NAMD2: Greater Scalability for Parallel Molecular Dynamics

Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner,
Attila Gursoy, Neal Krawetz, James Phillips,
Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten
Theoretical Biophysics Group,
Beckman Institute for Advanced Science and Technology,
University of Illinois at Urbana-Champaign,
Urbana, IL 61801.
{kale,skeel,milind,brunner,gursoy,
nealk,jim,ari,krishnan,kschulte}@ks.uiuc.edu

March 22, 1999

Abstract

Molecular dynamics programs simulate the behavior of biomolecular systems, leading to insights and understanding of their functions. However, the computational complexity of such simulations is enormous. Parallel machines provide the potential to meet this computational challenge. To harness this potential, it is necessary to develop a scalable program. It is also necessary that the program be easily modified by application-domain programmers.

The NAMD2 program presented in this paper seeks to provide these desirable features. It uses spatial decomposition combined with force decomposition to enhance scalability. It uses intelligent periodic load balancing, so as to maximally utilize the available compute power. It is modularly organized, and implemented using a parallel C++ dialect, so as to enhance its modifiability. It uses a combination of numerical techniques and algorithms to ensure that energy drifts are minimized, ensuring accuracy in long running calculations. NAMD2 uses a portable run-time framework that also supports interoperability among multiple parallel paradigms. As a result, different components of applications can be written in the most appropriate parallel paradigms. NAMD2 runs on most parallel machines including workstation clusters. This paper also describes the performance obtained on some benchmark applications.

1 Introduction and Motivation

Molecular dynamics (MD) serves a pivotal role in inferring functions of biomolecules from their structures. The number of structures obtained via x-ray crystallography and other means has increased tremendously in the past several years. Researchers believe that molecular dynamics simulations are critical in translating structure information into mechanisms underlying biomolecular functions. Successful simulations of biomolecular systems can lead to a deeper understanding of basic biological processes and provide new insight to aid rational drug design.

A major hurdle to successful deployment of molecular dynamics for these purposes is the sheer computational intensity of the required simulations. The biomolecular systems of interest today consist of ten thousand to one hundred thousand atoms. Most often, the dynamics of such systems must be studied over several nanoseconds to develop significant understanding of the phenomena being studied. At the same time, the time scale of atomic interactions requires that ome simulates their behavior in time steps as small as one femtosecond (10^{-15} seconds). In spite of the dramatically increased speeds of individual processors, the number of computational steps required to complete relevant simulations is prohibitive on any single-processor computer.

Parallel computing provides the potential for making the required large-scale simulations feasible. Parallel machines with tens, hundreds, and even thousands of processors are commercially available now. Such machines provide the potential to reduce the simulation time from years to mere days, thus making it possible to investigate functional properties of structures identified by contemporary scientific studies in a timely fashion.

To realize this potential of parallel computers, it is necessary to develop an efficient parallel molecular dynamics program. Such a program must be scalable, so that it can exploit newer generations of parallel machines with thousands of processors efficiently. Although the overall computational complexity of simulations is very large, the complexity results from the large number of time steps required. Each time step itself is relatively small. Since each time step must complete before the next one can begin, there is little opportunity to parallelize across time steps, requiring extensive parallelization of computation within each timestep. For example, simulating a biomolecular system with 3762 atoms sequentially required only about 1 second for a single time step in a particular simulation. To run this computation effectively on a large parallel machine implies that each time step must complete within tens of milliseconds. Given the communication requirements of molecular dynamics, and the relatively non-homogeneous nature of its computations, this becomes a challenging feat.

The program described in this paper, NAMD2, is one of the new parallel programs aimed at utilizing large parallel machines in a scalable manner. The nonbonded force computations, which constitute the dominant component of the overall computational cost, require calculation of pairwise interactions between atoms. In most common methods, a cutoff distance, R_C is used. Nonbonded interactions between atoms beyond this cutoff radius are either not calculated or calculated less often. This geometry-dependent nature of the computation complicates parallelization strategies for molecular dynamics programs. As described in section 2, NAMD2 uses a unique decomposition strategy that combines the advantages of spatial decomposition and force decomposition, which permits the program to utilize a large number of processors.

In addition to speed, maintainability and extensibility are also required in a parallel molecular dynamics program. Scientific studies of biomolecular systems often involve novel approaches. For example, one may be interested in steering small groups of atoms through a molecular system to study the forces experienced by them. Studies of specific systems may require periodic or non-periodic boundary conditions. Alternative numerical integrators are employed for increased accuracy or speed. A well-motivated application-domain programmer should be able to extend the parallel program to permit such experimentation. This requires a highly modular organization for the program. Ensuring such modularity is more difficult in a parallel program than a sequential one, because the new algorithm must correctly interleave the parallel communication logic. NAMD2 handles this problem in part by providing the infrastructure (class hierarchy) that embodies commonly needed molecular dynamic mechanisms. Parts of this infrastructure include generic force computation objects along with *patch* (see section 3.1.1) objects representing cubical regions of space. A numerical integration scheme, for example, can be written solely from the point of view of all the atoms within this cubical region, ignoring all other aspects of parallel communication. A scheme for maintaining *proxy* patches on each processor ensures that data communication is not visible to other components of the program. The generic force computation objects make it easy to add new types of forces, such as those required in steered molecular dynamics or free energy calculations (sections 5.3.1 and 5.3.3). This modular organization is further enhanced by using C++, a language that encourages modularity, and Converse, a portable parallel runtime framework that allows multiple parallel paradigms to co-exist in the single application. As a result, each module can be written in a parallel paradigm most appropriate to it. This extensibility-oriented design is elaborated on in section 3, which also further explains the parallelization strategy, and the specific algorithms used in various components of NAMD2.

Although the parallelization strategy enables scalability, a good load balancing scheme is necessary to achieve good performance. NAMD2 uses an adaptive measurement-based periodic load balancing strategy that is effective in achieving good performance even for difficult MD simulations (e.g. those involving nonperiodic configurations with high variations in atom density). This strategy and the results of some performance evaluation studies are discussed in section 4. The extensibility of NAMD2 design is demonstrated in section 5, which discusses several new features added to NAMD2. Section 6 contains a brief summary of simulation applications carried out using NAMD2 so far, which is followed by some concluding remarks.

2 Parallelization Approaches for Molecular Dynamics

The molecular dynamics computation involves calculating forces on all atoms during each time step, and "integration" — using these forces to update the positions and velocities of atoms. The integration requires a comparatively small fraction of time, and is a completely local operation. The force computations can be broadly divided into two categories: bonded force computations and non-bonded force computations. There are several categories of bonded forces, yet they all involve between two and four nearby (bonded) atoms. Non-bonded forces are due to Lennard-Jones and electrostatic interactions between atoms. Non-bonded forces decrease as the distance between atoms increases. Also, any given (sufficiently large) region of space is likely to be charge-neutral, thus reducing its effect on farway atoms. As a result, many simulations use a cutoff radius R_C to save computation time: non-bonded interactions between atoms beyond this distance are not considered. For some simulations, the cumulative effect of far-away interactions cannot be ignored. Even in such simulations, a cutoff distance can be used to reduce computation: Forces due to atoms beyond the cutoff distance vary relatively slowly, and therefore can be computed less often. It is therefore appropriate to focus our attention on cutoff based simulations for the purpose of discussing alternate decomposition strategies.

The non-bonded computations constitute between 80 and 95 percent of the overall computation, depending on the cutoff radius used. (When far-away interactions are not being ignored, one can often use a cutoff radius as small as 8 Å, compared with 12 to 18 Å used in cutoff simulations.)

Although the non-bonded force computations are dominant, one cannot ignore the computational cost of bonded forces in designing the decomposition strategy. For example, if one were to completely sequentialize their (say five to twenty percent) computations, the overall speed up would-be limited to at most twenty. We aim at making the program *scalable* to a larger number of processors.

Given a simulation of fixed size, there is naturally a limit to the number of processors one can use to achieve higher performance, i.e., no algorithm is continuously scalable. Instead, we use the following definition of scalability, based on [12, 15]. A parallel algorithm is said to be *isoefficiently scalable* if one can increase the number of processors used by it (P), and still retain its parallel efficiency by increasing the size of the problem being solved. (Thus, using a scalable algorithm, we will be able to solve larger problems with a larger number of processors).

The parallel efficiency η is defined through the following equation:

$$\eta = \frac{\textit{sequential time}}{P \times \textit{parallel time}} = \frac{T_s}{P \times T_P}$$

For a solving a particular problem, when the number of processors is increased, the efficiency will normally decrease (due to increase in communication costs, for example). But often when we increase the problem size, the ratio of communication to computation decreases, leading to improved efficiency.

Note that the parallel execution time T_p is at least: $T_p = \frac{T_s}{P} + T_{communication} = \frac{T_s}{P}(1+\gamma)$ (where γ is the communication to computation ratio).

For an algorithm to be scalable, the following must be true: Assume that the algorithm runs with efficiency η while simulating N atoms on P_1 processors. Then, given a larger number of processors P_2, one should be able to find a problem size N_2, such that the efficiency is η while simulating N_2 atoms on P_2 processors.

From the above equations, then:
$\frac{T_{s1}}{P1 \times T_{P1}} = \frac{T_{s1}}{P1 \times T_{P1}} \rightarrow \frac{T_{s1}}{P1 \times \frac{T_{s1}}{P1}(1+\gamma_1)} = \frac{T_{s2}}{P2 \times \frac{T_{s2}}{P2}(1+\gamma_2)} \rightarrow \frac{1}{(1+\gamma_1)} = \frac{1}{(2+\gamma_2)} \rightarrow \gamma_1 = \gamma_2.$

Thus, for efficiency to remain the same while increasing the number of processors, we must be able to find a larger problem size for which γ, the communication-to-computation ratio, remains the same. We next analyze several strategies for parallelizing molecular dymamics computations from this viewpoint of scalability. [1]

One of the earliest strategies used for parallelizing MD computations was to replicate all data (arrays containing attributes, coordinates and forces of atoms) on each processor. The force computations can then be evenly distributed across processors at will, as any processor is capable of carrying out any particular force computation. If there are N atoms, and P processors, the O(N) forces accumulated by each processor must be added up across all processors. This requires communication time proportional to $N \log P$. Assuming cutoff, the amount of computation required is proportional to N (with a large proportionality constant), leading to N/P computation per processor. As the parallel execution time of the program is the sum of its computation and communication times, the parallel efficiency is affected by the communication time. However, the

[1]In the following analyses, we focus on the volume of communication, i.e. the number of bytes of data sent or received by each processor. However, the cost of each message includes a relatively large startup cost on most current machines. This would argue in favor of counting the *number* of messages, rather than their volume. However, two factors allow us to focus on the volume. First, several message combining techniques exist that allow one to reduce the total number of messages. For example, If each processor is required to send a separate message to all other processors, this may require P messages in a straightforward implementation. But this can be reduced to $\sqrt{P}$ or even $logP$ messages, using techniques such as dimensional exchange. Secondly, in many of the algorithms analysed here, (especially the spatial decomposition variants) the number of messages is a constant, thus obviating the need to consider it as a factor in asymptotic analysis.

ratio of communication to computation times for replicated data (RD) is $P \log P$, and is independent of N. So, if we want to simulate a system twice as large as the current one, we cannot hope to double the number of processors to retain the same efficiency, because the fraction of time spent in communication will be larger. Thus, RD is said to be nonscalable. However, in practice, RD works effectively for tens of processors, and has the further advantage of being easier to implement, especially for an existing serial program. As a result, this strategy is used by many production quality MD programs, several of which are listed in Table I.

