ACHIEVING SCALABLE PARALLEL MOLECULAR DYNAMICS USING DYNAMIC SPATIAL DOMAIN DECOMPOSITION TECHNIQUES

LARS NYLAND, JAN PRINS, RU HUAI YUN, JAN HERMANS, HYE-CHUNG KUM, AND LEI WANG

ABSTRACT. To achieve scalable parallel performance in Molecular Dynamics Simulations, we have modeled and implemented several dynamic spatial domain decomposition algorithms. The modeling is based upon the Bulk Synchronous Parallel architecture model (BSP), which describes supersteps of computation, communication, and synchronization. Using this model, we have developed prototypes that explore the differing costs of several spatial decomposition algorithms, and then use this data to drive implementation of our Molecular Dynamics simulator, *Sigma*.

The parallel implementation is not bound to the limitations of the BSP model, allowing us to extend the spatial decomposition algorithm. For an initial decomposition, we use one of the successful decomposition strategies from the BSP study, and then subsequently use performance data to adjust the decomposition, dynamically improving the load balance. The motivating reason to use historical performance data is that the computation to predict a better decomposition increases in cost with the quality of prediction, while the measurement of past work often has hardware support, requiring only a slight amount of work to modify the decomposition for future simulation steps.

In this paper, we present our adaptive spatial decomposition algorithms, the results of modeling them with the BSP, the enhanced spatial decomposition algorithm, and its performance results on computers available locally and at the national supercomputer centers.

Keywords: Spatial decomposition, Adaptive load balancing, BSP cost modeling, Molecular dynamics, Parallel algorithms

1. INTRODUCTION

A driving goal of our research group is to develop a high performance Molecular Dynamics simulator to support biochemists in their research. Our goals are to study large timescale behavior of molecules and to facilitate *interactive* simulations where the scientist can control the simulation [9]. Two main characteristics of the problem impede our goal: first is the large number of interactions in realistic systems (solvated biomolecules), and second is the extremely small simulated timestep that is required to adequately capture the high frequency motions. Algorithmic improvements and approximation techniques have been used to successfully improve performance, some of which will be mentioned here. However, the primary focus of this paper is the parallelization of Molecular Dynamics simulations that achieves scalable performance. To develop parallelization strategies that meet our goals, we model the problem at a high-level where compelling results are developed, which are then used as a basis for an implementation. In the implementation, additional optimizations have been introduced, requiring further modification of the parallelization strategies, resulting in a scalable parallel implementation.

1.1. **Molecular Dynamics (MD) Simulation.**

Molecular dynamics simulators use classical models of motion and interaction, rather than the more modern and complex models of

This work has been supported in part by the National Institutes of Health's National Center for Research Resources (grant RR08102 to the UNC/Duke/NYU Computational Structural Biology Resource).

quantum mechanics, to compute and apply forces. Problems such as docking a ligand in a receptor, performing structure refinement or performing sequence analysis are among the many problems that can be explored with MD simulation.

In the simulation, a continuous process is broken down into discrete timesteps, chosen small enough that the discretization effects are minimized. At each step, the sum of all forces on each atom is calculated, and then applied with regard to the duration of the timestep, updating the position and velocity values. The forces originate from bonded and non-bonded forces between the atoms. The bonded forces seek to maintain bond lengths, bond angles, and dihedral angles on single bonds, two-bond chains and three-bond chains, respectively. The non-bonded forces are comprised of the electrostatic forces from electrical charges and the Lennard-Jones forces. The non-bonded forces are symmetric (equal and opposite), occur between each pair of atoms, and vary as an inverse power of the distance between the atoms involved.

One other aspect of MD that requires explanation is the handling of boundaries. There are two choices, open boundary conditions and periodic boundary conditions. The open boundary condition is a simulation of the molecular systems as if it is in a vacuum. In periodic boundary conditions (PBC), the system of atoms is typically box-shaped, and is conceptually repeated to fill space. Thus any atoms drifting out of the simulation space reenter the space on the opposite side with the same velocity and direction. Forces are also calculated using periodic space. In the simulations described here, periodic boundary conditions are used.

By far, the most time-consuming step of the simulation is the computation of the non-bonded forces. A single atom has bond-related forces with a limited number of other atoms, but the non-bonded forces exist between all pairs of atoms, yielding $O(n^2)$ interactions. It is in this part of the MD simulation where approximations are sought and accuracy is given up, all in an effort to improve performance. One solution to reduce the calculation introduces a cutoff radius, where non-bonded forces are calculated each simulation step only when the distance between a pair of atoms is less than some preset radius, R_c. The remaining longer-range forces are either calculated by some other method (the $O(n)$ Fast Multipole Algorithm [2], or the $O(n \log n)$ Particle-Mesh Ewald method [5]), calculated less frequently, or completely ignored.

Of course, the choice of timestep duration is another approximation, where longer timesteps yield faster calculations with less accurate results. This is an integration process of a non-linear system, thus the timesteps must be small enough to account for the most rapid changes that are likely to occur. Some work has been done to find algorithmic solutions that allow longer timesteps without reducing accuracy. Of primary importance to our group are the "shake" algorithm [12], and Langevin Dynamics [13].

1.2. **Parallel Execution.** Parallelization of MD is another method that improves performance with the benefit that accuracy is maintained. Good opportunities for parallelization exist; all of the forces on each of the atoms are independent, so they can be computed in parallel. Once computed, the non-bonded forces are summed and applied; again, these are operations where parallelism can benefit the performance.

The work involved in computing the non-bonded forces is perfectly characterized by the *pairlist* data structure. Each entry in the pairlist, (i, j), indicates that the non-bonded force between atoms (or atom groups) numbered i and j needs to be computed, where a pair (i, j) is added to the list if the distance between the atoms is less than a preset cutoff radius, R_c. This is just one of many approximations (or optimizations) that must be considered in parallelizing an MD simulator.

Parallelizing the non-bonded computation by splitting the pairlist into P equal parts divides the computational work perfectly, as every interaction requires the same amount of work. Accessing the atom data, however, has no particular coherence, so a processor working on its portion of the pairlist will be accessing data scattered throughout the list of atoms. Thus, for this decomposition to perform well, each processor must have uniform access to all memory, a feature that does not exist on most parallel computers. This decomposition has been used by us, but it scales poorly, since communication increases with P (there tends to be communication between all pairs of processors).

There is coherence in the pairlist that can be exploited to minimize communication. Two nearby atoms both interact with all atoms that are within R_c of both. Thus, a parallel decomposition based on spatial location (usually into box-shaped regions) provides two opportunities for reduced communication improving parallelization. First, the atoms placed on each processor are near each other, thus accessing the data for many nearby atoms data will not require communication. Second, for those interactions that require data from neighboring regions, atomic data can be fetched once and then reused many times, due to the similarities of interactions of nearby atoms.

