Sandia Technical Report SAND94–1862 — June 1994
to appear in J of Computational Chemistry

A New Parallel Method for Molecular Dynamics Simulation of Macromolecular Systems

Steve Plimpton and Bruce Hendrickson
Parallel Computational Sciences Department 1421, MS 1111
Sandia National Laboratories
Albuquerque, NM 87185-1111
(505) 845-7873
sjplimp@cs.sandia.gov

Keywords: molecular dynamics, parallel computing, macromolecular modeling

Abstract

Short–range molecular dynamics simulations of molecular systems are commonly parallelized by replicated–data methods, where each processor stores a copy of all atom positions. This enables computation of bonded 2–, 3–, and 4–body forces within the molecular topology to be partitioned among processors straightforwardly. A drawback to such methods is that the inter–processor communication scales as N, the number of atoms, independent of P, the number of processors. Thus, their parallel efficiency falls off rapidly when large numbers of processors are used. In this article a new parallel method for simulating macromolecular or small–molecule systems is presented, called force–decomposition. Its memory and communication costs scale as $N/\sqrt{P}$, allowing larger problems to be run faster on greater numbers of processors. Like replicated–data techniques, and in contrast to spatial–decomposition approaches, the new method can be simply load–balanced and performs well even for irregular simulation geometries. The implementation of the algorithm in a prototypical macromolecular simulation code ParBond is also discussed. On a 1024–processor Intel Paragon, ParBond runs a standard benchmark simulation of solvated myoglobin with a parallel efficiency of 61% and at 40 times the speed of a vectorized version of CHARMM running on a single Cray Y–MP processor.

This work was partially supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the DOE under contract No. DE–AC04–76DP00789.

1 Introduction

Molecular dynamics (MD) is a widely-used computational tool for simulating liquids and solids at an atomistic level [1]. Macromolecular systems such as polymers, proteins, and DNA are particularly interesting to study with MD because the conformational shape of the molecules often determines their functional and catalytic properties. Such systems are also computationally challenging to simulate because (1) in the absence of crystal periodicity large numbers of atoms must often be included in the model, and (2) interesting events such as molecular diffusion or conformational changes typically occur on long timescales relative to the femtosecond–scale timesteps of the MD model.

MD simulations are natural candidates for implementation on parallel computers because the forces on each atom or molecule can be computed independently [15, 25, 31]. In this paper, we focus on MD simulations of molecular systems which require computation of *bonded* forces within the topology of the simulated molecules in addition to the standard *non–bonded* van der Waals and Coulombic forces. To achieve good parallel performance these bonded computations must be distributed evenly across the processors in conjunction with parallelization of the non–bonded pairwise computations. We also limit our scope to short–range MD models where the non–bonded forces are truncated, so that each atom interacts only with other atoms within a specified cutoff distance. Examples of widely–used commercial and research codes which implement this model include CHARMM, GROMOS, AMBER, and DISCOVER. The computation in this case scales linearly with N, the number of atoms, since each atom interacts with a roughly constant number of neighbors. While more accurate, MD models with long–range forces are more expensive to compute with, even if hierarchical methods [3] or multipole approximations [17] are used. The computation in these methods is typically partitioned into short– and long–range portions. The short–range portion involves a direct summation of pairwise interactions with near neighbors and thus can be parallelized by the methods described in this paper. The long–range portion requires different strategies to efficiently parallelize [13, 33] and is beyond the scope of this paper.

The most commonly used technique for parallelizing short–range MD simulations of molecular systems is known as the *replicated–data* (RD) method [31]. Numerous parallel algorithms and simulations have been developed based on this approach [7, 11, 12, 18, 21, 23, 29, 32]. Typically, each processor stores a copy of all the atom positions in the simulation. It uses this vector of information to compute non–bonded forces for the subset of atoms assigned to it. The bonded force computation can be simply parallelized in this scheme, since each processor can compute the force between any group of bonded atoms. The drawback to the RD method is that its memory and communication cost scale as N *independent* of P, the number of processors used. Thus, on large numbers of processors communication costs dominate, and the algorithm becomes inefficient.

A competing parallel method is known as *geometric–* or *spatial–decomposition* (SD) [15, 25]. In this approach, the simulation domain is broken into P pieces, one per processor. Each processor computes forces on only the atoms in its sub–domain. In the large N limit, the technique scales optimally as N/P, since each processor need only acquire information from processors who own neighboring sub–domains. However, SD algorithms have not been as widely used for molecular simulation [8, 9, 10, 14, 33] due to their complexity (particularly for bonded force computation) and the difficulty of achieving good load–balance across processors for irregularly–shaped or dynamically–changing domains.