A simple alternative to replicating data is atom decomposition (AD). The array containing atoms is arbitrarily partitioned across processors. Atoms that are close by in space need not be close by in the array. As a result, each processor may need data from all the other processors, leading to a $O(N)$ communication cost per processor, and loss of strict scalability.

Force decomposition (FD) is a technique that distributes the sparse force matrix in a blockwise fashion across processors. Each processor's share of the $N \times N$ force matrix is a block of size $(N/\sqrt{P}) \times (N/\sqrt{P})$. To compute this block the processor needs the coordinates of $2N/\sqrt{P}$ atoms, which come from $\sqrt{P}$ different processors. The communication cost per processor is thus $O(N/\sqrt{P})$ leading to the communication to computation ratio of $\sqrt{P}$. While this is much better than RD, it is still not scalable according to our strict criteria: as we increase the number of processors, the proportion of communication cost increases even if we simulate a larger biomolecule. Research by Plimpton [38, 37] and Hwang *et al* [19] shows that this method provides a better speedup than RD, and can be used with good speedups up to hundreds of processors. Results obtained by Hwang show that the proportion of communication cost goes from 6 percent of the computation cost on 32 processors to 36 percent on 128 processors in their particular implementation.

As the force matrix is sparse, and the atoms that are next to each other in the array are typically physically closer (note that the converse is false), one may get a nonuniform distribution of work with FD. Plimpton *et al* suggest randomly reordering the atoms to eliminate all traces of spatial locality, and restore load balance. (To handle symmetry arising out of Newton's third law, they compute the interaction between atoms i and j in one of the two possible blocks containing $F[i, j]$ and $F[j, i]$ respectively, depending on whether $i+j$ is even.) Hwang *et al* use a diametrically opposite technique: they use recursive bisection to partition the atoms so that nearby atoms are assigned to the same processor, and reorder the force matrix accordingly. The resultant load imbalance is handled by assigning an irregular-shaped piece of force matrix to each processor. The effect of this scheme has not been quantified formally. Empirically, the communication costs are seen to be smaller than AD.

The idea of assigning nearby atoms to the same processor, called spatial decomposition (SD), has been used and elaborated upon by several researchers. There are broadly three ways of doing this:

- Partitioning space into P boxes, one per processor.
- Partitioning space into fixed-size boxes, with dimension larger than the cutoff distance R_C, requiring communication only between neighboring boxes.
- Partitioning space into a large number of small boxes, requiring each box to communicate with a large number of boxes.

With the first option, with sufficiently large N, the communication cost is proportional to the surface of the box, while the computation cost is proportional to its volume. Thus, this algorithm is highly scalable. However, the number of atoms is usually not as large as to validate this approximation. Also, this method is hard to use if the number of processors cannot be factored into three roughly equal factors.

With the second option, each box needs to communicate with a constant number (26) of neighboring boxes. If multiple boxes happen to be mapped to each processor, each will have to communicate with even fewer processors. Thus, the communication is proportional to N/P, and the algorithm is scalable with linear isoefficiency: If the program is running with a high parallel efficiency, and one doubles the size of molecule as well as the number of processors, one is assured that there will be the same number of boxes per processor and, therefore, the same parallel efficiency. This method was used in the earlier version of our program [29]. As other researchers have noted, load imbalance can be a severe problem for SD, especially for simulating non-periodic systems. NAMD addressed this problem by assigning the force computations corresponding to a pair of neighboring cubes (called patches in NAMD) to the patch on a processor with lower load. Although this strategy improves the load balance somewhat, it is clearly limited: neighbors of low-load patches (near the boundary) tend to be low-load as well. A second potential problem is that the number of processors one can utilize is limited to the number of patches.

The third strategy, implemented in EulerGromos by Clark *et al* [8], involves using boxes with sizes smaller than the cutoff distance, so that each box needs data from non-neighboring boxes. Again, multiple boxes can be mapped to each processor. The large number of messages resulting from the scheme can be reduced (but not the volume of data transferred) using a multi-stage communication scheme, first described in [36, 43] (called the "north–south–east–west" exchange algorithm; also called the "shift algorithm" in [8]). A prototype implementation of EulerGromos has demonstrated good speedups with the strategy. The programs UHGromos (with replicated data) and EulerGromos were written by the same team, and a performance comparison showed that EulerGromos was faster if there are enough processors, the crossover point in their experiment being 64 processors [8]. In our own experimentation, we found this "small boxes" strategy to lead to high overhead, possibliy due to the following reason: when multiple time stepping is used, the cutoff distances are short (e.g. 8-9 Å). As a result, boxes with the width half of the cutoff distance are too small (with about 12 atoms per box) to amortize the overhead.

In another variant of spatial decomposition, the FAMUSAMM algorithm implemented in latest versions of EGO [16] uses hierarchical decomposition of space, based on structural features of biomolecules, to implement a structure-adapted multipole algorithm.

The strategy used in NAMD2 can be thought of as a synthesis of space decomposition and force decomposition. NAMD2 decomposes atoms into boxes of size $= R_C + margin$, and distributes these boxes to processors as described above. (The margin parameter will be explained in section 3.1.1.) In addition, NAMD2 allows the computation of nonbonded forces between each pair of neighboring boxes to be freely assigned to any processor. The number of such pairs is 13 times the number of boxes. Thus, in principle, NAMD2 can utilize a much larger number of processors than a purely spatially decomposed program (such as the original NAMD). Furthermore, the ability to assign each pair to any arbitrary processor allows fine-tuning via load balancing. This strategy, and its object oriented implementation, is elaborated in the section 3.

2.1 Parallelization of Bonded Force Components

Parallelizing computation of bonded forces in the context of spatial decomposition is also challenging. Bonded forces represent covalent chemical bonding, and are parameterized on distances, angles between atom triplets (angles), and angles between planes formed by four atoms (dihedrals and impropers). Here we will discuss the parallelization of angles.

We can assume that the three atoms involved in an angle are within a $2 \times 2 \times 2$ box of neighboring patches. This is valid as long as bond lengths are smaller than half of the patch dimension. Assume that each patch is on a separate processor. The simple implementation (used in the original version of NAMD) is to have each patch send atom coordinates to each of the 26 neighboring patches. Each patch looks at the neighboring coordinates and computes the forces on its atoms due to the angles associated with the neighboring patch data. In this scheme these computations are duplicated for those angles involving atoms in multiple patches.

A simple reorganization of the computation allows one to reduce the number of processors to which each patch must deliver position data. Figure 1 illustrates the algorithm in a two-dimensional decomposition. The older algorithm would require patch $(1,1)$ to send eight position messages. The new algorithm uses the rule that the angle forces among three atoms in patches (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) are computed on the processor owning patch $(\min(x_1, x_2, x_3), \min(y_1, y_2, y_3), \min(z_1, z_2, z_3))$. Note that this may be a processor where none of the three atoms reside. A patch (a_1, a_2, a_3) sends its data only to *downstream patches* with coordinates (b_1, b_2, b_3) such that $a_i - 1 \leq b_i \leq a_i$. In the two-dimensional case of figure 1, patch $(1,1)$ needs to send data to only three other processors, $(1,0)$, $(0,1)$, and $(0,0)$. So now an angle composed of atoms in patches $(1,1)$, $(2,1)$ and $(2,0)$ is calculated on the processor that owns patch $(1,0)$. In three dimensions, there are seven downstream patches with whom each patch communicates (instead of 26). In addition, this algorithm avoids redundant computation by applying Newton's third law, although this increases the number of messages to 14 per patch to return the computed forces. Even when there are multiple patches per processor, the new scheme still results in fewer messages than the straightforward implementation. This method has similarities to work done by Brown *et al* [7], except that Brown's method requires additional communication, including a global reduction to guarantee that each angle is computed on exactly one processor. They also use a staged communication strategy, also described by Clark [8]. This strategy reduces the number of messages from seven to three by combining messages, but the *amount* of data exchanged remains the same. Esselink and Hilbers [10] consider a similar scheme, but opt in favor of one requiring communication from half of the neighbors within the cutoff distance, citing the complexity involved in identifying the angle forces to be calculated on a processor. Our scheme, along with the tuple search algorithm described in section 3.2.2, demonstrates that an efficient implementation is possible.

3 Design

The three overall goals of NAMD2 were improved sequential performance, scalability with respect to both simulation size and number of processors, and modifiability. We also wanted the program to deal with periodic as well as nonperiodic configurations, and provide the option of calculating long-range electrostatic forces. Modifiability was required to support ease of maintainance as well as the ability to add new features. Each of the design decisions described below contributes to achieving these goals.

The hybrid spatial/force decomposition scheme described above provides the basis for scalability of NAMD. This scheme requires that a number of entities (multiple "patches" and force computations) reside on each processor. Instead of a monolithic program that orchestrates all these diverse actions on a single processor, we chose a message-driven object based paradigm. Each of the computational entities were implemented as objects, and they are scheduled based on availability of data needed for their continued execution. With this, a force computation (for example) is scheduled for execution only when all the data it needs is available on the local processor, thus avoiding the possibility that any particular entity will block the processor while waiting for specific data.

NAMD2 is written using Charm++ [24], a parallel C++ extension that supports message-driven execution.

Like the original version of NAMD, NAMD2 uses CHARMm force fields and X-PLOR coordinate and molecular structure files. In addition to non-periodic simulations, NAMD2 can use periodic boundary conditions over any combination of the three coordinate axes. It performs cutoff simulations or full-electrostatic simulations employing multiple timestepping using the DPMTA [40] or DPME [46] libraries. NAMD2 can connect to VMD [18], the visualization component of MD-SCOPE [30], to allow monitoring of and interaction with ongoing simulations.