The communication costs of a spatial decomposition tend to decrease with increasing P, since the surface area of each box decreases as P increases. As long as the dimensions of the boxes are larger than R_c, only neighboring regions share data. Once the dimensions become smaller than R_c, the number of regions that communicate increases, however the total communication volume per processor still decreases as P increases. Thus, latency and bandwidth costs of accessing remote values both play an important role as P becomes large.

The atom positions change during the simulation (leading to density changes), thus there is a need to update the pairlist. Since the atoms do not move very far during a single step of the simulation and the recalculation of the pairlist is not trivial, the same pairlist is used for 10–50 simulation steps. Parallelizing such a dynamic simulation requires that the decomposition be adjusted periodically to ensure the equitable distribution of work.

The use of a spatial decomposition for MD has become widespread in recent years. It is used by AMBER [3, 11], Charmm [7], Gromos [4] and NAMD [10], all of which run in a message-passing paradigm, as opposed to our shared-memory implementation. In general, each of these implementations found good scaling properties, but it is difficult to compare overall performance of the parallel MD simulators, as machine speeds have improved significantly since publication of the cited reports.

1.3. **Modeling MD.** To understand and evaluate the complexities of parallelizing MD simulations, we rely on a high-level model of parallel computing, the Bulk Synchronous Parallel (BSP) architecture, to help us evaluate different decomposition strategies for their parallel efficiency. The model not only reveals characteristics of the parallel algorithm, it also helps in regard to making choices based upon the capabilities of the parallel hardware, for instance, weighing the cost of recomputation against sending the same result between processors. In the next section, we describe an exercise in modeling molecular dynamics computation exploring several decomposition strategies, showing results with regard to parallel (in)efficiencies, communications costs, and synchronization costs.

1.4. **Parallel Implementation.** Our work continues with the parallelization of an MD simulator that was developed and is in use by biochemists in our group. We rely on the results of the BSP modeling, but must develop the parallelization further, since the model does not completely describe the computation. In section 4, we describe the complexities and a decomposition strategy that adapts over the span of the simulation that efficiently and effectively rebalances the work while keeping communication costs at an insignificant level. To show the effectiveness, performance results are shown and discussed.

We conclude with guidelines for building efficient parallel MD simulators and present our goals for future work.

2. Modeling Parallel Computation with the BSP Model

The Bulk Synchronous Parallel (BSP) model has been proposed by Valiant [14] as a model for general-purpose parallel computation. It was further modified in [1] to provide a *normalized BSP cost* of parallel algorithms, providing a more uniform comparison of algorithms. The BSP model is both simple enough to quickly understand and use, but realistic enough to achieve meaningful results for a variety of parallel hardware systems. It also aids in the choice of tradeoffs (computation vs. communication) when parameterized by hardware capabilities.

A parallel computer that is consistent with bulk synchronous parallel architecture has a set of processor-memory pairs, a communication network that delivers messages in a point-to-point manner, and a mechanism for efficient barrier synchronization of the processors. There are no broadcast or combining features specified. Parallel hardware can be parameterized with 4 values:

1. The number of processors, P.
2. The processor speed, s, measured in floating-point operations per second.
3. The latency, L, which reflects the minimum latency to send a packet through the network, which also defines the minimum time to perform global synchronization.
4. The gap g, reflecting the network communication bandwidth on a per-processor basis, measured in floating-point operation cycles taken per floating-point value sent.

Thus, the BSP model is adequate for modeling any parallel machine that can be characterized by a 2-level memory hierarchy such as message-passing machines and shared-memory machines (e.g. cache vs. shared-bus memory).

An algorithm for the BSP is written in terms of S supersteps, where a single superstep consists of some local computation, external communication, and global synchronization. The values communicated are not available for use until after the global synchronization. The cost of a single superstep is:

$$C_i = w_i + gh_i + L$$

where w_i is the maximum number of local computation steps executed by any processor during the superstep, h_i is the maximum number of values sent or received by any processor during the superstep, and L is the cost of synchronizing all processors. The total cost of executing a program of S steps is:

$$C_{tot} = \sum_{i=1}^{S} C_i = W + Hg + SL$$

where

$$W = \sum_{i=1}^{S} w_i \text{ and } H = \sum_{i=1}^{S} h_i$$

The normalized cost is the ratio between the BSP cost using P processors and the optimal work (perfectly) distributed over P processors. The optimal work W_{opt} is defined by an optimal (or best known) sequential algorithm. The normalized time is expressed as

$$C(P) = \frac{C_{tot}}{W_{opt}/P}$$

The normalized cost can be reformulated as

$$C(P) = a + bg + cL$$

where

$$a = W/(W_{opt}/P), b = H/(W_{opt}/P), \text{ and } c = S/(W_{opt}/P)$$

When the triplet $(a, b, c) = (1, 0, 0)$, the parallelization is optimal. Values where $a > 1$ indicate extra work is introduced in the parallelization and/or load imbalance among the processors. Values of $b > 1/g$ or $c > 1/L$ indicate that the algorithm is communication bound, for the architecture described by particular values of g and L.

Using this model, we can develop parallel algorithms, implement them enough to acquire performance metrics (operation counts, message counts, and step counts) to compute normalized execution costs. Additionally, we can characterize typical parallel computers by assigning values for P, g, and L, and compute values for the normalized cost of particular algorithms (assigning values to s would give timing information as well). We demonstrate the process in the next section exploring a variety of decompositions for MD.

3. Modeling Parallel MD domain decompositions using a BSP model

In this section we describe a modeling procedure that helped us evaluate several domain decomposition strategies for Molecular Dynamics computations. We describe the simplified algorithm, the domain decomposition techniques, and show the results of the the model.

3.1. **A Simplified MD Algorithm.** The most time-consuming step of MD simulations is the calculation of the non-bonded forces, typically exceeding 90% of the execution time. Thus we limit our modeling study to this single aspect. Our simplified algorithm for computing the non-bonded forces is shown in figure 1. It consists of an outer loop that updates the pairlist every k steps, with an inner loop to perform the force computations and application. Computing the pairlist is expensive and changes very little between each simulation step. Thus, the value k ranges from 1 to 50 steps, and is often referred to as the *cycle* or pairlist calculation frequency. This is yet another approximation that helps improve MD performance, and can be used since the atoms do not move too far between pairlist recalculations.

In our modeling of MD, we examine the cost of executing a single cycle. This amortizes the cost of pairlist calculation over the k interaction steps. Computing the cost of one cycle is adequate as the cost of subsequent cycles is roughly the same, so no new information is gained by modeling more than one cycle.

