pairwise force computation because the excluded forces would be many orders of magnitude larger than remaining forces. An earlier strategy of only checking for excluded pairs within a small radius was rendered obsolete by an efficient method of conducting such checks.

3.1 NAMD design details

In the following section, we will summarize the parallel decomposition used in NAMD. A more detailed description can be found in [9].

The cubes described above are represented in NAMD by objects 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*. 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 compute 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 integrating.

Some compute objects are placed on particular processors at the start of the simulation, but others may move during load balancing. 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 convenient to assume that some compute objects will not move at all during the course of the simulation. In general, the bulk of the computational load is represented by the non-bonded (electrostatic and van der Waal's) interactions, and certain types of bonds. These objects are designed to be able to migrate during the simulation to optimize parallel efficiency. 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.

3.2 Load balancing in NAMD

NAMD uses a measurement-based load balancer, employing the Charm++ load balancing framework, to achieve unsurpassed parallel performance for molecular dynamics. Load balancing occurs in three stages. When a simulation begins, patches are distributed according to a recursive coordinate bisection scheme, so that each processor receives a number of neighboring patches. When there are more processors than patches, this method reduces to a simple round-robin distribution, so that some processors have one patch, and the rest have none. All compute objects are then distributed to a processor owning at least one home patch, taking advantage of the upstream patch distribution to insure that each patch has at most seven proxies. Clearly this is not an efficient load distribution for large machines, since processors with no patches will also receive no compute objects, leaving many processors with no work at all. However, this procedure does achieve the goals of distributing the integration work reasonably well, and distributing the compute objects in such a way to avoid excessive communication. Thus this static load balancing step prepares the computation for a subsequent measurement based remapping.

The dynamic load balancer uses the load measurement capabilities of Converse to refine the initial distribution. 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 timesteps (typically several seconds to several minutes), the program suspends the simulation to trigger the initial load balancing. NAMD retrieves the object times and background load from the framework, collects the load information on one processor, and redistributes the migratable compute objects. The new object distribution is determined using the following algorithm.

- Select the biggest (longest-executing) compute object.
- Select a destination processor for the compute object such that:
  - Adding this compute object will not overload the processor much (an overload threshold permits some overload).
  - The compute object will utilize as many home patches as possible.
  - The assignment will create as few new proxy patches as possible.
  - Among multiple processors selected by the above criteria, select the least loaded processor as the destination processor.
- Assign the compute object to the selected processor
  - Add the compute object load to the processor's total load

  - Record the creation of new proxies, so that future compute objects may also use the proxy.

- Repeat until all compute objects are assigned.

Immediately after assigning the compute objects with this strategy, a refinement algorithm further reduces the load imbalance, by tolerating the creation of additional proxy patches. The refinement algorithm is almost identical to the initial procedure, except that the overload threshold is smaller, only compute objects from overloaded processors are considered for migration, and only underloaded processors are considered as destinations for migrating computes.

After determining which compute objects will migrate, the load balancer moves the objects, constructs new proxies as necessary, and resumes the simulation. However, the new communication patterns resulting from relocation of so many objects produce actual object times which differ from the times used in the initial load balancing. Therefore, after measuring a few more simulation steps, another load balancing cycle begins. This time, only the refinement procedure is used, resulting in only a few additional object migrations. After this second load balancing, the processor loads remain balanced, except for the slow large-scale movements of atoms in the simulation. Periodically thereafter, the refinement procedure is repeated to account for the slow changes of the simulation.

4 Optimizing performance

In the six years we have been developing NAMD, we have been optimizing its parallel performance for the types of machines that our users had access to at that particular time. In 1994, users were most interested in 8 to 16 processor workstation clusters. By 1998, many of our users had easy access to several dozen processors on machines such as the T3E and Origin 2000. We discovered that additional optimizations were necessary to run well on 64 processors. Now, users are gaining access to machines with hundreds and thousands of processors, and to use them efficiently, another round of optimization has proved necessary. A side benefit of each round of optimization is that it improves the performance on smaller machines as well.

4.1 Program instrumentation

Three levels of instrumentation were used to diagnose and optimize the performance of NAMD. The simplest information to collect is the time per iteration for various size parallel machines. This information is useful for detecting that a parallel performance problem exists, and for quantifying the benefits of particular program modifications. However, raw timestep times provide little information to assist in locating the causes of poor performance.