We will first describe the core structure of NAMD2. The specific algorithms and interesting data structures used for force computations are described next. To make a modular implementation of NAMD2 possible, it was necessary to utlize and integrate several different parallel programming paradigms. How such multiparadigm programming was supported and why it is useful is discussed in section 3.3.

3.1 Object-Oriented Design

For the sake of extensibility, we designed a class hierarchy that incorporates the structure of molecular dynamics. The core of this hierarchy is constituted of *Patch*, *Compute* and *Sequencer* objects. Patches represent cubical (actually, orthorhombic) regions of space, and all the atoms within such a region. *Compute* objects signify computation of different kinds of forces on atoms in a set of patches. A *Sequencer* specifies the numerical integration scheme, and drives the simulation of atoms within one patch. These base classes, and associated service code, define basic functionality needed in any MD computation. A programmer wishing to add features to NAMD2 can often overload an existing class to add some particular functionality.

3.1.1 Patches

The simulation space is divided into regular cubic regions called *patches*, each of which is responsible for atoms within its boundaries, their coordinates, and the forces exerted on them. The dimensions of the patches are chosen to be slightly larger than the cutoff radius. Thus one can make the assumption that each patch only needs the coordinates from the 26 neighboring patches to compute non-bonded forces. To ensure that all atoms in a patch indeed belong there, it is necessary to check all atoms at the end of each timestep, and migrate them to their proper home patches, if needed. This will be prohibitively expensive. Using a well-chosen margin allows one to optimize this: With margins, even if an atom strays outside the coordinate space of its patch, the assumption stays valid as long as the straying is less than the margin. NAMD2 uses 4-8 timesteps before migrating atoms to new locations, and detects if there are any violations of the above assumption. The margin is normally chosen to be about 1.5Å.

3.1.2 Compute Objects

The compute objects, together, are responsible for all the force computations in NAMD2. Each compute object is associated with 1, 2 or more patches. The compute object utilizes coordinates and other information from these patches, calculates a specific type of force (e.g. nonbonded), and provides its client patches with the forces it calculated. The runtime code manages dependencies: once a compute object has informed the system of the patches it depends on, the system triggers each compute object when its client patches are ready with the coordinates for the next time step. After the forces have been calculated, the compute object informs the patch so that it can take the next step in its algorithm. Adding a new type of force computation thus becomes a simple matter of specifying which patches a compute depends on, and providing the code to calculate the new force terms.

3.1.3 Proxies

If a compute object and the patches it depends on reside on differrent nodes, then the compute object must get the coordinate information from the patches on remote processors, and send forces back to them. Often, there will be multiple compute objects on a processor that are dependent on data from the same remote patch. To avoid redundant communication in this common case, a representative of the home patch called a *proxy patch* is created on the procesor. The compute object then contacts the proxy patch for the atom positions, and deposits forces into the local proxy as well. Thus each patch sends all its coordinates to its proxies, and receives forces from them once per time step. As the average number of proxies is often substantially smaller than the number of compute objects that depend on a patch, this represents a considerable optimization. In addition, the proxy patch provides the compute objects with the same interface as the home patch, enhancing the ease of programming: except for the code that deals with updating coordinate information in proxies at the begining of each timestep, and sending forces back at the end, the rest of the program can be written without any communication code. (Some communication is needed to form global sums of various energy terms, but again each object simply deposits its contribution into an existing library which does the necessary communication).

3.1.4 Sequencers

Every home patch has a *sequencer thread* associated with it. It loops over all the timesteps in the simulation and essentially describes the life cycle of a single patch. The sequencer provides the strategy, while the other objects such as the compute objects and patches provide the mechanisms. The *sequencer* abstraction allows it to be replaced by alternate modules for experimenting with different algorithmic strategies, thus enhancing modifiability– one of the primary design goals of NAMD2.

The pseudo code for the sequencer algorithm is shown in figure 3. The algorithm uses multiple time stepping to reduce the computational cost of computing electrostatic forces due to all pairs of particles in the system. The total electrostatic force acting on each atom is divided into two parts, a short-range (local) component consisting of forces due to pairs that are seperated by less than the local interaction length (electrostatic cutoff) and a long range component consisting of the remaining pairs. The long-range forces vary slowly, hence they are evaluated less frequently.

In each timestep, depending on the frequency parameters specified by the user, it either computes

1. only the bonded forces, or
2. bonded as well as non-bonded forces within a cutoff, or
3. bonded as well as non-bonded forces

The velocities are updated based on the forces, velocities in turn are used to update the positions. The sequencer then informs the home patch that its positions are ready. At every cycle boundary, the patch sends the atom positions that don't belong to its space to the neighboring patches. The proxy patches are populated next. The home patch sends all its atom positions to all its proxies.

Once the proxies are populated, it notifies all the dependent compute objects that the coordinates are available for computing the forces for the next timestep and suspends the sequencer thread. The compute objects calculate the forces and awakens the sequencer. The newly calculated forces are used to update the velocities, and reductions and collections are initiated for the calculation of total energy and trajectory output.

3.2 Force Computation Schemes and Algorithms

The specific algorithms used to carry out and, in the case of bonded forces, partitioning the computation, are influenced by the parallel decomposition scheme. The methods discussed below contribute significantly to both the serial and parallel performance of NAMD2, as well as to its modifiability and clarity of design.

3.2.1 Non-bonded Interactions

The calculation of local (cutoff) electrostatic and Lennard-Jones components of the force field is the most expensive step in a simulation. As such, it must be calculated both efficiently and in a manner which can be readily parallelized and load-balanced. We have enabled parallelization within the NAMD2 framework by dividing most of the non-bonded calculation between two classes of compute objects: self and pair. The self compute objects calculate pairwise interactions between atoms residing within a particular patch. The pair compute objects calculate pairwise interactions between pairs of atoms residing in neighboring patches. The size of a patch is guaranteed to be sufficiently large that all atoms not in neighboring patches are beyond the cutoff distance. A third class, the exclusion compute objects, efficiently handles the modifications to the non-bonded force field related to the connectivity of the molecule.

The non-bonded force field used in biomolecular dynamics is complicated by the bonding structure and variety of atom types present in the molecule. Non-bonded interactions are generally excluded (set to zero) for pairs of atoms which are directly bonded (1-2 pairs) or are bonded to a common atom (1-3 pairs). In addition, these interactions are modified for pairs of atoms separated in the molecule by three bonds (1-4 pairs), usually by scaling down the electrostatic interaction and providing a second set of Lennard-Jones parameters. Testing for these full and modified exclusions during the non-bonded force evaluation may be made more efficient by first constructing a list for each atom of all other atoms from which it is excluded, and additional improvements are described below.

In most molecular dynamics programs, not every pair of atoms is tested against the cutoff distance at every timestep, since this operation scales as N^2. Often, a *pair-list* [48], a list of atom pairs within or near the cutoff, is generated periodically. Alternately, *link-cells* [17] may be used, grouping atoms into cells by location and calculating interactions between atoms in pairs of nearby cells. Link-cells may also be used to efficiently generate a pair-list. The exclution list is generally used during pair-list construction to eliminate or flag as modified certain pairs of atoms.

Patches function as link-cells to provide an efficient method of distance checking. Although the original version of NAMD did use pair-lists, the additional group-based optimizations described below make pair-lists unnecessary in NAMD2. However, without pair-lists an expensive exclusion check would need to be performed for every pair of atoms within the cutoff at every timestep. The solution to this problem is to first calculate all non-bonded interactions without regard to exclusions, and then correct the interactions between excluded pairs by iterating over a list of those pairs directly. The parallelization of the second step is the role of the exclusion compute objects, which are identical in implementation to the bond compute objects discussed below.

There is, unfortunately, one additional complication. While subtracting the electrostatic interaction between bonded atoms is reasonable, adding and subtracting the $1/r^{12}$ repulsive term of the Lennard-Jones potential between covalently bonded atoms would result in an unacceptable loss of numerical precision since such atoms are much closer than energetically possible for non-bonded pairs of atoms. We define therefore, an additional cutoff outside of which all exclusions are subtracted by the exclusion compute objects, and inside of which exclusion checking is performed to avoid loss of precision.

Hence, the algorithm carried out by the non-bonded self and pair compute objects is:

```
for all pairs of atoms in the same or neighboring patches:
   if the atoms are within the cutoff:
      if the atoms are within the exclusion cutoff:
         if the atoms are 1-4 excluded:
            calculate modified interaction
         else if the atoms are not excluded:
            calculate normal interaction
      else:
         calculate normal interaction
```

and the complementary algorithm of the exclusion compute objects is:

```
for all pairs of excluded atoms:
   if the atoms are beyond the exclusion cutoff:
      calculate and subtract normal interaction
      if the atoms are 1-4 excluded:
         calculate modified interaction
```

NAMD2 takes advantage of the typical biomolecular structure in which one, two, or three hydrogen atoms are bonded to every heavy atom by ensuring that all atoms in such a “hydrogen group” are on the same patch and performing a preliminary group-based cutoff check. Comparing the distance between the oxygen atoms in a pair of water molecules against an extended “group cutoff” is nine times faster than comparing all pairs of atoms. Although this speedup must be balanced against the cost of increasing the patch dimension by the maximum hydrogen group diameter, a significant improvement is obtained because at best only 19% of cutoff checks succeed (in a uniform, periodic system with patch size equal to cutoff). The initial NAMD2 design used patches smaller than the cutoff length to increase this success rate, but early experiments demonstrated that the overhead from the factor of eight increase in the number of patches and the factor of five hundred increase in the number of compute objects greatly exceeded the expected gain. NAMD2 also exploits the fact that nonbonded interactions are excluded among all atoms in a hydrogen group to bypass the exclusion algorithm described above, eliminating all exclusions for water molecules and many for biopolymers.

Support for full electrostatics and multiple-timestep integrators in NAMD2 requires that the non-bonded code provide a variety of services. For cutoff simulations, a shifting function is applied to the electrostatic term to allow energy conservation. (A similar modification is always made to the Lennard-Jones potential.) The fast multipole algorithm (FMA) calculates full electrostatics for all atoms, but the non-bonded code must still subtract off exclusions and apply a smooth splitting function to remove the short-range part from the full electrostatics forces and to remove the long-range part from the cutoff electrostatics which are calculated every timestep. The electrostatic splitting function is different from the cutoff shifting function, and a variety of splitting functions are available. The particle-mesh-Ewald (PME) or particle-particle-particle-mesh (PPPM) algorithms use FFT's to calculate long-range electrostatics forces to which an algorithm-specific short range component must be added before performing the same operations required for FMA.