The target parallel architectures of our group are primarily shared-memory computers, such as those currently manufactured by SGI, Convex and Intel, thus the meaning of remote access to us is a hardware fetch from shared memory into either local memory (if it exists) or into a processor's cache. While this represent a bias of our group (and therefore the choices in machine parameters), using the BSP model applies equally well (some might say better) to message-passing computers such as the IBM SP-2.

```
for t = 1 to T by k {
  if Processor == 0 then
     distribute atoms to processors    // decomposition
  calculate local pairlist
  for s = t to t+k - 1 {
    get remote atom information        // communication
    synchronize
    calculate forces on local atoms    // computation
    apply forces to update local positions/velocities
  }
}
```

FIGURE 1. A high-level algorithm for performing parallel molecular dynamics computations.

3.2. **The Outer Loop.** The outer loop (re)distributes the atoms to processors for each cycle. We modeled the implementation with three supersteps, in the following sequence:

1. Computation (P_0 only): Calculate the decomposition. Communication: Send the atoms to each assigned processor. Synchronize.
2. Computation: Sort local atoms by position, build communication lists. Communication: send atomic data of *perimeter atoms* to adjacent processors. Synchronize.
3. Computation: build local pairlist. Synchronize.

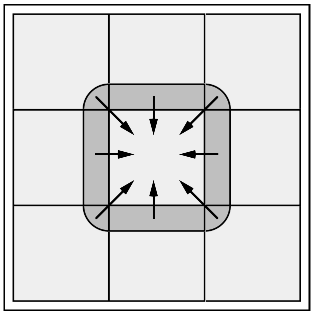

FIGURE 2. This figure shows the perimeter regions for each processor that pertain to the central region in this spatial decomposition. The atoms contained in the regions within the cutoff radius, R_c, of the edge must be sent to the central region, as indicated by the arrows.

Figure 2 shows a spatial decomposition that indicates the perimeter atoms that must be sent to a single processor.

3.2.1. *Decomposition Calculation.* The calculation of the decomposition varies substantially depending on the decomposition used. The decompositions generally rely on decomposing the space in half along the largest dimension recursively, until the number of subdomains equals the number of processors. Each of the decompositions is described in detail in section 3.4.1.

3.2.2. *Building Communications Lists.* Positional information for atoms that are within R_c of any boundary is sent to the processors that share the boundaries. These atoms are referred to as *perimeter atoms* and are the sole source of interprocessor communication. During this step, each processor further subdivides the atoms into bins that are close to the cutoff radius, R_c, on a side. This placement of atoms in bins makes the calculation of communication lists relatively simple and dramatically reduces the pairlist calculation. All of the regions in the spatial decomposition have neighbors on all sides, due to the nature of using periodic boundary conditions.

3.2.3. *Building Local Pairlists.* Once all the perimeter data has been accumulated, each processor can calculate which pairs of atoms interact. One choice made in this study has to do with the symmetric nature of Newton's third law, where $F_{ij} = -F_{ji}$. In the algorithm described here, a processor can only update its own data, so we reuse the data from the force calculation where atoms i and j are local, but duplicate the calculation when atom j is a perimeter atom (atom i is always a local atom). This is not necessarily the way forces must be calculated, but keeps the BSP MD algorithm simpler by having only one superstep per inner-loop iteration. It has the negative effect that the amount of total work approaches twice the sequential version as P increases, since each processor has fewer atoms (given the same input data) and there is less reuse of an expensive calculation.

3.3. **The Inner Loop.** There is only one superstep in the inner loop. It consists of distributing positions of perimeter atoms to nearby processors; a synchronization barrier to ensure all computation is using data from the same iteration; followed by a force calculation and application. The computations performed by the inner loop are the same for all decompositions.

3.4. **Modeling Experiment.** The goal of the experiment is to model four data decomposition strategies using nine actual molecular data sets with varying numbers of processors to find values of a, b, and c for each combination. Normalized execution costs can be computed by choosing values of g and L (to represent available parallel computers). The values of a, b, and c show how work and communication affect parallel performance, and are computed using

$$a = (w_{outer} + k \cdot w_{inner})/(W_{opt}/P)$$
$$b = (h_{outer} + k \cdot h_{inner})/(W_{opt}/P)$$
$$c = (S_{outer} + k \cdot S_{inner})/(W_{opt}/P)$$

to compute

$$C(P) = a + bg + cL$$

Total costs show how the different decomposition strategies scale.

3.4.1. *Decompositions Explored.* The four decomposition strategies in this study are:

- **Uniform Geometric Decomposition.** This decomposition is the most intuitive and easiest to compute. The decomposition simply splits the simulation space (or subspace) equally in half along each dimension until the number of subspaces equals the number of processors (implying that $P = 2^p$ for some integer p).
- **Orthogonal Recursive Bisection Decomposition (ORB).** This method is based upon the orthogonal recursive bisection decompositions used in some N-body simulators [15], but is extended to partition space in addition to work. ORB is used here to recursively split the longest dimension by placing a planar boundary such that half the

Decomposition Model Parameters	
P	the number of processors
N	the total number of atoms
k	the number of interaction steps per cycle
pn_i	the number of atoms on processor i
B_i	the total number of bins on processor i
$AtomNeighbor_{Bin_{ij}}$	the number of atoms in neighbor bins for bin i on processor j.
PL_i	the total length of the pairlist for all atoms on processor i
$K_{comm} = 8$	the number of values sent per atom (position, velocity, charge, type)
$W_{comp} = 60$	the work performed in computing the force between 2 atoms
$R_c = 10$ Angstroms	the cutoff radius that bounds interactions

FIGURE 3. The parameter values used to determine the cost of execution using a BSP model with various spatial decompositions.

atoms are on one side, and half are on the other. This yields an assignment of atoms to processors that varies by at most 1.

- **Pairlist Decomposition.** The non-bonded work in MD simulators is directly proportional to the number of entries in the pairlist. Thus a decomposition that yields perfect load-balancing evenly decomposes the pairlist among processors, which is how this decomposition is defined. A potential drawback is that it does not have spatial locality, thus we have included it to better understand this aspect.
- **Spatial Pairlist Decomposition.** To remedy what might be a locality problem with the pairlist decomposition, we also consider a spatial decomposition that is based upon the number of entries in the pairlist assigned to each processor, placing spatial boundaries (using ORB) based on pairlist length rather than atom count.

3.5. **Model Parameters.** To evaluate the cost of a decomposition, certain parametric values must be determined to evaluate a cost equation. The parameters used in the model are shown in figure 3.