The second level of instrumentation uses the *summary profile* information produced by the Charm++ run-time system. Two types of trace information are stored in the summary profile. The first is the processor utilization for every processor throughout the program run. The second is the execution time consumed by every *entry method* in the parallel program. Since Charm++ is a message-driven system, execution of a particular operation (entry method) is triggered by the receipt of a particular message. The runtime generates summary profiles by accumulating the total time consumed by that method (including any functions or methods it calls, but not methods it invokes with messages) across invocations. This is similar to traditional profiling, but with two main advantages. Traditional profiling generally times execution at the function-call level, producing large amounts of profiling data associated with thousands of different functions ("thousands" may seem unrealistically large, but a modern object-oriented program indeed has such a large number of functions. NAMD, in particular, consists of several thousand functions). Summary profiles are smaller, since there are typically only dozens to a few hundred entry methods to keep track of. Also, in traditional profiling, since times are recorded for each function call, timing itself may produce noticeable perturbations in program execution.[3] Since summary times are associated with messages, timing overhead is not significant compared to message overhead.

Parallel execution traces, generated by the *Projections* component of Charm++, provide the most complete performance information. Projections data provides the most comprehensive information about program performance: for example, the trace information includes such data as: method A on processor 5 sent a message to method B on processor 4 at time 5.441 seconds, which started execution at 5.451 seconds, and finished at 5.478 seconds. This kind of information is invaluable for diagnosing some varieties of parallel performance problems such as detecting ways to shorten long critical paths. However, traces of this nature can be large, so often full-size runs can not be instrumented in this way. Shorter runs with tens of timesteps are used in when full traces are desired. The interference from instrumentation is not much higher with full traces, though, because the trace data is stored in memory buffers till the end of the program, and output only at the end. Thus, the file I/O time is not incurred during the critical timesteps being instrumented.

4.2 Performance enhancements

In this section, we present several examples of optimizations carried out to improve the performance to the level achieved. As a a significant portion of the technical effort represented by this paper was spent on such optimizations, it is useful to discuss a representative sample of such optimizations.

The molecule we have primarily used for testing is the ApoA-I system. This full model of a high density lipoprotein particle is composed of 92,224 atoms, and the simulation used a 12 Å electrostatic cutoff radius. The simulation parameters for this simulation result in 245 cubes (a $7 \times 7 \times 5$ array). The 12 Å cutoff was selected as being typical of the cutoffs used in actual simulations when full-range electrostatics are not used. Results for two other molecules are also presented, demonstrating that the optimizations are not specialized to a particular benchmark.

4.2.1 Grainsize control

The non-bonded computation can make up eighty percent or more of the total computation of a simulation. The initial implementation of NAMD called for 14 non-bonded compute objects per cube (13 with neighbors, and one to calculate the interactions within the cube). For ApoA-I, this resulted in 3430 non-bonded compute objects. Also, the computational load represented by these compute objects would vary widely, since a compute calculating the forces between two cubes meeting at a corner would have fewer pairs of atoms that actually interact than a compute calculating the interactions within a single cube. For efficient load balancing, we need many objects per processor, but on 1024 processors, this partitioning results in fewer than four non-bonded compute objects. Therefore, we modified the generation of compute objects to potentially create several compute objects to calculate the within-cube non-bonded atom pairs. The number of compute objects created is determined by the number of atoms initially assigned to the cube, so that the load balancer has more objects to distribute, and so that those objects have more uniform computational loads.

Each timestep of the ApoA-I benchmark took about 57 seconds on one processor of the ASCI-Red. To achieve perfect speedup on, say, 2000 processors (which is impossible due to inherent

[3]Profiling using statistical sampling is less likely to produce performance degradation, but may result in inaccuracy, and does nothing to reduce the size of the trace data