For efficiency, all calculations on pairs of atoms which employ a cutoff-check should be performed in a single step to avoid repeated distance calculations or (even if a pair-list is used) repeated memory accesses. The use of if-statements within an inner loop like this would degrade performance, even on modern CPU's with branch prediction, and impede optimization by the compiler. Hand-writing separate code for all cases would result in a maintenance nightmare of 27 different functions (self, pair and exclusion classes; cutoff, FMA, and PME algorithms; three splitting functions). Instead, the C preprocessor was used to parse a single, easily maintained function definition containing flags to differentiate the different methods. This single function definition allowed hand tuning of the initial cutoff check through explicit pre-fetching of atom coordinates, which resulted in significant performance improvement. Given sufficient compiler support, template meta-programming could probably be used to achieve the same end with better error-checking.

3.2.2 Bonded Interactions

A trivial operation in a serial program, the efficient calculation of the bond, angle, dihedral, and improper terms of the force field is complicated by the separation of atoms into patches. The atoms required for a single dihedral calculation may reside on up to four patches. The calculation of these bonding terms must be divided among compute objects on different nodes without introducing excessive communication or overhead.

In NAMD2, a single compute object for each type of bonded interaction (bond, angle, dihedral, improper, and exclusion) is created on each node. This compute depends on all patches local to that node and the proxies of non-local patches which are *upstream neighbors* of local patches, defined as being adjacent to a local patch while having greater or identical x,y, and z coordinates. The proxies of upstream patches are also sufficient to calculate non-bonded interactions, so no extra communication is necessary (although migrating compute objects to improve load balance will require additional proxies). All of the coordinates for a particular bonded interaction are guaranteed to be present on the node containing their *common downstream patch*, the patch with the minimum x, y, and z coordinates of the patches containing the atoms of that interaction. For example, the coordinates for an interaction involving atoms on patches at (2,3,4), (3,3,4), and (3,2,4) are guaranteed to be on the node with patch (2,2,4).

The algorithm for determining which bonded interactions should be calculated on which node can have a major impact on performance, often a negative one. In NAMD2, the bond, angle, dihedral, improper, and exclusion compute objects all derive from a single template base class which treats bonded interactions as generic N-tuples (or "tuples"). This permits easy tuning of the interaction-selection algorithm which must be run whenever atoms migrate between patches. To support this algorithm, lists are maintained on every node for every type of tuple identifying for each atom those tuples in which it is the first atom. A map is also maintained to allow the efficient lookup of the patch and local index, given the global index of an atom present in any home patch or proxy on that node. The tuple-selection algorithm is as follows:

```
for all home or upstream patches:
   for all atoms in this patch:
      for all tuples with this atom as the first atom:
         look up the patches containing atoms in this tuple
         calculate the common downstream patch of these patches
      if the common downstream patch is local:
         add this tuple to the list of interactions to calculate
```

Developing a good algorithm for dividing the bonded calculation among the processors turned out to be much more important than anticipated. The original design called for every patch to have access to all of its neighboring patches. Bonded forces would be calculated for all patches on a node, but no forces would be returned to proxies. This eliminates the need for a point of synchronization and allow proxies to return results as soon as possible. The current algorithm was devised to more than halve communication when running on large numbers of nodes while eliminating redundant force calculations.

3.2.3 Periodic Boundary Conditions and Isobaric Methods

Periodic boundary conditions are dealt with as much as possible at the patch/proxy level. All atoms within a patch are kept in a single consistent coordinate system with periodicity accounted for when atoms are migrated across cell boundaries. When a non-bonded pair compute object requests coordinates from a patch it also specifies a unit cell translation necessary to bring both of its patches to the same coordinate system. All required transformations are performed by the patch or proxy and cached for other compute objects requiring the same calculation; transformed coordinates are never communicated between nodes. Compute objects for bonded interactions, however, may have to deal with the entire system at once and hence use a utility function to find nearest-image vectors.

To allow implementation of constant pressure simulation methods which rescale the periodic cell, the patches provide standard methods for rescaling coordinates, cell basis vectors, and velocities. Compute objects are expected to not store the periodic cell definition between invocations and are provided a correct version by the each patch at every timestep. This allows a variety of constant pressure algorithms to be implemented without modifying either the patches or the compute objects.

3.3 Multiparadigm Programming Approach

One of our major design decisions to use the multiparadigm approach for implementation of NAMD2 was based on the following requirements:

- NAMD2 must be able to utilize the DPMTA module which is written in PVM [45], without having to pay the overhead of interacting with separate PVM processes.
- The performance-critical core of parallel computation must be written in Charm++ [25], to take advantage of its message-driven execution for adaptive overlap of communication with computation, and the ability to prioritize messages.
- The coordination logic of the program must be modularly separated from the parts of the program that implement the parallel mechanics needed to execute it.
- Nearly half of the code involved parallel initialization, using synchronous message passing. Reusing code from the original NAMD without extensive modification was desirable.

Traditional implementations of these paradigms cannot work together in a single application, because the control-regimes for these paradigms are seemingly incompatible. Fortunately, Converse [23], described in the next subsection, allowed us to use these different paradigms in NAMD2.

3.3.1 Converse: An Interoperable Runtime System

Converse provides a portable framework that supports multilingual parallel programming by using (a) a common scheduler's queue for entities across languages, (b) by providing a customizable scheduler that is exposed to language runtimes, and (c) by providing a component-based architecture with support for many common building blocks of parallel runtimes.

A specific language implementation in the Converse framework incorporates only the necessary components, customizing them as needed, thus incurring minimal overhead. The components provided by Converse include a machine interface (that supports communication, timers and other operating system calls), scheduler queues, a threads package, a message manager, and a load-balancing package. Over a dozen languages based on different paradigms, including Charm++ and PVM, have been implemented on top of Converse [26].

Converse has been implemented on several parallel platforms, including workstation cluster, IBM SP2, Convex Exemplar, SGI Origin 2000, Cray T3E, and Intel Paragon. Porting Converse to a new machine is relatively easy because only a small machine-dependent component needs to be rewritten.

3.3.2 Paradigms Used in NAMD2

Using different languages for implementing different modules, we were able to address all of the above requirements (see figure 4.)

The initialization code involved reading in the data from the disk, and installing individual data structures on the parallel processors. To avoid race conditions, it used synchronous message passing, with tagged messages and blocking receives. SM language provides such message passing with lower overhead than PVM or MPI. Also, in SM, an innovative use of message streams allowed us to streamline messages without fully assembling them. This resulted in major saving in memory requirements in the initialization phase.

We were able to reuse the DPMTA module by simply linking it with the implementation of PVM on Converse.

The remaining requirements presented a major challenge. Writing the code for core computations entirely in Charm++ would lead to good performance, and also reasonable perspicuity. However, we wanted to separate the algorithmic logic from the parallel logic, so that someone wishing to change the basic numerical algorithm (e.g. Verlet-I integration, Langevin dynamics) can do so without being encumbered by the parallel logic. This was accomplished by observing that the algorithmic logic involves the description of the life cycle of a patch: sequencing of the steps of the algorithm, and choosing individual sub-algorithms for each step. In contrast, the parallel logic involves movement of data, triggering of computations based on the availability of data, and interprocessor synchronization. The latter can be thought of as mechanisms under control of the algorithmic logic, encapsulated in the sequencer object.

As there are multiple patches per processor, multiple sequencers must be active on individual processors, with the possibility that they may not be in lock-step. Some sequencers may be lagging behind, while others race ahead, constrained only by the data dependencies. Periodically each sequencer must wait for its data to become available, while one of the other sequencers on the processor can continue. Also, the bulk of parallel computations in Charm++ modules must be allowed to execute interleaved with the sequencers. This suggested the use of threads for implementing sequencers. Normally, it would be difficult to concurrently run multiple threads and message-driven objects in the same program. For one thing, Charm++ would use its own message-driven loop to schedule messages for its objects, which would conflict with the thread scheduler.

Running Charm++ in one of the threads is not a good solution, because of the stack space limitations imposed by threads, and the context switching overheads of threads which we wish to avoid for the fine-grained dominant computations. However, the threads implemented within Converse allow applications to mix threaded and non-threaded computations effectively. The common scheduler can switch between threads and objects, while allowing unlimited stack space to the objects.

In spite of the modular separation of the sequencing operations provided by the use of threads, two significant problems remained in using threads. The first problem was the overhead of thread context-switching, and the second was wastage of memory space due to over-estimating the stack size needed for threads. Our experiments comparing the overheads of thread context switching to the message-driven scheduling showed [22] that in many programs, this overhead can be a significant factor in program performance. Also, our efforts to run NAMD2 on machines with limited swap space indicated that many real parallel machines do not have enough memory to run thousands of threads. However, creating thousands of message-driven objects is almost always possible on these machines. Our research on creating a low-overhead and lightweight programming abstraction resulted in Structured Dagger, a coordination language for message-driven parallel programming [22]. We have since reimplemented one of the sequencer algorithms using this language.

4 Scalability and Parallel Performance

Although the previous section described how NAMD2 performs its calculations efficiently, if the load is not properly balanced, the program will not achieve good scalability. This section describes the load balancing algorithm used by NAMD2 and the resulting performance.

4.1 Load Balancing

The parallel efficiency of NAMD2 is highly dependent on efficient load distribution. Computational time is consumed mainly by two object types, patches and force objects. To distribute these objects, NAMD2 performs two types of load balancing, initial load balancing and dynamic load balancing.

Initial load balancing occurs at program startup. Patches are distributed using a recursive bisection algorithm, to distribute atoms approximately evenly, while still maintaining data locality. All of the bond force objects are distributed next, one per processor. Finally, the non-bonded force objects are distributed. Non-bonded force objects for patch self interactions are first placed on the same processors as their patches. Patch-pair force objects are next distributed to the *upstream processor* for the two patches. This guarantees that communication is initially minimized, since the same proxies required for the bond computations are also used for the non-bonded force objects. It does not necessarily result in an optimum load balance, since patch locality probably results in clusters of computationally intensive force objects being distributed to a small number of processors, but by minimizing communication, the strategy promotes effective dynamic load balancing.