3.6. **Optimal Cost.** The optimal cost is needed as a basis for comparison. The times for the outer and inner loops are expressed as:

$$\begin{aligned} W_{opt} = \; & \text{the work to compute the pairlist} + \\ & k \cdot (\text{interaction cost over all pairs} + \text{integration cost}) \end{aligned}$$

This includes the force symmetry optimization that if the pair (i, j) is in the pairlist, the pair (j, i) is not.

3.7. **Inner Loop Cost for All Parallel Decompositions.** The cost of the inner loop can be calculated with the same formula for all decompositions.
The work is expressed as:

$$w_{inner} = \left[\max_{i=1..P}(PL_i) + \max_{i=1..P}(pn_i)\right] \times W_{comp}$$

The data transmission time is:

$$h_{inner} = \max_{j=1..P}\left(\sum_{i=1}^{B_j} AtomNeighbor_{Bin_{ij}}\right) \times K_{comm}$$

And the latency incurred is:

$$S_{inner} = 2$$

3.8. **Spatial Decomposition Costs.** The cost of executing the outer loop of each decomposition can be determined in one of two ways. Certainly, it can be determined by cost equations similar to those above, or the cost can be counted by measuring programs that perform the decomposition. We follow the latter scenario, counting operations performed in the decomposition. What follows is a description of how each of the spatial decomposition strategies works.

3.8.1. *Uniform Geometric Decomposition Cost.* This decomposition recursively halves space in the longest dimension in $\log P$ steps yielding P subspaces. The cost of the outer loop is broken down into the following components:

1. Decompose data, using P_0 only.
2. Send approximately $N - N/P$ atoms to other processors, according to step 1.
3. Build bins and place all atoms in bins.
4. Find the initial communications list. On each processor, all bins near the surface of the subregion contain atoms that must be sent to other processors.
5. Send the atoms in communication list to the other processors.
6. Calculate the pairlist. Each processor examines all of its local atoms against all local and perimeter data to compute the local pairlist (using bins).

3.8.2. *Orthogonal Recursive Bisection Cost.* This decomposition recursively halves each subspace based on atom-count rather than dimension (still along the longest dimension, to maintain as cubic a shape as possible). While this appears to require sorting of atoms by position in the dimension being split, an $O(N)$ algorithm for partitioning the data into two halves can be employed for higher performance. The partitioning operation is the only difference from the uniform geometric decomposition.

3.8.3. *Non-spatial Pairlist Cost.* The goal of the non-spatial pairlist decomposition is to split the pairlist into P equal segments, assigning one segment to each processor. The pairlist is ordered by atom number, therefore nothing is known about the spatial locality of the decomposition (except that atoms are typically numbered sequentially along the protein backbone, so there is some locality between sequentially numbered atoms). Additionally, as in other decompositions, a single atom's entire pairlist is assigned to a single processor to avoid an extra collection step at the end of each inner loop iteration.

To compute the pairlist in parallel, a tentative assignment of atoms to processors is made using the uniform geometric decomposition, and then a rebalancing step takes place to redistribute the pairlist entries for each processor. When calculating the pairlist, we include both interactions for a single pair ((i, j) and (j, i)), since the locality optimization cannot be predicted.

The non-spatial pairlist decomposition follows the steps 1–5 outlined in section 3.8.1, and then executes the following:

6. Calculate the full pairlist ($n^2 - n$ entries) using bins as before.
7. Send length of pairlist for each atom i to P_0.
8. Decompose based on pairlist, splitting the pairlist into P nearly-equal segments, keeping each atom's entire pairlist intact (assigned to a single processor).
9. Assign the atoms to processors according to pairlist decomposition in step 8.
10. Calculate communication list.

Molecular Input Data	
Name	Number of atoms
Alanin	66
Dipeptide (wet)	231
SS Corin	439
Water	798
Argon	1728
SS Corin (wet)	3913
Eglin	7065
Water	8640
Polio (segment)	49144

FIGURE 4. The input dataset names and number of atoms used for measuring different decompositions

11. Send the atoms, their pairlists, and communication list to other processors according to step 9 and 10.
12. Purge the local pairlist to utilize Newton's 3rd law when both atoms are local.

3.8.4. *Spatial Pairlist Decomposition Cost.* The spatial pairlist decomposition follows steps 1–7 in section 3.8.3. After those are performed, the following additional steps are performed.

8. Decompose based on ORB and pairlist, by using location to order the atoms and the pairlist length as the metric for placing boundaries. We still keep each atom's entire pairlist intact.
9. Send atoms to other processors, according to step 8.
10. Repeat steps 3–6 in section 3.8.1.

This decomposition is similar to the non-spatial pairlist, except in step 8, the second decomposition is based the ORB decomposition, using location to order the atoms and the pairlist length as the metric for placing boundaries. In other words, instead of using the atom number to decompose the pairlist, we recursively split the longest dimension by placing an orthogonal planar boundary such that the pairlist is (nearly) evenly divided. We then eliminate all entries where $j > i$ and atoms i and j are local. For the data sets in our study, this elimination step produced nearly equal reduction in each processor's pairlist.

3.9. **Experimental Results.** Using the estimates of computation, communication and synchronization costs developed in the previous section, along with several prototype programs that measured the costs of the various decompositions, normalized coefficients were computed for all decompositions with all data sets where the cycle cost (k) was set to 50 (available in [8]).

3.9.1. *The Input Data.* Rather than generate atomic position data according to some probability distribution function, we chose nine input files for MD simulations. Our goal was to explore the decompositions over a wide range of atom populations, so that we could determine if any of the decomposition strategies were sensitive to data size. From each input file, we used the positional data only, ignoring all other information. The table in figure 4 summarizes the input data sets.

FIGURE 5. A comparison showing the magnitude of difference between a and b, and a and c for all (a, b, c) triplets in all decompositions in our experiment. The values of a are plotted along the x-axis and are in the range of 1–7. The b values are plotted with solid markers against a, where the normalized b values vary from 10^{-4} to 0.2. Similarly, the c values are plotted along the y-axis with hollow markers against a, where the c values fall in the range of 10^{-8} to 10^{-3}. The difference of several orders of magnitude indicates that load-balancing and parallel overhead have far more effect on parallel inefficiency than communication bandwidth and latency and should be the focus of reduction to achieve efficient parallel MD simulators.

A general conclusion is that the parallel overhead plus load-imbalance (amount that $a > 1$) far outweighs the cost of communication and synchronization on virtually all of the results in the study. Any decomposition that seeks a more evenly balanced load (reduction of a) will improve performance far more than solutions that seek reduced communication (lower b) or reduced synchronization (lower c). Recall that the normalized cost is $C(P) = a + bg + cL$, where values for g fall in the range of 1 – 100, and L values are typically 25 – 10000 for most parallel machines, so it is clear that communication and synchronization add to the cost only when g and L are very large. Thus, even parallel computers with the slowest communications hardware will execute well-balanced, spatially-decomposed MD simulations with good parallelism.