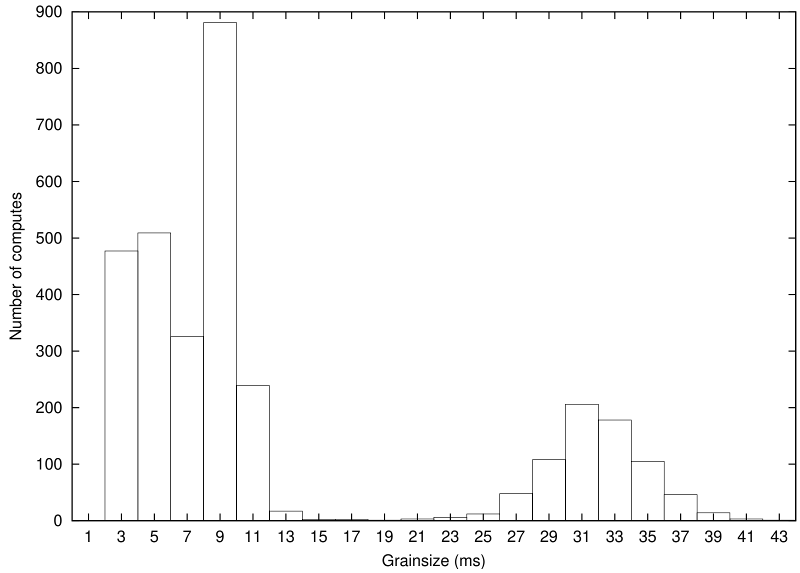

Figure 1: The grainsize of compute objects before splitting

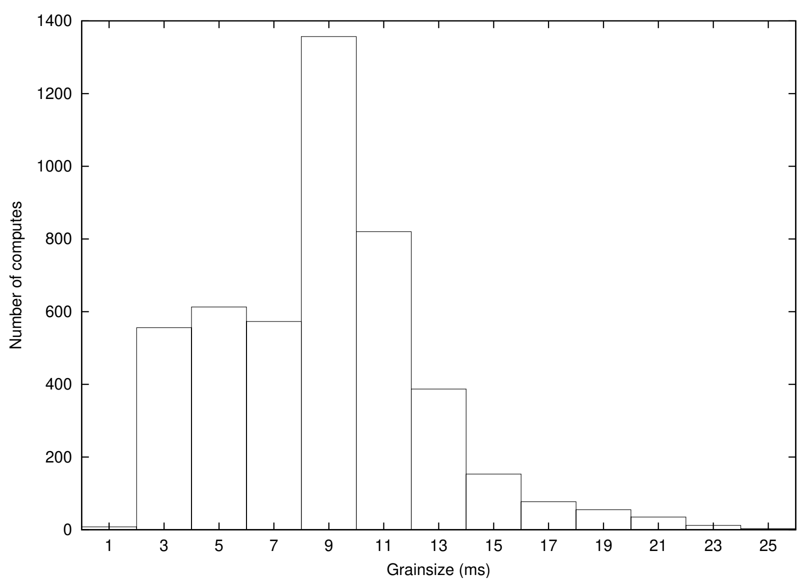

Figure 2: The grainsize of compute objects after splitting

communication and parallelization costs), we would need each processor to finish its part in 28 ms. A prerequisite, therefore, is that no single task (i.e. object method invocation) should run longer than 28 ms. To check this, we used the trace information generated by the Projections performance visualization tool. The critical method (task) is the one that computes non-bonded forces between neighboring patches. The data obtained using Projections on the spread of granularity of this method is shown in Figure 1. Each bar represents the number of instances of tasks with the grainsize indicated by its x-coordinate. (Thus there were about 880 tasks of grainsize 9 ms, or more precisely, of grainsize between 8 and 10 ms, during an average timestep.) The largest grainsize was around 42 ms, which is clearly undesirable. (This can be thought of as a corollary of Amdahl's law: In an object-based program, the best speedup is limited to $\frac{T_{sequential}}{T_{largest\text{-}object}}$.) A bimodal distribution of grainsizes is clearly visible in the plot. This suggested a particular type of computational objects were culprits. The objects that calculate interactions between two cubes that interface at a face (as opposed to a corner or edge) have many more interactions to compute, because more atoms from one cube are included within the cutoff radius of an atom of the other cube. We therefore implemented a strategy that splits such objects into multiple components (the number of pieces depends on the number of atoms in each interacting cube). This led to the distribution of grainsizes shown in Figure 2. This allowed the program to scale to a larger number of processors, and made the task of load balancer easier by providing smaller pieces of work.

4.2.2 Increased parallelism

Since the non-bonded work was now being parallelized effectively, other parts of the computation started to become problems. The next part of the computation requiring improvement was the parallelization of the bond computations. Although there are three types of bonds (2, 3, and 4-atom bonds) in NAMD, they have considerable similarity in the computations performed. Also, since they were a small part of the total computation, we simplified their implementation by making them non-migratable work, and by placing them so as to minimize communication. However, after distributing the non-bonded work across 1024 processors, the bond computation could no longer be ignored. Our solution for this problem rested on two facts. First, we considered the arrangements of bond with respect to the cubes. Although some bonds cross the boundaries between cubes, most are contained completely within a single cube. Therefore, we created two bond objects for each bond type associated with a cube. One computed the bond forces for atoms entirely within the cube; the other computed inter-cube bonds. Since this object still requires data from multiple cubes, its execution is delayed, but now it has much less work to do, so it shortens the computation on the critical path. The second change was to recognize that the intra-cube bond objects now communicate in exactly the same way as non-bonded self compute objects. Therefore, they could be made migratable, using the mechanisms already in place for non-bonded compute objects, and load balanced more efficiently.