NAMD2 relies on a dynamic, measurement-based load balancing scheme to achieve efficient throughput. Heuristic-based static load balancing schemes exhibit several disadvantages in molecular dynamics. The first difficulty we ran across was determing the computational load represented by a particular force object. Processor speed and inter-processor communication costs both affect the heuristic model of load necessary for static load balancing. Determining an accurate model for all the different architectures NAMD2 runs on was prohibitively time consuming. Therefore, NAMD2 was designed to measure the actual time consumed by various objects and use that data to redistribute these objects efficiently. This method produces the additional benefit that it can be repeated during a run, to react to changes in load distribution as the atoms move during the evolution of the simulation.

Migratable objects are those objects which can be moved to a new processor during a simulation. Ideally, any work should be assignable to any processor, but making objects migratable complicates their code. In NAMD2 we chose to make only non-bonded compute objects migratable. These objects consume the majority of the simulation time, yet there are enough of them to provide sufficient freedom to balance the load. The load balancer treats non-migratable work, including integration and bond force evaluation, as background load which does not change, but must be considered when distributing the migratable objects.

Simulations execute the following procedure to perform load balancing. First, the simulation runs for a small number of steps, typically lasting a few minutes. Migratable object times are measured during this time. In addition, the Converse system provides a means of measuring idle time for each processor, so non-migratable load is determined by $t_{non\text{-}migratable} = t_{total} - t_{idle} - t_{migratable}$. After a particular simulation step, the program collects the load data from each processor, computes a new load distribution, and moves the migratable objects.

The load balancing algorithm we selected uses a modified greedy algorithm to distribute load while keeping the amount of communication small. A min-heap stores the currently-assigned load on each processor, so the most lightly-loaded processor can be easily retrieved. The initial processor load is the measured background load for that processor. Migratable objects are also stored in a max-heap to allow the most expensive compute objects to be assigned first. The load balancer also uses a map of required proxies, ie., the proxies required for the calculations of bonded forces. The load balancer makes no distinction between patches and proxies, so when the following description mentions proxies, it also refers to patches. An average load is computed from the total measured load. Then each migratable object is examined, starting with the most expensive. The load balancer considers three possible locations for the compute: the least-loaded processor already receiving both proxies, one proxy, or no proxies. The object is assigned to the two-proxy processor if the resulting load on the processor is less than the average load times an overload factor. Otherwise the one-proxy processor is considered, and finally, if necessary, the no-proxy processor. Then the load balancer adds the object load to the selected processor's total load, and updates the proxy map. A fairly generous overload factor is selected, so that data locality is more important than exact load balance. This procedure repeats until all compute objects are assigned.

A refinement procedure is next used, to fine-tune the load balancing produced by the sorting algorithm. The load balancer makes a second pass through the processor assignments. A smaller overload factor is selected, and any processors whose load exceeds the average by more than the overload factor has a number of force objects moved to lighter loaded processors, using the same selection criteria described above. Since the threshold is now lower, this procedure is likely to add a few more proxies, but produces good load balance with small communication overhead.

After refinement, the simulation resumes using the new object placements. Although the new load balance is good, the changes in the communication patterns produce changes in the background load, which the previous load balancing did not take into account. Therefore, the simulation runs for a few more steps, then stops for a second load balancing. This time the load balancer only performs the refinement step, resulting in just a few adjustments to the object placement. The resulting load assignment produces well-balanced load distribution, as shown in figure 5.

4.2 Parallel Performance

NAMD2 has demonstrated good performance on many parallel machines. Table II shows the performance data obtained while running three benchmarks of varying sizes on the Cray T3E. The systems chosen range in size from 3,800 atoms to 92,000 atoms. The largest configuration used (ApoA-I) was a periodic simulation, while the other two were non-periodic. The tables show time per integration step in seconds, and the corresponding speed-up obtained. In some cases, the memory per processor was insufficient to allow runs on small numbers of processors. In those cases, the speed-up has been estimated using the speed-up factors obtained on smaller molecules. This is conservative, as the speed-ups typically get better when a larger number of atoms are involved on the same number of processors. Figure 6 demonstrates that NAMD2 achieves good scaling even on large numbers of processors, for sufficiently large simulations.

As can be seen, speed-ups around 100 were obtained with 128 processors, and they seem to increase with the same efficiency beyond 128 processors. The speed-ups were calculated in comparison with the one processor performance of NAMD2. This raises the question of how well NAMD2 performs sequentially. As table IV shows, the sequential performance of NAMD2 is better than its predecessor and X-PLOR, which was used as a specification for both NAMD and NAMD2. Although benchmark times of other popular MD programs suggest that NAMD2 is comparable in sequential performance and superior in speedup, we have not yet done experiments to compare the performance of NAMD2 with other programs for identical molecules with identical force-field parameters. We plan to perform these comparisons soon.

NAMD2 is designed to be portable, and uses a portable runtime framework (Converse). Porting NAMD2 to new machines still required us to modify the program to satisfy the vagaries of C++ compilers on different machines. Other than that, the porting was straightforward. In particular, no machine specific performance tuning was necessary. Table III shows the performance obtained while simulating ER-ERE on a variety of parallel machines.

Although the parallel performance of NAMD2 is quite good, it still leaves room for improvement. The efficiency on 128 processors is close to 80 percent. Although part of the efficiency loss is inevitable due to communication overhead, we believe that further improvement is possible. Our analysis indicates that about half of this loss (ten percent) is due to communication overhead and increased time for some of the force calculations when done in parallel. The other ten percent is idle time, spent waiting for messages. This is due to a combination of load imbalance and message scheduling. We believe that improvements to our load balancer, combined with the use of priorities for messages (which are supported by Charm++) will allow us to decrease the idle time further.

Some of the features of C++ do impose a speed penalty, so profiling information was employed to pinpoint which parts of the code penalized performance, and these portions were hand-tuned for best performance.

NAMD2 currently does not use pair-lists, relying instead on the fact that only a small subset of the atoms (those in the neighboring patches) must be examined for each atom. Although this strategy yields good performance, it is possible that either incorporating pair lists, or partitioning atoms into cells smaller than the patch-size will yield further performance improvements.

The current load balancer does not take the interconnection topology of the host computer into account. This is valid for many parallel machines, such as the Cray T3E. However, an increasing number of parallel machines exhibit a two-level architecture: a collection (from two to thirty-two) of shared memory processors, connected to other similar nodes via an interconnection topology. In other machines, the communication time (and bandwidth utilized) varies widely depending on which processors are communicating. We plan to develop a load balancing strategy that adapts itself to the communication architecture.

Our benchmarks above describe only cut-off electrostatics performance of NAMD2. The long-range interactions in NAMD can be computed using the DPMTA library developed at Duke University. The parallel performance of DPMTA has been documented in [40]. The performance of the long-range interactions phase is orthogonal to that of the rest of NAMD2. In the future, we plan to experiment with other methods for long-range electrostatics, such as particle-particle particle-mesh methods, and with tighter integration of such libraries to improve performance.

5 Extensions

This section describes a set of features added to NAMD2 on top of the basic design described above. In addition to demonstrating the added functionality of the program, the subsections below also illustrate the extensibility of the core design. Current extensions to the program center on full electrostatics methods, modifications and improvements to the integrator, and new ways for the user to control and analyze the simulation via additional forces. Not detailed below are simple single-body forces present in NAMD2 such as harmonic restraints and soft-wall boundaries.

5.1 Full Electrostatic Evaluation

NAMD2 provides three methods for incorporating long-range electrostatic forces into a simulation. The communication patterns employed by these methods do not correspond to the spatial decomposition used for short-range interactions and the code may be developed externally. Hence, a standard interface to these external parallel libraries has been developed. Every full electrostatics calculation is carried out by one compute object on each node which receives charges and coordinates for all of the atoms on that node. These objects are responsible for calculating a simple electrostatic potential and returning energies and forces for all atoms. No exclusions, cutoffs, or splitting functions are calculated in the full electrostatics modules.

5.1.1 Direct Calculation

The naive approach to electrostatic evaluation, in which forces are computed directly between all pairs of atoms, is primarily used for testing. The algorithm is implemented by passing coordinates and forces halfway around a ring, using Newton's third law to eliminate redundant calculations. Ewald summation is not implemented; the nearest-image convention is used for periodic systems. Despite scaling as N^2, direct calculation is more efficient for small systems than methods with better scaling, but more complexity.

5.1.2 DPMTA: Fast Multipole Algorithm

The variant of the fast multipole algorithm (FMA) [13] and Barnes-Hut treecode algorithms [2] implemented in DPMTA [40] begins by dividing simulation space into cells. Multipole representations of the charges within each cell are constructed and combined within an oct-tree to form a full hierarchy of representations. Multipoles from cells beyond a cutoff distance are used to construct a local Taylor series expansion of the electrostatic field within each cell, which is then used to calculate forces on individual particles. Forces between atoms in cells which are too close together to be treated as multipoles are calculated directly. Unlike many multipole-based implementations, DPMTA does support periodic boundary conditions. FMA scales linearly (under certain conditions, more generally it is of order $N \log N$) but with a large constant factor, making it expensive for small systems. In addition, the forces calculated by DPMTA change discontinuously as particles cross boundaries between cells. For this reason, greater accuracy and larger numbers of multipoles must be used for a simulation to conserve energy than would otherwise be needed to obtain a sufficient approximation of the long-range electrostatics of the system, further slowing the calculation. DPMTA is implemented with PVM and hence its incorporation would have been much more difficult without the multi-paradigm capabilities of Converse.

5.1.3 DPME: Particle Mesh Ewald

The particle-mesh-Ewald (PME) [9, 11] algorithm in NAMD2 is currently based on DPME [46]. As the name implies, this method is only applicable for periodic boundary conditions. Similar to particle-particle-particle-mesh (P^3M) [17, 28, 39], PME divides the electrostatic potential into a local interaction which must be calculated within a cutoff, and a smooth, long-range interaction which is evaluated on a grid via a three dimensional fast Fourier transform. Owing to the FFT, the complete algorithm scales as $N \log N$. Since the local interaction has a simple analytical form, it has been incorporated into the cutoff nonbonded evaluation mechanism of NAMD2. At present, all data for the FFT is collected on a single node rather that attempting to distribute the relatively inexpensive FFT. However, for the method to scale beyond tens of processors this serial bottleneck will have to be eliminated, although this may be accomplished by distributing the FFT among a small subset of the processors to limit communications. DPME is implemented such that the calculated forces are analytical derivatives of a smooth potential energy approximation. Therefore the required accuracy for a simulation will be determined by the required accuracy to capture the effects of long-range electrostatics rather than energy conservation. DPME also uses PVM, again reinforcing the utility of the multi-paradigm capabilities of Converse.