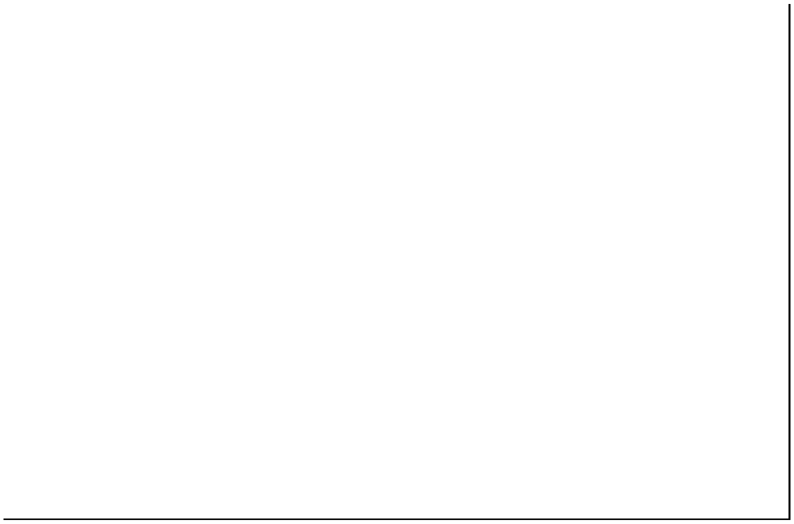

FIGURE 6. This graph shows the normalized cost for different decompositions as the number of processors is increased (1 is perfect parallelization). The parametric values of (g, L) are $(8, 25)$, representing a typical NUMA computer.

The data in figure 6 shows the effect of using additional processors on two of the larger data sets (Eglin and Polio) on a NUMA-class machine $((g, L) = (8, 25))$ over all of the defined decomposition strategies. Ideally, the normalized cost would be flat, with a value of 1, indicating perfect parallelization. But, as can be seen, all decompositions lead to increases in normalized cost, where it is clear that ORB and spatial pairlist decompositions perform better than uniform and non-spatial pairlist decompositions. The increase of the normalized cost on the best decompositions is reflected in large part by the diminishing reuse of $F_{ij} = -F_{ji}$. Thus with larger and larger values of P, a normalized cost of 2 will be the minimum that can expected.

The graph in figure 7 shows the effect of communication speed on the overall performance of the different decompositions. In this dataset, P is set to 32, and two different machine classes are examined. The first is a uniform memory access machine (UMA), with $(g, L) = (1, 128)$, representing machines such as the Cray vector processors that can supply values to processors at processor speed once an initial latency has been charged. The second is a non-uniform memory access machine (NUMA), much like the SGI parallel computers, the Convex SPP, and the now defunct KSR-1. The parameters are $(g, L) = (8, 25)$, indicating that it takes the same amount of time to perform 8 floating-point operations as it does to transmit one value.

There are two interesting conclusions to be drawn from figure 7. The first is that executing MD on a machine with extremely high communication bandwidth (UMA) performs, in normalized terms, almost identically with machines with moderate communications bandwidth. This is seen in the small difference between the same data using the same decomposition, where the normalized execution cost for both architectures is nearly the same.

The second interesting point in figure 7 is that decomposition matters much more than communications bandwidth. The decompositions that attempt to balance work and locality (ORB and spatial pairlist) have a bigger impact on performance than parallel computers

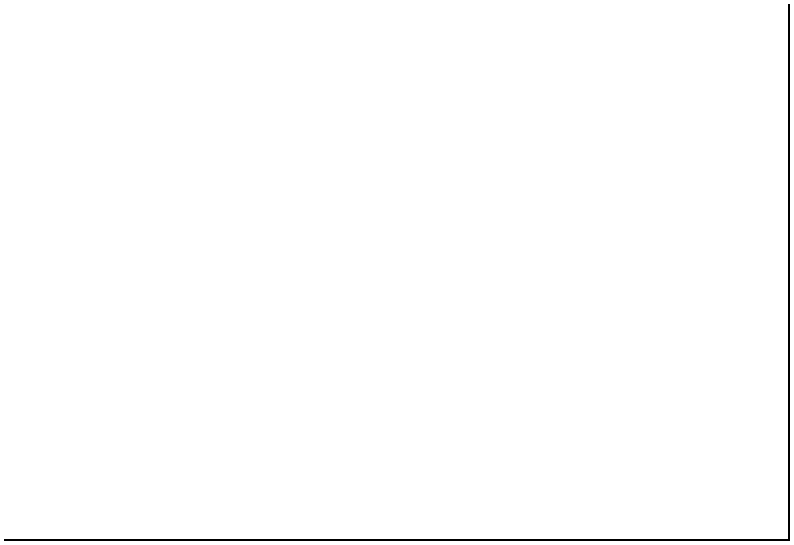

FIGURE 7. This graph shows the normalized execution cost on 32 processors, comparing different decomposition strategies on machines with differing communication performance. For the UMA architecture, $(g, L) = (1, 128)$; for NUMA, $(g, L) = (8, 25)$. Note that the normalized cost of a program on a machine with very high performance communication is only marginally better than machines with substantially lower communication performance (except for pairlist decomposition).

with extremely high performance communications. This is a good indication that either the ORB or spatial pairlist decomposition should be used for a parallel implementation on any parallel computing hardware.

3.11. **Conclusions from BSP Study.** The most significant conclusion drawn from this study is that load-balancing is by far the most important aspect of parallelizing non-bonded MD computations. This can be seen in the significantly larger values of a when compared to values of b and c, as well as the results in figure 7 that show the improvement gained in using load-balanced decompositions. The spatial decomposition using pairlist-length as a measure shows the advantage that is achieved by increasing locality over the non-spatial pairlist decomposition. Additional conclusions are that the ORB decomposition performs as well as any in load-balancing, and that adequate performance can be achieved on small numbers of ethernet connected workstations. These results are important not only in our work implementing simulators, but to others as well, guiding them in the choices of their parallel algorithms.

The amount that the values of a exceed 1 indicate parallel overhead and/or load imbalance. Our conclusions about the excess are that it is caused primarily by load imbalance, except in the non-spatial pairlist case, since it performs twice as much work. The reasons are that the serial decomposition computation is relatively inexpensive and that the sizes of the data and parallel computers generate very little redundant work, leaving load-imbalance as the main reason a exceeds 1.