4.2.3 Optimized multicast

To understand how the performance could be further improved, during the tuning stage, we kept a performance audit using summary data provided by Charm++ tracing routines. Table 1 shows a snapshot of the audit at an intermediate stage, when the time per step for a 1024 processor run of ApoA-I was around 86 ms. The audit compares ideal and actual 1024 processor data, where the ideal performance is computed by assuming that the single processor performance could scale perfectly. The 30 ms difference in total time was distributed across several columns: clearly load

	Time (milliseconds)							
	Total	Non-bonded	Bonds	Integration	Overhead	Imbalance	Idle	Receives
Ideal	57.04	52.44	3.16	1.44	0	0	0	0
Actual	86	49.77	3.9	3.05	7.97	10.45	9.25	1.61

Table 1: Performance audit for the ApoA-I simulation on 1024 processors of ASCI-Red. The Ideal times are calculated from the single processor times, assuming perfect scaling.

imbalance was a major factor (measured as the difference between maximum and average loads on processors). The communication overhead was significant, but relatively small. The extra work one had to do only in a parallel setting accounted for about 7.97 ms of the difference. In addition, the work done in the bonded force calculation and the integration methods increased somewhat (0.5 ms and 1.5 ms respectively) in the 1024 processor run. This led us to further improve the components that could be improved. For example, we noted that the integration component doubled. This observation lead to the following optimization.

The normal timestep of molecular dynamics algorithm alternates between force computation and integration. Integration is carried out only by the patches. As patch-size is decided by the cutoff radius, there are a fixed number of them available. For example, in the case of ApoA-I, there were 245 patches. So, on machines with more than 245 processors, the remaining processors tend to be idle during integration. Figure 3, obtained via Projections, shows this effect clearly, with time-lines for a few processors, in an "Upshot"-style diagram. Each rectangle on a processor's line represent an asynchronous method execution (or task). We noted that if the red (darkest) sections (representing integration) can be shortened, the gaps representing idle time on other processors (beyond processor 244) can be shortened. Further analysis via Projections showed that more than half of the time in this method was spent in sending 20-30 identical messages. The allocation and packing of messages was consuming most of the time. A simple utility was then added to the Charm++ runtime (as it is useful for other programs as well) that carries out the multicast by using only one user level packing and allocation. This shortened the duration of this critical entry method by half, as shown in Figure 4. The reduced gaps on processors that do not carry out integration are also clearly seen.[4]

Referring back to Table 1 it is clear that further improvement can be attained using better load balancer strategies and by reducing the overhead of processing coordinate and force messages (which is most of the 7.97 second component).

4.3 Results

The following tables present NAMD performance results for three different simulations, and three different parallel platforms. Most of our performance studies were conducted on the Sandia National Laboratories ASCI-Red computer, containing 9,536 333 MHz Pentium II Xeon processors, using the `-proc 1` coprocessor mode. Additional performance results are also presented for the Pittsburgh Supercomputer Center Cray T3E-900 and the NCSA SGI Origin 2000, with 250 MHz processors. The GFLOPS rating for each run was determined by using the instruction counters of the Origin 2000 to determine the number of floating-point operations per simulation step for a single-processor run. That number was divided by the time per step for each parallel run to

[4]Note that the x-axis scale on the latter graph is shorter

Figure 3: A Projections timeline view of two timesteps before optimizing the multicast.

Figure 4: A Projections timeline view of two timesteps after optimizing the multicast.

Processors	Time (s/step)	Speedup	GFLOPS
1	57.1	1	0.0480
4	14.7	3.9	0.186
8	7.31	7.8	0.375
32	1.9	30.1	1.44
64	0.964	59.2	2.84
128	0.493	116	5.56
256	0.259	221	10.6
512	0.152	376	18.0
768	0.102	560	26.9
1024	0.0822	695	33.3
1536	0.0645	885	42.5
2048	0.0573	997	47.8

Table 2: Program performance for the ApoA-I simulation (92,224 atoms) on the ASCI-Red

calculate the GFLOPS for the run. This procedure provides a conservative rating, since it ignores extra floating-point instructions due to parallelization.