We have developed a new parallel algorithm, called *force–decomposition* (FD), which is particularly appropriate for simulations of molecular systems, be they linear–chain polymers or large macromolecules. Though nearly as simple to implement as the RD technique, it reduces the communication cost and memory requirements by a factor of $\sqrt{P}$. This allows many more processors to be used efficiently in a given simulation. Although the new algorithm's scaling is not the optimal N/P of the SD methods, it is similar to RD in that it is geometry–independent and thus can be simpler to load–balance effectively than SD methods. In Section 7 we show that when used on more than a few dozen processors the FD method can offer faster parallel performance than RD techniques and argue that it will often be the fastest of the three parallel methods (RD, FD, SD) for many kinds of molecular simulations of moderate size (up to many tens of thousands of atoms).

In earlier work we described FD techniques for simulations with pairwise Lennard Jones [25] and embedded–atom method [26] potentials for metals. The addition of many–body bonded forces is an important complication requiring special treatment; our previous efforts in this direction are briefly discussed in references [20, 27]. In this paper we detail an enhanced version of the FD method with reduced communication cost (motivated by an algorithm for parallel matrix–vector multiplication discussed in reference [22]). We also provide, for the first time, a full description of how to parallelize the computation of bonded forces within the context of the FD method.

In the next section we outline the computations performed in MD simulations of molecular systems. The following Sections 3 and 4 briefly describe RD and SD methods, to put the new algorithm in context. The FD method is detailed in Sections 5 and 6. We have implemented both RD and FD algorithms in a macromolecular MD code called ParBond. We use it to compare the performance of the two approaches in Section 7 for benchmark simulations of myoglobin and liquid–crystal systems.

2 MD Simulations of Molecules

In MD simulations of molecular systems two kinds of interactions contribute to the total energy of the ensemble of atoms — non-bonded and bonded. These energies are expressed as simple empirical relations [6]; the desired physics or chemistry is simulated by specifying appropriate coefficients. The energy E_{nb} due to non–bonded interactions is typically written as

$$E_{nb} = \sum_i \sum_j \frac{q_i q_j}{r} + \sum_i \sum_j \epsilon_{ij} \left[(\frac{\sigma_{ij}}{r})^{12} - (\frac{\sigma_{ij}}{r})^6 \right] \tag{1}$$

where the first term is Coulombic interactions and the second is van der Waals, r is the distance between atoms i and j, and all subscripted quantities are user–specified constants. In short–range simulations, the summations over i and j are evaluated at each timestep so as to only include atom pairs within a cutoff distance r_c, such that $r < r_c$. The bonded energy E_b for the system in the harmonic approximation can be written as

$$E_b = \sum_{\text{bonds}} K_b(r - r_0)^2 + \sum_{\text{angles}} K_\theta(\theta - \theta_0)^2 + \sum_{\text{dihedrals}} K_{\phi_p}[1 + d_p \cos(n_p\phi)] + \sum_{\text{impropers}} K_\phi(\phi - \phi_0)^2 \quad (2)$$

where the first term is 2–body energy, the second is 3–body energy, and the last two are 4–body interactions for torsional dihedral and improper dihedral energies within the topology of the molecules. The distance r and angles θ and ϕ are computed for each interaction as a function of the atomic positions; the subscripted quantities are constants. In contrast to the non–bonded energy, the summations in this equation are explicitly enumerated by the user to setup the simulation, i.e. the connectivities of the molecules are fixed. In the MD simulation, derivatives of equations 1 and 2 yield force equations for each atom which are integrated over time to generate the motion of the ensemble of atoms.

3 Replicated–Data Algorithm

The first of the two basic methods that have been used to parallelize the computation of equations 1 and 2 is *replicated–data* (RD) or *atom–decomposition* algorithms. Each processor is assigned a subset of N/P atoms and updates their positions and velocities for the duration of the simulation, regardless of where they move in the physical domain. In this setting, the computational work involved in evaluating equation 1 can be represented by the $N \times N$ force matrix F. The (ij) element of F represents the non–bonded force on atom i due to atom j. Note that F is sparse due to short–range forces. To take advantage of Newton's 3rd law, we also set $F_{ij} = 0$ when $i > j$ and $i + j$ is even, and likewise set $F_{ij} = 0$ when $i < j$ and $i + j$ is odd. Conceptually, F is now colored like a checkerboard with red squares above the diagonal set to zero and black squares below the diagonal also set to zero. Zeroing half the matrix elements can also be accomplished by striping F in various ways [30]. We also define x and f as vectors of length N which store the position and total force on each atom. For a 3–d simulation, x_i would store the three coordinates of atom i.