4. Implementation of Dynamic Load Balancing in Molecular Dynamics

As stated in the introduction, the overall goal of our work is to provide biochemists with simulations that run at interactive speeds and produce large timescale results as quickly as possible. The results of the previous section, while providing interesting modeling results, are simply a stepping stone in pursuit of our overall goal. In this section, we describe the parallelization (using spatial decomposition for shared-memory computers) of a long-lived MD simulator, Sigma (known as Cedar in earlier versions). The performance results in this section show that the modeling provides a good starting point, but without dynamic load-balancing, there will be a loss of performance. The parallel implementations were performed during roughly the same time period as the modeling, thus not all the modeling results were available to guide the implementation. Further, there are optimizations in the MD simulator whose effects are difficult to model, leading to more interesting decomposition strategies in the actual simulator.

4.1. **Parallelization History of Sigma.** The Sigma MD simulator has been parallelized several times to achieve high performance on available hardware. Many of the early efforts rely on compiler facilities for parallelizing or vectorizing Fortran loops for machines such as the Cray Vector Processors (X-MP, Y-MP, C-90 and T-90). The Fortran versions of the non-bonded force calculations are highly optimized to mesh well with particular Fortran compilers. Fortran and C versions of several key functions are still maintained, to ensure the use of the best (or only) compiler available. These versions have no concern for locality.

A pairlist domain decomposition was developed for the MasPar MP-1, a SIMD architecture where we had access to 8192 processors. The decomposition strictly decomposed the pairlist into P equal parts (± 1). Once the forces were calculated for each entry in the pairlist, a machine-wide segmented summation took place to compute the forces on individual atoms. The parallelization of non-bonded forces had very high performance, exceeding that of the Cray Y-MP, but the parallelization of the remaining parts required substantial communication and had reduced parallelism, thus the overall performance was not that high.

With the availability of the KSR-1 shared-memory computer (where communications use a ring-of-rings), we quickly ported the pairlist decomposition code, and ran with moderate success as long as only a single ring was used (32 processors or fewer). Any attempts to use additional processors yielded very little performance improvement, due to the high communication overhead when multiple rings were used.

It was with the KSR where we first explored spatial domain decompositions, developing a version that used a uniform geometric decomposition. This decomposition works particularly well with 8 processors on most molecular systems, since most datasets usually have a protein molecule centered in a solvent bath, and the decomposition divides space in half along each dimension ($2 \times 2 \times 2$). The protein and solvent atoms are evenly divided among processors, yielding a well balanced simulation. However, any attempt to use additional processors resulted in surprisingly poor scaling.

It was hypothesized that this was due, in part, to the optimizations that had been developed for treating water molecules specially, instead of as 3 independent atoms. Water is a well-understood molecule, and there are several models recognized for modeling water without modeling each of its atoms individually (e.g. TIP3P). Thus, for any decomposition with more than 2 partitions in a single dimension, the partitions on the edge had mostly water, while the center-most partitions were predominantly full of protein atoms. We had developed an Orthogonal Recursive Bisection decomposition algorithm for Sigma (using atom count as the partitioning metric), which yielded a moderate improvement, especially when $P > 8$. Still lacking what we felt was high performance, we modified this version to take measurements, and determined that the work associated with one protein atom was about twice that of each atom in water. We used this metric to define a new ORB spatial decomposition based on estimated work, and had still more improvement over the previous decomposition.

The performance was still less than desired, especially when $P > 32$. It was apparent from timing diagrams that the load balance was far from ideal, as individual processors were idle as much as 30% of each simulation step. This could be seen by the repetitive idle gaps in each step of the simulation on a time history of each processor that showed when they were busy or idle. It was also clear from the same plots that the amount of time spent in the overhead of computing the decomposition was small, but still noticeable. We sought a strategy that was quick to compute and more evenly balanced, yielding high parallel efficiency.

4.2. **The Effect of Optimizations.** Optimizations in sequential programs often hamper parallelization, as they usually reduce work in a non-uniform manner. If the non-uniformity is not taken into consideration, the decomposed work will be unbalanced and parallelization will be less than expected.

There are (at least) two optimizations that hamper the success of an ORB decomposition in Sigma. The first is the special treatment of water, as mentioned above. Any decomposition based on atom count will have less work assigned when the percentage of atoms from water molecules is higher.

The second is the creation of *atom groups*, where between 1 and 4 related atoms are treated as a group for certain calculations. An excellent example is in the creation and storage of the pairlist, where atom groups are used (positioned at their center of mass) instead of individual atoms. This reduces the number of distance comparisons and the number of entries in the pairlist by a factor of about 9.

These optimizations and others reduce work in a non-uniform manner, which is not adequately modeled in section 3. As with any parallel algorithm, it is desirable to perform no more total work than an optimized sequential version, so a parallelism strategy must be implemented that considers the optimizations used in our simulator. We describe such an effort in the next section.

4.3. **A Dynamic Domain Decomposition Strategy.** One interesting result from our experience with the KSR was the consistency of the imbalance in the load over a long period. Typically, one processor had a heavier load than the others, and it was this processor's arrival at the synchronization point that determined the overall parallel performance. The state-vs.-time charts of execution on the KSR-1 convinced us that an adaptive decomposition was necessary, and that performance improvements could be made if the computational loads among the processors were more balanced.

Fortunately, high-performance computers often have built-in support to give detailed performance quantities about a program. For instance, the Cray Vector Processors yield an exact count of how many instructions (floating-point, integer, and misc) were executed on behalf of a program. The KSR-1 had a set of built-in registers that indicated not only clock cycle counts, but memory operations as well (cache hits, local memory hits, and cycle stalls waiting for remote shared-memory access). The data in these registers provide a cost-free measure of the work performed by a program on a processor-by-processor basis, and as such, are useful in determining an equitable load balance.

To obtain an evenly balanced decomposition in our MD simulations, we use past performance as a prediction of future work requirements. One reason this is viable is that the system of molecules, while undergoing some motion, is not moving all that much. The combination of this aspect of MD with the accurate performance information available leads to dynamic spatial decompositions that provide improved performance and are quick to compute.

The following definitions are needed to describe our work-based decomposition strategy.

- The dynamics work, w_i, performed by each processor since the last load-balancing operation (does not include instructions used for communication and synchronization)
- The total work, $W = \sum_{i=1}^{P} w_i$, performed by all processors since the last load-balancing operation
- The ideal (average) work, $\overline{w} = W/P$, to be performed by each processor for future steps
- The average amount of work, $a_i = w_i/n_i$, performed on behalf of each atom group on processor i (with n_i atom groups on processor i)
- The number of decompositions, d_x, d_y, d_z, in the x, y and z dimensions

4.4. **Spatial Adaptation.** The initial intuition for adjusting the spatial decomposition is to somehow change the shape of each processor's sub-domain to achieve $\overline{w}$ work on every processor. This is a difficult problem to solve considering the three-dimensional nature of the simulation space.