In addition to the ApoA-I simulation previously described, results for two other simulations are also presented. The BC1 simulation is a very large simulation, composed of 206,617 atoms in 378 patches. The bR simulation is a small simulation of 3,762 atoms in 36 patches. Both of these simulations also use a 12 Å cutoff. During the optimization process, most experiments were run on the Sandia ASCI-Red computer.

Table 2 shows the performance of the program on the Sandia ASCI Red computer. Early in our performance study on this machine, our step times on 1024 processors was 120 ms, for a speedup of 475. Our optimization efforts led to a per step time of about 82 ms, leading to a speedup of 695 for this benchmark on 1024 processors. Table 3 demonstrates the performance of NAMD on the larger BC1 simulation. As expected, the larger problem makes better use of large numbers of processors. This simulation achieves a speedup of 1252 on 2048 processors (58.4 GFLOPS). Table 4 demonstrates the performance of NAMD on the much smaller bR simulation. Even on a system this small, NAMD is able to use up to 64 processors efficiently.

Table 5 shows the performance of NAMD on the Pittsburgh Supercomputer Center Cray T3E-900. Per-processor performance and scalability are both better than that achieved by the ASCI-Red, but the maximum program performance is limited by the number of available processors. Table 6 shows the performance of NAMD on the NCSA Origin 2000 (250 MHz processors).

We note again that the uni-processor speed of our program is on par with or better than other molecular dynamics programs. So, the impressive speedups were not attained by using a "bad sequential algorithm". Of course, the nature of MD computations precludes it from approaching the peak FLOP rating on a single processor as some other numerical kernels can. So, comparison with other efficient MD programs (often written in Fortran) appears to be a fair method. Even with that caveat, 110 MFLOPS on a single Origin 2000 processor (for example) can be claimed to be a good performance for a complete application.

Processors	Time (s/step)	Speedup	GFLOPS
2	74.2	2	0.0933
4	37.8	3.9	0.183
8	19.3	7.7	0.359
32	4.91	30.3	1.41
64	2.49	59.6	2.78
128	1.26	118	5.49
256	0.653	227	10.6
512	0.352	422	19.7
768	0.246	603	28.1
1024	0.192	773	36.1
1536	0.141	1052	49.1
2048	0.119	1252	58.4

Table 3: Program performance for the BC1 simulation (206,617 atoms) on the ASCI-Red. Performance is scaled relative to the speedup on two processors=2.0, since the simulation uses too much memory to run on 1 processor.

Processors	Time (s/step)	Speedup
1	1.47	1
2	0.759	1.94
4	0.384	3.83
8	0.196	7.50
32	0.071	20.7
64	0.0358	41.1
128	0.0299	49.2
256	0.0300	49.0

Table 4: Program performance for the bR simulation (3,762 atoms) on the ASCI-Red

Processors	Time (s/step)	Speedup	GFLOPS
4	10.7	4.0	0.256
8	5.28	8.1	0.519
16	2.64	16.2	1.04
32	1.35	31.7	2.03
64	0.688	62.2	3.98
128	0.356	120	7.69
256	0.185	231	14.8

Table 5: Program performance for the ApoA-I simulation on the PSC T3E-900. Scaling is relative to 4 processors= 4.0, since the problem is too large to run on fewer processors.

Processors	Time (s/step)	Speedup	GFLOPS
1	24.4	1	0.112
2	12.5	1.95	0.219
4	6.30	3.89	0.435
8	3.18	7.68	0.862
16	1.60	15.2	1.71
32	0.860	28.4	3.19
64	0.411	59.4	6.67
80	0.349	70.0	7.86

Table 6: Program performance for the ApoA-I simulation on the NCSA Origin 2000

5 Conclusion

We have demonstrated unprecedented performance and speedups on the molecular dynamics application. The lessons learned from this exercise can be summarized as follows:

1. Multi-partition decomposition, supported by data-driven execution and automatic measurement based load balancing is a promising methodology for dealing with complex parallel applications.
2. Within this approach, it is essential to follow the rule of dividing work into small pieces, as small as possible as long as they amortize the overhead. A few places where this rule was not followed turned out to be obstacles when attempting to scale beyond the number of processors used in earlier implementations. In general, with today's technology constants, it is desirable to aim at a grain-size (i.e. computation per message) of around 5 ms on the average. (10-50 times the message-passing overhead.)
3. Sophisticated load balancing strategies are necessary, that can attempt to increase load balance while keeping communication overhead low. Further progress on improving scalability will require strategies that consider the dependency chains, and load-balance within distinct phases of a single time step.
4. Low-overhead instrumentation, coupled with source language-level feedback and visualization can be used effectively to identify performance bottlenecks quickly, and to fix them.
5. A flexible and modular parallel program design allows for experimentation with alternate strategies.