5.2 Alternate Integration Methods

One of the goals for NAMD2 was to allow the integration method to be obvious and modifiable. This was addressed in the design via the sequencer threads, which allow the parallel algorithm to be written sequentially. With a master sequencer thread to support operations on global quantities such as temperature and pressure, NAMD2 is capable of implementing most *explicit* isothermal and/or isobaric simulation methods using single or multiple timesteps. These features are currently being developed. Currently all of the production simulation algorithms are incorporated into a single sequencer function while an alternate test sequencer is used for verifying certain parallel methods.

5.2.1 Triple Timestepping

For greater efficiency, NAMD2 allows the force field to be split into three parts based on their frequency of variation. All bonded forces (bonds, angles, dihedrals, and impropers) are considered quickly varying, nonbonded forces (electrostatics and Lennard-Jones) within a cutoff are slower, and long-range electrostatics (separated from local electrostatics by a smooth splitting function) vary on the slowest time scale. In a typical simulation, the three parts of the force field would be evaluated on 1, 2, and 4 fs intervals, respectively, and the symplectic r-RESPA integrator [47, 50] would be used. In fact, long-range electrostatics vary much more slowly than 4 fs, but frequent evaluation is necessary to avoid resonance [4]. Although support for splitting forces into different classes is required throughout the code, the integrator itself is fully contained in the sequencer.

5.2.2 Rigid Bonds to Hydrogen

Bonds to hydrogen atoms, as well as H-O-H angle in water, can be fixed using the SHAKE/RATTLE algorithm [42, 1] to increase the timestep or implement certain water models. This calculation is localized to a single patch since hydrogen atoms are stored on the same patch as the heavier atoms to which they are bonded. Rigid bonds increase simulation speed by eliminating the fasted vibrations in the molecule, allowing the timestep to be roughly doubled to 2 fs. In contrast to triple timestepping above in which nonbonded forces are evaluated less often than bonded forces, rigid bonds avoid the extra communication required for bonded-only force evaluations. In addition, rigid bonds eliminate the highest frequency resonances in the system, allowing long-range electrostatics to be evaluated at 8 fs intervals.

5.3 Simulation Control and Analysis

The processes of interest in many biomolecules occur often on microsecond timescales. Therefore it is necessary to enforce a desired process in order to gain any information during the few nanoseconds of a typical MD run. With its modular design, NAMD2 has addresses this need well as demonstrated by the four methods currently implemented. The Steered Molecular Dynamics module seeks maximum parallel efficiency by performing most force calculations locally and only communicating across nodes when absolutely necessary. In contrast, the Tcl, free energy, and MD-Scope modules enjoy a simpler implementation by deriving from a common subclass which collects requested atoms to the master node and broadcasts forces generated by the module. Typically a small number of atoms are involved and this communication overlaps with the force computation, resulting in minimal effects on performance.

5.3.1 Steered Molecular Dynamics

Steered Molecular Dynamics (SMD) [14, 27, 20] is a simulation technique where time-dependent forces are applied to certain atoms of a molecule. Recording the applied forces and resulting positions over time gives insight into problems such as unbinding of ligands from proteins. SMD is implemented as additional force objects acting on each patch. The applied forces are combined with the normal bonded and nonbonded forces, so only isolated changes, and the creation of a new compute class were needed to add the fuctionality to NAMD2.

5.3.2 Tcl Scripting

In order to allow the end user to automatically monitor or manipulate a biomolecular system during a simulation without modifying the NAMD2 source code, the Tcl scripting language [32] has been incorporated. Although rudimentary, the interface allows the user to obtain the coordinates of particular atoms or groups of atoms (center of mass) and to apply forces to any atom or group of atoms at every timestep. The user is required to define an initialization function, which requests coordinates of particular atoms, and a force function to be passed these coordinates at every timestep. The force function can return forces for any atom and monitor simple quantitites based on the requested atoms. When additional utilities have been incorporated the module will be particularly useful for rapid prototyping of SMD methods.

5.3.3 Free Energy Calculations

The same mechanism used to add Tcl scripting to NAMD2 was readily extended to accomodate an externally developed free energy difference calculation [3] module from the group of Jan Hermans at the University of North Carolina. A script in the configuration file specifies distances, angles, or dihedrals to which additional restraining forces should be applied. These restraints are monitored and modified during the course of the simulation to calculate the free energy difference between two configurations of the molecule.

5.3.4 MDScope: Interactive Modeling with VMD

In MDScope[2] [31], NAMD2 is connected to the molecular visualization package VMD[3] [18], allowing the user to view a remotely running simulation and to apply forces to atoms, residues, or molecules. All atoms of the simulation are sent to VMD using the same coordinate collection methods used for trajectory output to a file. The forces currently provided by VMD are constant and atom-based, making no use of the ability to calculate forces based on coordinates. However, in future versions of MDScope the user will be able to restrain atoms to movable locations or to each other. Due to network lag, it would be unreasonable to have VMD update these forces at every timestep. At this point the force calculation framework developed for free energy calculations and Tcl scripting will pay off in simplified development.

6 Initial Applications

In order to demonstrate to the reader that NAMD2 is a useful, production-quality program, we now outline three studies to which it has been applied. The earliest of these was completed during the spring of 1997, while the more recent examples highlight the SMD features of NAMD2.

6.1 High density lipoprotein disk

High density lipoproteins (HDL) circulate in the blood of vertebrates, transporting cholesterol from various body tissues to the liver for excretion or recycling. HDL particles are protein-lipid complexes of apolipoprotein A-I (apoA-I), several minor proteins, phospolipids, cholesterol, and cholesterol esters. Reconstituted HDL (rHDL), developed in the Jonas lab, have provided the best opportunities to experimentally study the structure-function relationships of apoA-I because of their defined compositions and sizes [21]. Nascent rHDL particles consist of a phospolipid bilayer disk surrounded by two apoA-I molecules. The amphipathic helices of apoA-I shield the hydophobic lipid tails, solubilizing the rHDL particle in water.

The structure of rHDL has not been observed experimentally as protein-lipid complexes are extremely difficult to crystallize. Phillips *et al.*[4] have constructed a model of the lipid-binding domain of apoA-I in rHDL particles based on experimental evidence and sequence analysis [34]. The total system, comprising two apolipoproteins, 160 POPC lipids, and 6,224 water molecules, 46,522 atoms in all, was tested via simulated annealing using NAMD2 for a total of 250 ps. A larger, periodic simulation of rHDL in water containing 92,224 atoms has also been used as a test case for NAMD2.

[2]URL: http://www.ks.uiuc.edu/Research/mdscope/
[3]URL: http://www.ks.uiuc.edu/Research/vmd/
[4]URL: http://www.ks.uiuc.edu/Research/apoa1/

6.2 Binding Pathway of Arachidonic Acid in Prostaglandin H_2 synthase-1

The enzyme prostaglandin H_2 synthase-1 (PGHS-1) catalyzes the transformation of the essential fatty acid, arachidonic acid (AA), to prostaglandin H_2 [44]. Aspirin, flurbiprofen, and other non-steroidal anti-inflammatory drugs directly target PGHS-1 and inhibit the first step of this transformation by preventing access of AA to its cyclooxygenase active site. Based on the crystal structure of PGHS-1, with flurbiprofen bound at the active site, a model for AA embedded in the enzyme has been suggested, in which AA replaces the inhibitor [35]. Molnar *et al.*[5] seek to eludicate the folding of AA into the narrow binding channel of the cyclooxygenase site and to identify key residues guiding AA binding.

Steered molecular dynamics (SMD) calculations of enforced unbinding were carried out with NAMD2 on one monomer of the PGHS-1 homo-dimer, with AA bound in its putative cyclooxygenation site leading to the exit of the ligand from its narrow hydrophobic binding channel. AA contains four rigid *cis* double bonds connected to each other by a pair of conformationally flexible single bonds. The unbinding mechanism can be described as a series of rotations around these single bonds which leave the "rigid backbone" of the fatty acid formed by the conformationally inflexible *cis* double bonds relatively unaffected.

6.3 Modeling ligand binding to nuclear hormone receptors

Retinoic acid (t-RA) is an important regulator of cell growth and differentiation in both the adult and developing embryo. The effects of t-RA are mediated through the retinoic acid receptor (RAR) that binds all-trans-RA (t-RA) but the underlaying mechanism is still not known [33]. Kosztin *et al.*[6] study the transition between the bound and unbound form of the retinoic acid receptor, known from experiment to be accompanied by a conformational change that enables the hormone–receptor complex to bind to specific sequences of DNA and other transcriptional coactivators or repressors. The crystal structure of the ligand binding domain (the receptor domain responsible for recognizing and binding the hormone) of the human retinoic acid receptor hRAR-gamma bound to all-trans retinoic acid [41] is used for simulating different unbinding pathways. Examination of the crystal structure of the hRAR-gamma bound to t-RA suggests three entry/exit points for the hormone that were explored using SMD. In all simulations, the protein–ligand system was surrounded by a water bath, the total number of atoms being 15,000. One atom of the hormone was harmonically restrained ($K = 10$ kcal/mol Å^2) to a point moving with a constant velocity $v = 0.032$ Å/ps in a chosen direction. NAMD2 was used to compute three different trajectories of 750 ps each. The results of our simulations show that it is possible to extract the hormone out of the binding pocket with little or almost no effect on the protein structure if strong enough forces are applied. Along one of the pathways the hormone has to overcome the strong electrostatic forces between its carboxylate group and the charged Lys and Arg residues lining the opening. In the other two cases, before the hormone is completely out of the protein, the carboxylate group of the hormone interacts very strongly with some of the neighbouring residues. It may be speculated that those residues are the ones to attract and guide the hormone toward the binding pocket.

[5]URL: http://www.ks.uiuc.edu/Research/pghs/
[6]URL: http://www.ks.uiuc.edu/Research/pro_DNA/ster_horm_rec/

7 Conclusion

We have described the design of the parallel molecular dynamics program NAMD2. The program uses a novel combination of spatial and force decomposition to attain both theoretical and practical scalability. It uses an adaptive load balancing scheme to attain high parallel efficiency even while simulating non-periodic systems. It uses a highly modular design with a class hierarchy implemented in C++ that incorporates mechanisms needed in molecular dynamics codes while permitting easy experimentation with algorithmic strategies. Using the message driven language Charm++ allows adaptive overlap of communication and computation, while using Converse, an interoperability framework, allows NAMD2 to integrate modules written in different parallel paradigms. Performance results obtained on a variety of parallel machines were described, which demonstrate the speed and parallel efficiency of NAMD2.