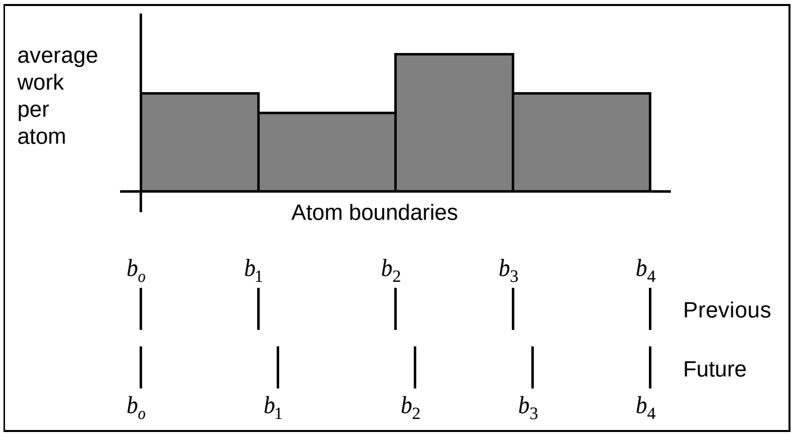

FIGURE 8. Unbalanced work loads on a set of processors. If the boundaries are moved as shown, then the work will be more in balance.

Instead, if we change the focus to the boundaries instead of the volumes, and place the boundaries one dimension at a time (as is done in the ORB decomposition), then a straightforward $O(P)$ implementation can be developed. Figure 8 shows a single dimension split into n subdivisions, with $n+1$ boundaries, of which only $n-1$ are movable (b_0 and b_n are naturally at the beginning and end of the space being divided). In Sigma, we first divide the space along the x-dimension into d_x balanced parts, then each of those into d_y parts, and finally, each of the "shafts" into d_z parts, using the following description.

Consider the repartitioning in a single dimension as shown in figure 8. Along the x-axis, the region boundaries separate atoms based on their position (atoms are sorted by x-position). The height of a partition represents the average work per atom in a partition, which as stated earlier, is not constant due to density changes in the data and optimizations

FIGURE 9. This graph shows two views of the adaptive decomposition working over time using 8 processors. The upper traces show the number of T4-Lysozyme atom groups assigned to each processor. The lower traces show the percentage of time spent waiting in barriers by each process since the previous rebalancing step. At step 0, an equal number of atom groups is assigned to each processor, since nothing is known about the computational costs. Subsequently, the decomposition is adjusted based on the work performed (alternatively, the percent of time spent waiting). If a processor has a high waiting time at step k, then it should receive a larger number of atom groups than it had at step $k-1$. The graph also shows the performance of our adaptive method over an ORB method using atom count as a weight (the decomposition used for the first step). Our adaptive decomposition reduces the waiting time of all processors to less than 5%, vs. the 20% waiting time of the ORB decomposition.

that have been introduced. Thus, the area of the box for each partition is w_i, and the sum of the areas is W. The goal is to place b_i far enough from b_{i-1} such that the work (represented by area) is as close to $\overline{w}$ as possible. This placement of the boundaries can be computed in $O(n)$ time for n boundaries. While this does not lead to an exact solution, a few iterations of work followed by balancing yield very good solutions where the boundaries settle down.

Figure 9 shows how the boundary motion in Sigma settles down as the simulation progresses. Initially, space is decomposed as if each atom group causes the same amount of work. This decomposes space using ORB such that all processors have the same number of atom groups. As the simulation progresses, boundaries are moved to equalize the load based on historical work information. This makes the more heavily loaded spaces smaller, adding more volume (and therefore atoms) to the lightly loaded spaces. As the simulation progresses, the number of atom groups shifted to/from a processor is reduced, but still changing due to the dynamic nature of the simulation and inexact balance.

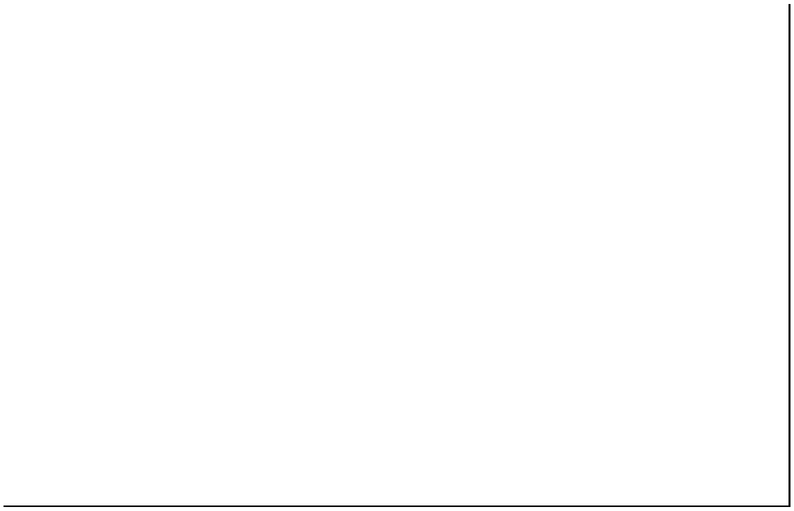

FIGURE 10. Parallel Performance of Sigma. This graph shows the number of simulations steps per second achieved with several molecular systems, T4-Lysozyme (13642 atoms), SS-Corin (3948 atoms), and brH (3964 atoms). The data plotted represent the performance of the last 200fs of a 600fs simulation, which allowed the dynamic decomposition to stabilize prior to measurement. A typical simulation would carry on from this point, running for a total of 10^6fs (500,000 simulation steps) in simulated time, at roughly these performance levels.

Figure 10 shows the performance of several different molecular systems being simulated on varying numbers of processors on both the Power Challenge and the Origin2000 (at NCSA). The y-axis shows the number of simulation steps executed per second, which is indeed the metric of most concern to the scientists using the simulator. We ran tests using decompositions where we set $P = (d_x \cdot d_y \cdot d_z)$ to $1 = (1 \cdot 1 \cdot 1)$, $2 = (2 \cdot 1 \cdot 1)$, $4 = (2 \cdot 2 \cdot 1)$, $6 = (3 \cdot 2 \cdot 1)$, $8 = (2 \cdot 2 \cdot 2)$, $9 = (3 \cdot 3 \cdot 1)$, $12 = (3 \cdot 2 \cdot 2)$ and $16 = (4 \cdot 2 \cdot 2)$.