The basic load balancing technology developed is being applied to other dynamic problems, including rocket simulation, crack-propagation, and simulation of physical processes. We expect the methodology to lead to effective parallelization of new classes of parallel applications in science and engineering.

6 Acknowledgments

NAMD was developed as a part of biophysics research at the Theoretical Biophysics Group (Beckman Institute, University of Illinois), which operates as an NIH Resource for Macromolecular Modeling and Bioinformatics. This resource is led by principle investigators Professors Klaus Schulten

(Director), Robert Skeel, Laxmikant Kalé and Todd Martinez. We are thankful for their support, encouragement, and cooperation. Prof. Skeel and coworkers have designed the specific numerical algorithms used in NAMD. We are grateful for the funding to the resource provided by the National Institutes of Health (under Grant NIH PHS 5 P41 RR05969-04).

NAMD itself is a collaborative effort. The early version of NAMD was implemented by a team including: Robert Brunner, Andrew Dalke, Attila Gursoy, Bill Humphrey and Mark Nelson. NAMD2, the current version, was implemented and is being enhanced by by a team consisting of: Milind Bhandarkar, Robert Brunner, Justin Gullingsrud, Attila Gursoy, David Hardy, Neal Krawetz, Jim Phillips, Ari Shinozaki, Krishnan Varadarajan, and Gengbin Zheng.

The authors of this paper were assisted in the parallelization, performance visualization, and efficiency improvement efforts described in this paper by: Milind Bhandarkar, Terry Wilmarth, Puneet Narula and Neelam Saboo. Milind Bhandarkar also implemented several parallel strategies, including the optimized multicast described in this paper, and provided Charm++ and Converse support.

The load balancing framework and strategies developed as a part of the Charm++ framework are also being used by the Center for Simulation of Advanced Rockets (CSAR or simply the Rocket Center) at the University of Illinois at Urbana-Champaign, funded by the Department of Energy. We are thankful to CSAR for allowing the use of the Turing cluster of Linux PCs, and the ASCI machines to test the load balancing strategies in the context of a challenging application.

The parallel runs were carried out at the Cray T3E at the Pittsburgh Supercomputing Center, the ASCI Red at Sandia National Laboratories, SGI Origin2000 at National Center for Supercomputing Applications (NCSA), and the Turing Cluster at CSAR at University of Illinois. We are thankful to these organizations and their staff. Although our runs were short, they needed to be run many times, and the staff was helpful in getting them to run (and keeping these machines operational!).

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] Robert Brunner. Versatile automatic load balancing with migratable objects. TR 00-01, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, January 2000.

[3] Robert K. Brunner and Laxmikant 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] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[5] Yong Duan, Lu Wang, and Peter A. Kollman. The early stage of folding of villin headpiece subdomain observed in a 200-nanosecond fully solvated molecular dynamics simulation. *Proceedings of the National Academy of Sciences, USA*, 95:9897–9902, August 1998.

[6] Y.-S. Hwang, R. Das, J. H. Saltz, Milan Hodošček, and Bernard R. Brooks. Parallelizing molecular dynamics programs for distributed-memory machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.

[7] 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.

[8] L. V. Kale, Milind Bhandarkar, Robert Brunner, and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[9] 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.

[10] 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.

[11] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kale, 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.

[12] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, March 1995.

[13] S. Plimpton and B. Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. *Journal of Computation Chemistry*, 17(3):326–337, February 1996.

[14] S. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics simulations. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, pages 8–21, 1997.

[15] 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].

[16] Abdulnour Toukmaji, Daniel Paul, and John A. Board Jr. Distributed particle-mesh Ewald: A parallel Ewald summation method. In *Proceedings of the Parallel and Distributed Processing Techniques and Applications Conference, Aug 9-11, 1996, Sunnyvale, CA*, 1996.

[17] 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.