NAMD2 has been used effectively in many scientific studies. Its design makes it feasible for a motivated programmer to add new features to it, while permitting its designers to experiment with alternate algorithms. Several such extensions made to NAMD2 were discussed, demonstrating the extensibility of the program.

NAMD2 is part of the MDScope framework, which includes visualization and interaction components. Coupling these capabilities with the high parallel efficiency of NAMD2 for current and future parallel machines, we expect MDScope to become a very useful tool for researchers. NAMD2 and the other components of MDScope are available on the web at `http://www.ks.uiuc.edu/`.

Acknowledgements

The primary developers of NAMD2 were J. Phillips, A. Shinozaki, R. Brunner, N. Krawetz, M. Bhandarkar and A. Gursoy. Jesús Izaguirre and Nick Dietz performed regression testing on NAMD2, showing that it produced correct results by comparison with NAMD 1.5. NAMD development is a service of the National Institutes of Health Resource for Macromolecular Modeling and Bioinformatics under the supervision of principal investigators L.V. Kalé, R. Skeel, and K. Schulten. The steered MD module was contributed by Sergei Izrailev, also a member of the Resource. The free energy module was contributed by David Hurwitz from the group of Jan Hermans at the University of North Carolina. The DPMTA and DPME full electrostatics packages are made available by the group of John Board at Duke University. Joshua Yelon tested object-based load balancing on the ASCI Red machine. The developers thank Dorina Kosztin, Sergei Izrailev, and Ferenc Molnar for their early adoption of NAMD2, their patience during the development process, and their contributions to the Initial Applications section of this paper. This work was supported by the National Institutes of Health (NIH PHS 5 P41 RR05969-04) and the National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827 EQ). JCP was supported by a Computational Science Graduate Fellowship from the United States Department of Energy.

References

[1] H.C. Andersen. Rattle: a 'velocity' version of the shake algorithm for molecular dynamics calculations. *J. Comput. Phys.*, 52:24–34, 1983.

[2] J. Barnes and P. Hut. A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] D. L. Beveridge and F. M. DiCapua. Free energy via molecular simulation: Applications to chemical and biological systems. *Ann. Rev. Biophys. Biophys. Chem.*, 18:431–492, 1989.

[4] Thomas C. Bishop, Robert D. Skeel, and Klaus Schulten. Difficulties with multiple time stepping and the fast multipole algorithm in molecular dynamics. *J. Comp. Chem.*, 18:1785–1791, 1997.

[5] B. R. Brooks and M. Hodoscek. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News*, page 16, July 1992.

[6] David Brown, Hervé Minoux, and Bernard Maigret. A domain decomposition parallel processing algorithm for molecular dynamics simulations of systems of arbitrary connectivity. *Computer Physics Communications*, 103:170–186, 1997.

[7] David Brown, Julian H. R. Clarke nad Motoi Okuda, and Takao Yamazaki. A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers. *Computer Physics Communications*, 83:1–13, 1994.

[8] Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 95–102, Los Alamitos, Calif., 1994. IEEE Computer Society Press.

[9] T.A. Darden, D.M. York, and L.G. Pedersen. Particle mesh Ewald. an N·log(N) method for ewald sums in large systems. *J. Chem. Phys.*, 98:10089–10092, 1993.

[10] K. Esselink and P. A. J. Hilbers. Efficient parallel implementation of molecular dynamics on a toroidal network. part ii. multi-particle potentials. *Journal of Computational Physics*, 106:108–114, 1993.

[11] Ulrich Essmann, Lalith Perera, Max L. Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh Ewald method. *J. Chem. Phys.*, 103(19):8577–8593, 1995.

[12] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology*, 1(3), August 1993.

[13] L. Greengard and V. Rokhlin. A fast algorithm for particle simulation. *J. Comp. Phys.*, 73:325–348, 1987.

[14] Helmut Grubmüller, Berthold Heymann, and Paul Tavan. Ligand binding and molecular mechanics calculation of the streptavidin-biotin rupture force. *Science*, 271:997–999, 1996.

[15] J. L. Gustafson. Reevaluating amdahl's law. *Commun. of the ACM*, 31, 5:532–533, 1988.

[16] Helmut Heller Helmut Grubmüller and Paul Tavan. Famusamm: A new algorithm for rapid evaluation of electrostatic interaction in molecular dynamics simulations. *J. Comput. Chem.*, 18:1729–1749, 1997.

[17] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.

[18] William F. Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graphics*, 14:33–38, 1996.

[19] Y.-S. Hwang, R. Das, J.H. Saltz, M. Hodoscek, and B.R. Brooks. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.

[20] Sergei Izrailev, Sergey Stepaniants, Barry Isralewitz, Dorina Kosztin, Hui Lu, Ferenc Molnar, Willy Wriggers, and Klaus Schulten. Steered molecular dynamics. In P. Deuflhard, J. Hermans, B. Leimkuhler, A. E. Mark, S. Reich, and R. D. Skeel, editors, *Computational Molecular Dynamics: Challenges, Methods, Ideas*, volume 4 of *Lecture Notes in Computational Science and Engineering*, pages 39–65. Springer-Verlag, Berlin, 1998.

[21] A. Jonas. Reconstitution of high-density lipoproteins. *Methods Enzymol.*, 128:553, 1986.

[22] L. V. Kalé and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[23] 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.

[24] 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.

[25] L.V. Kalé and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[26] Robert Brunner L. V. Kale, Milind Bhandarkar 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, pages 111–122, March 1998.

[27] Hui Lu, Barry Isralewitz, Andre Krammer, Viola Vogel, and Klaus Schulten. Unfolding of titin immunoglobulin domains by steered molecular dynamics simulation. *Biophys. J.*, 75:662–671, 1998.

[28] Brock A. Luty, Malcolm E. Davis, Ilario G. Tironi, and Wilfred F. van Gunsteren. A comparison of particle-particle, particle-mesh and Ewald methods for calculating electrostatic interactions in periodic molecular systems. *Molecular Simulation*, 14(1):11–20, 1994.

[29] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. NAMD – A parallel, object-oriented molecular dynamics program. *J. Supercomputing App.*, 10:251–268, 1996.

[30] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert Skeel, Klaus Schulten, and Richard Kufrin. MDScope – a visual computing environment for structural biology. *Comput. Phys. Commun.*, 91(1, 2 and 3):111–134, 1995.

[31] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert Skeel, Klaus Schulten, and Richard Kufrin. MDScope – A visual computing environment for structural biology. In S.N. Atluri, G. Yagawa, and T.A. Cruse, editors, *Computational Mechanics 95*, volume 1, pages 476–481, 1995.

[32] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[33] M. G. Parker, editor. *Nuclear Hormone Receptors.* Academic Press, San Diego, CA, 1991.

[34] James C. Phillips, Willy Wriggers, Zhigang Li, Ana Jonas, and Klaus Schulten. Predicting the structure of apolipoprotein A-I in reconstituted high density lipoprotein disks. *Biophys. J.*, 73:2337–2346, 1997.

[35] Daniel Picot, Patrick J. Loll, and Michael Garavito. The X-ray crystal structure of the membrane protein prostaglandin H_2 synthase-1. *Nature*, 367(20):243–249, 1994.

[36] M. R. S. Pinches, D. J. Tildesley, and W. Smith. Large scale molecular dynamics on parallel computers using the link-cell algorithm. *Molecular Simulation*, 6(1):51, 1991.

[37] Steve Plimpton and Bruce Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. *J. Comput. Chem.*, 17(3):326, 1996.

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

[39] E. L. Pollock and Jim Glosli. Comments on P^3M, FMM, and the Ewald method for large periodic Coulombic systems. *Comput. Phys. Commun.*, 95(2–3):93–110, 1996.

[40] W. Rankin and J. Board. A portable distributed implementation of the parallel multipole tree algorithm. *IEEE Symposium on High Performance Distributed Computing*, 1995. In press. [Duke University Technical Report 95-002].

[41] J. P. Renaud, N. Rochel, M. Ruff, V. Vivat, P. Chambon, H. Gronemeyer, and D. Moras. Crystal structure of the RAR-γ ligand–binding domain bound to all–*trans* retinoic acid. *Nature*, 378:681–689, 1995.

[42] Jean-Paul Ryckaert, Giovanni Ciccotti, and Herman J. C. Berendsen. Numerical integration of the Cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *J. Comp. Phys.*, 23:327–341, 1977.

[43] M. Mitchell Smith. Histone structure and function. *Curr. Opinion Cell Biol.*, 3:429–437, 1991.

[44] W.L. Smith and D.L. DeWitt. Prostaglandin endoperoxide H synthases-1 and -2. *Adv. Immunol.*, 62:167–215, 1996.

[45] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[46] 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.

[47] M. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *J. Chem. Phys.*, 97(3):1990–2001, 1992.

[48] L. Verlet. Computer 'experiments' on classical fluids: I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159:98–103, 1967.

[49] J. Vincent and K. M. Merz. A highly protable parallel implementation of amber using the message passsing interface standard. *Journal of Computational Chemistry*, 11:1420–1427, 1995.

[50] Masakatsu Watanabe and Martin Karplus. Simulation of macromolecules by multiple-time-step methods. *J. Phys. Chem.*, 99(15):5680–5697, 1995.

[51] A. Windemuth. Advanced algorithms for molecular dynamics simulation: The program PMD. In T. G. Mattson, editor, *Parallel computing in computational chemistry*, pages 151–169. ACS Books, Washington, DC, 1995.

PE 0 Patch (0,0)	PE 1 Patch (0,1)	PE 2 Patch (0,2)
PE 3 Patch (1,0)	PE 4 Patch (1,1)	PE 5 Patch (1,2)
PE 6 Patch (2,0)	PE 7 Patch (2,1)	PE 8 Patch (2,2)

Figure 1: Neighboring patches in the two-dimensional downstream patch scheme.