There are several conclusions to be drawn from the performance graph, the most important of which is the steady linear improvement in performance with increasing processors. The similar slopes of the performance trajectories for the different datasets shows that the performance scales similarly for each dataset. The average speedup on 8 processors for the data shown is 7.59.

The second point is the “hiccup” on the Origin2000 when we increased the number of processors from 8 to 9. Our conjecture is that it is not change in the balance of decomposition by dimension, but the inability to measure performance adequately to include remote memory reference costs on the Origin 2000, since the slowdown occurs in all molecular systems only when run on the Origin 2000. The memory system of the Power Challenge is a shared-bus, thus in going from 8 to 9 processors, the only impact on performance is that 9 processors are sharing the same bus instead of 8. On the Origin 2000, memory is tree-based, so an additional ninth processor’s remote-memory accesses will all be much further than the initial 8 processor’s (assuming they are on a local tree). While this conjecture has not been proved, it is our best guess at this time.

The third point is that the performance difference between the two architectures is generally very small, despite the improved memory bandwidth of the Origin 2000 over the Power Challenge. Our conjecture to explain this, based on this experiment and the BSP modeling in the previous section, is that the calculation of non-bonded interactions involves a small enough dataset such that most, if not all, atom data can remain in cache once it has been fetched.

5. Conclusions

We are excited to achieve performance that enables interactive molecular dynamics on systems of molecules relevant to biochemists. Our performance results also enable rapid execution of large timescale simulations, allowing many experiments to be run in a timely manner. The methodology described shows the use of high-level modeling to understand what the critical impediments to high-performance are, followed by detailed implementations where optimizations (including model violations) can take place to achieve even better performance.

Prior to our BSP modeling study, we could only conjecture that load-balancing was the most important aspect of parallelism to explore for high performance parallel MD using a spatial decomposition. The BSP model developed supports this claim, and also leads us to the conclusion that the use of 2 or 4 workstations using Ethernet communications should provide good performance improvements, despite the relatively slow communications medium. Unfortunately, we have not yet demonstrated this, as our implementation is based upon a shared-memory model, and will require further effort to accommodate this model. From the BSP study, we are considering using the BSP library from BSP-Worldwide [6], as a basis for a distributed memory implementation, due to its simplicity, portability, predictability, and our familiarity with the model. Of course, with the growing market of multiprocessor desktop workstations, the need for a message-passing implementation is diminishing.

Our BSP study of MD also shows that processor speed is far more important than communication speed, so that paying for a high-speed communications system is not necessary for high performance MD simulations. This provides economic information for the acquisition of parallel hardware, since systems with faster communication usually cost substantially more.

And finally, we’ve shown that good parallelization strategies that rely on information from the underlying hardware or operating system can be economically obtained and effectively used to create scalable parallel performance. Much to our disappointment, we have not been able to test our method on machines with large numbers of processors, as the trend with shared-memory parallel computers is to use small numbers of very fast processors.

The results presented here represent a circular feedback mechanism, where the difficulties parallelizing an existing MD simulator laid the foundation for a useful modeling study. The model eliminates details and helps expose impediments, allowing solutions to be developed and studied. But the model could not have been successfully developed without the earlier implementation experiences. The demand for even higher performance (which cannot be met with hardware) will again force us to explore models, where we can quantify the benefits of algorithmic modifications such as multi-frequency recalculation of forces, a problem that would be too difficult to explore in a detailed implementation model.

We gratefully acknowledge the support of NCSA and their “friendly user account” program in support of this work.

REFERENCES

[1] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. Technical report, Department of Mathematics, Utrecht University, April 27 1994.

[2] John A. Board, Jr., Ziyad S. Hakura, William D. Elliott, and William T. Rankin. Scalable variants of multipole-accelerated algorithms for molecular dynamics applications. Technical Report TR94-006, Electrical Engineering, Duke University, 1994.

[3] David A. Case, Jerry P. Greenberg, Wayne Pfeiffer, and Jack Rogers. AMBER – molecular dynamics. Technical report, Scripps Research Institute, see [11] for additional information, 1995.

[4] 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*, Knoxville, TN, May 1994. ftp://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93356-S.

[5] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An $n\log(n)$ method for ewald sums in large systems. *J. Chem. Phys.*, 98(12):10089–10092, June 1993.

[6] Jonathan Hill and Bill McColl. *An Initial Proposal for the BSP Worldwide Standard Library*. Oxford University. http://www.bsp-worldwide.org/standard/stand1.htm.

[7] Yuan-Shin Hwang, Raja Das, Joel H. Saltz, Milan Hodoščček, and Bernard Brooks. Parallelizing molecular dynamics programs for distributed memory machines: An application of the chaos runtime support library. In *Proceedings of the Meeting of the American Chemical Society*, August 21–22 1994.

[8] Monica Kum and Lei Wang. Analysis of MD simulations using bulk synchronous parallel architectures. Independent study report, UNC Computer Science, 1996.

[9] Jonathan Leech, Jan F. Prins, and Jan Hermans. SMD: Visual steering of molecular dynamics for protein design. *IEEE Compuational Science & Engineering*, 3(4):38–45, Winter 1996.

[10] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kale, Robert D. Skeel, and Klaus Schulten. NAMD - a parallel, object-oriented molecular dynamics program. *Journal of Supercomputing Applications and High Performance Computing*, In press.

[11] D.A. Pearlman, D.A. Case, J.W. Caldwell, W.R. Ross, T.E. Cheatham III, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman. AMBER, a computer program for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to elucidate the structures and energies of molecules. *Computer Physics Communications*, 91:1–41, 1995.

[12] J. P. Ryckaert, G. Ciccotti, and H. 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.

[13] T. Schlick, E. Barth, and M. Mandziuk. Biomolecular dynamics at long timesteps: Bridging the time scale gap between simulation and experimentation. *Annu. Rev. Biophys. Biomol. Struct.*, 26:179–220, 1997.

[14] L. F. Valiant. A bridging model for parallel computation. *CACM*, 33:103–111, 1990.

[15] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Supercomputing ’93*, pages 12–21, Los Alamitos, 1993. IEEE Comp. Soc.

LARS S. NYLAND (NYLAND@CS.UNC.EDU), JAN F. PRINS (PRINS@CS.UNC.EDU), HYE-CHUNG “MONICA” KUM (KUM@CS.UNC.EDU), LEI WANG (WANGLE@CS.UNC.EDU): DEPARTMENT OF COMPUTER SCIENCE, CB #3175, UNIVERSITY OF NORTH CAROLINA, CHAPEL HILL, NC 27599-3175

Jan Hermans (hermans@med.unc.edu), Ru Huai Yun (huai@femto.med.unc.edu), Department of Biochemistry and Biophysics, University of North Carolina, Chapel Hill, NC 27599