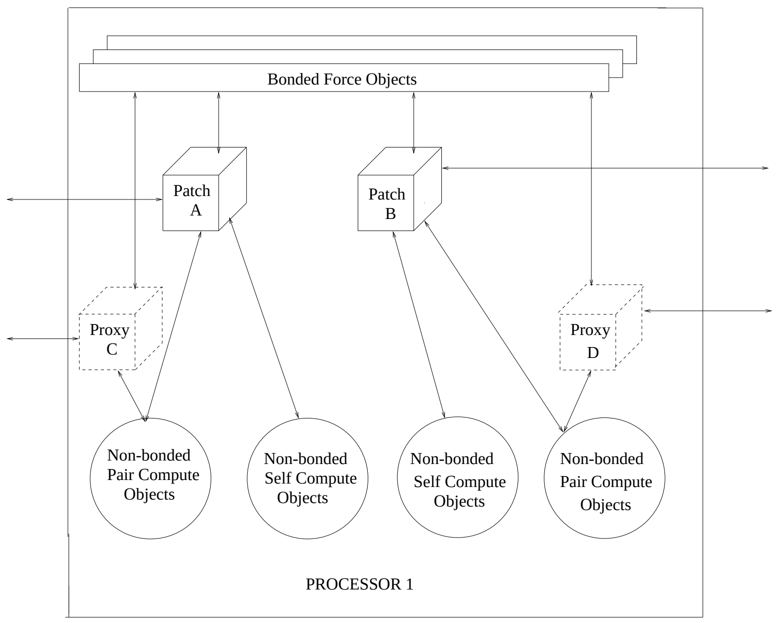
Bonded Force Objects
Patch A
Patch B
Proxy C
Proxy D
Non-bonded Pair Compute Objects
Non-bonded Self Compute Objects
Non-bonded Self Compute Objects
Non-bonded Pair Compute Objects
PROCESSOR 1

Figure 2: In NAMD2 forces are calculated by independent compute objects which depend on one or more patches for atomic coordinates. Proxy patches (shown as dotted cubes) are used to reduce the communication cost. Communication between the different entities are representd by arrows.

Method	Communication cost per node	Theoretical scalability	Comment	Programs
RD	O($N \log P$)	Not scalable	Easy to parallelize	CHARMM [5] AMBER [49] UHGromos X-PLOR
AD	O(N)	Not scalable		EGO (Early version)
FD	O($N/\sqrt{P}$)	Not scalable	Tolerable communication costs	LAMMPS [37] CHARMM [19]
SD	$(N/P)^{2/3}$	isoefficiently scalable	Possible load imbalance	ddgmq [6] SIGMA
QSD	$O(N/P)$	isoefficiently scalable	Possible load imbalance	NAMD PMD [51] EulerGromos [8]
QSD+FD	$O(N/P)$	isoefficiently scalable	Load balance feasible	NAMD2

Table I: Scalability of decomposition strategies with examples of programs that use them. SD is spatial decomposition with exactly one domain per processor. QSD stands for *quantized* spatial decomposition, where the size of boxes is fixed, independent of the number of processors.

Simulation # of atoms		Processors								
		1	2	4	8	16	32	64	128	160
bR 3,762 atoms	Time	1.138	0.578	0.315	0.158	0.086	0.048			
	Speedup	1.0	1.97	3.61	7.20	13.2	23.7			
ER-ERE 36,573 atoms	Time		6.115	3.099	1.598	0.810	0.397	0.212	0.123	0.098
	Speedup		(1.97)	3.89	7.54	14.9	30.3	56.8	97.9	123
ApoA-I 92,224 atoms	Time			10.760	5.464	2.850	1.470	0.729	0.382	0.321
	Speedup			(3.88)	7.64	14.7	28.4	57.3	109	130

Table II: Execution time (seconds) per timestep for several simulations on the CRAY T3E (450MHz processors). All the simulations were run using a 12Å cutoff. Some simulations could not run on small numbers of processors due to lack of memory, so speed-up numbers in parantheses are estimates.

		Processors									
		1	2	4	8	16	32	64	128	160	192
T3E 450MHz	Time		6.12	3.10	1.60	0.810	0.397	0.212	0.123	0.098	
	Speedup		(1.97)	3.89	7.54	14.9	30.3	56.8	97.9	123	
Origin2000 195MHz	Time	8.28	4.20	2.17	1.07	0.542	0.271	0.152			
	Speedup	1.0	1.96	3.80	7.74	15.3	30.5	54.3			
ASCI-RED 200MHz PPro	Time	28.0	13.9	7.24	3.76	1.91	1.01	0.500	0.279	0.227	0.196
	Speedup	1.0	2.01	3.87	7.45	14.7	27.9	56.0	100	123	143
HP735/125 125MHz	Time	24.1	12.4	6.39	3.69						
	Speedup	1.0	1.94	3.77	6.54						
LINUX 400MHz PII	Time	12.34	6.42	3.28	1.69	1.01	0.62				
	Speedup	1.0	1.92	3.77	7.30	12.21	19.90				

Table III: Execution time (seconds) per timestep for ER-ERE (36,573 atoms, 12Å cutoff) on several parallel machines. The HP735 workstations are interconnected with a 100Mb/s ATM switch. The Linux machines use 100Mb/s switched Ethernet.

Program	X-PLOR	NAMD 1.5	NAMD2
Time	8.1	8.8	6.1

Table IV: Execution time (seconds) per timestep for several simulation programs on a single HP735/125 workstation. The simulation is a 14,532 atom simulation of a protein in water with an 8.5Å cutoff.

```
{
    runComputeObjects();
    addForceToMomentum(0.);
    submitReductions(step);
    submitCollections(step);
    rescaleVelocities(step);
    berendsenPressure(step);
    langevinVelocities(step);

    for ( ++step; step <= numberOfSteps; ++step )
    {
        addForceToMomentum(0.5*timestep);
        if (doNonbonded)
                addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
                addForceToMomentum(0.5*slowstep,Results::slow);

        addVelocityToPosition(timestep);

        runComputeObjects(!(step%stepsPerCycle));

        addForceToMomentum(0.5*timestep);

        if (doNonbonded)
                addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
                addForceToMomentum(0.5*slowstep,Results::slow);

        submitReductions(step);
        submitCollections(step);
        rescaleVelocities(step);
        berendsenPressure(step);
        langevinVelocities(step);
        rebalanceLoad(step);
    }

    terminate();
}
```

Figure 3: Sequencer algorithm for a dual-timestep long-range non-bonded force simulation

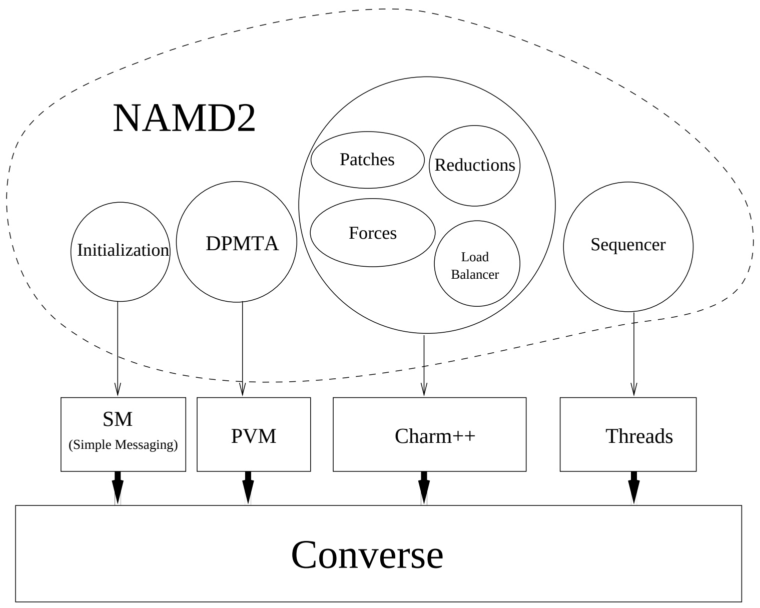

Figure 4: NAMD Architecture

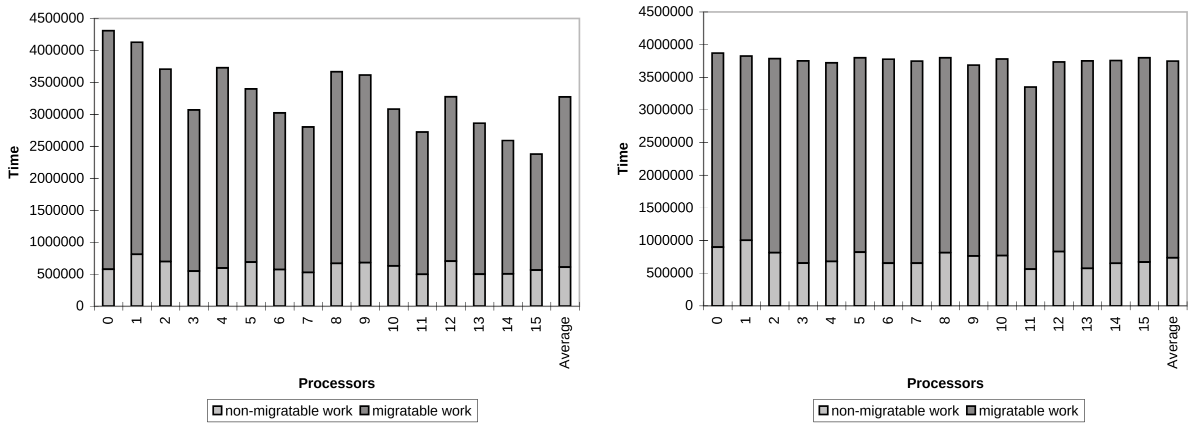

Figure 5: Load distribution for a 16 processor simulation, showing the load before (left) and after (right) running the load balancer.

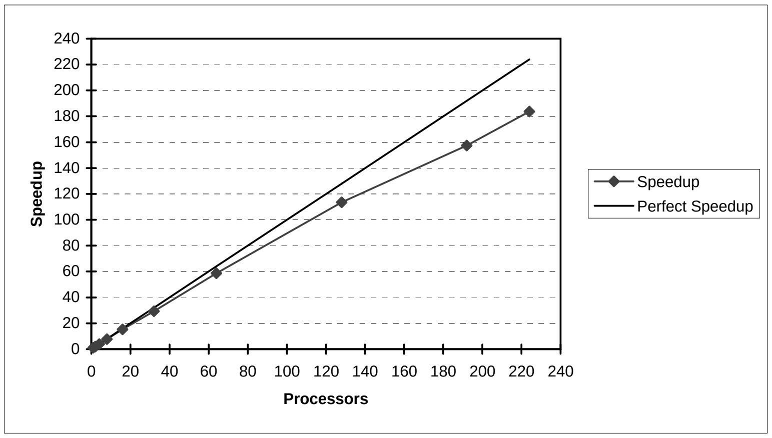

Figure 6: The speed-up for ApoA-I (92,224 atoms, 12Å cutoff) on the ASCI-RED.