With these definitions, RD algorithms assign each processor a sub–block of F which consists of N/P rows of the matrix, as shown in Figure 1. If z indexes the processors from 0 to $P - 1$, then processor P_z computes non–bonded forces in the F_z sub–block of rows. It also is assigned the corresponding sub–vectors of length N/P denoted by x_z and f_z.

The computation of the non–bonded force F_{ij} requires only the two atom positions x_i and x_j. But to compute all the forces in F_z, processor P_z will need the positions of many atoms owned by other processors. In Figure 1 this is represented by having the horizontal vector x at the top of the figure span all the columns of F. This implies each processor must store a copy of all N atom positions. To take advantage of Newton's

3rd law, each processor will also compute forces on atoms it does not own, so it must also store a copy of all N forces represented by the force vector f.

Using these definition, a typical RD algorithm is outlined in Figure 2 with the dominating term in the computation or communication cost of each step listed on the right. We assume at the beginning of the timestep that each processor knows the current positions of all N atoms, i.e. each has an updated copy of the entire x vector. In step (1) of the algorithm, the non–bonded forces in matrix sub–block F_z are computed. This is typically done using neighbor lists to tag the interactions that are likely to be non–zero at a given timestep. In the parallel algorithm each processor constructs lists for its sub–block F_z once every few timesteps. As each pairwise non–bonded interaction between atoms i and j is computed, the force components are summed twice into the processor's local copy of f, once in location i and once (negated) in location j, so that F_z is never actually stored as a matrix. Step (1) scales as N/P, the number of non–zero non–bonded interactions computed by each processor. In step (2) the bonded forces in equation 2 are computed. This can be done by spreading the loops implicit in the summations of equation 2 evenly across the processors. Since each processor knows the positions of all atoms, it can compute any of the terms in equation 2, and sum the resulting forces into its local copy of f. This step also scales as N/P, since there are a small number of bonded interactions per atom.

In step (3), the force vector copies are summed across all P processors in such a way that each processor ends up with the total force on only its N/P atoms. This is called a *fold* operation [4, 16, 25] and scales optimally as N, the volume of data in the force vector f. We note that the fold operation is less costly than a global sum operation where each processor ends up with the total force on all N atoms, as is done in the RD algorithms discussed in [12, 21, 23, 32]. A global sum operation typically scales as $N \log_2(P)$ and so on 256 processors is 8 times more expensive than a fold. The N/P forces resulting from the fold are used to update atom positions and velocities in step (4). Finally, in step (5) the new atom positions in x_z are shared among all P processors in preparation for the next timestep. This is called an *expand* operation [4, 16, 25] and is essentially the inverse of the fold operation. Now each processor starts with a small N/P piece of the position vector and ends up with a copy of the entire N–length vector. The cost of this communication step also scales as N.

The chief advantage of the RD algorithm we have outlined is simplicity. It straightforwardly divides the MD force computation and integration evenly across the processors so long as each F_z block has roughly the same number of non–zero elements. This may not be the case if, for example, some atoms have fewer neighbors than others due to non–uniform densities; load–imbalance is the result. This can often be overcome by randomly permuting the order of the atoms before the simulation begins. This is a *static* load–balancing method which scatters non–zero elements uniformly throughout the F matrix. Alternatively, Young and Brooks [34] have proposed a dynamic method for preserving load–balance in RD algorithms. They adjust the size of the sub–blocks in Figure 1 as the simulation progresses. Individual processors lose or gain atoms to insure continual load–balance.

The chief drawback of the RD algorithm is that it requires global communication in the fold and expand steps, as each processor must acquire information held by all the other processors. Because this communication scales as N, independent of P, it limits the number of processors that can be used effectively. Similarly, the RD algorithm requires each processor store a copy of the entire N–length position and force vectors. This memory overhead often limits the size of problems that can be simulated.

4 Spatial–Decomposition Algorithm

The second parallel method for computing equations 1 and 2 exploits the locality of the short–range forces by assigning to each of the P processors a small region of the simulation domain, call it D_z, where again z indexes the processors from 0 to $P-1$. This is a *geometric–* or *spatial–decomposition* (SD) of the workload; an algorithm of this form is outlined in Figure 3. Each processor will update the positions x_z of only the atoms in its sub–domain. To do this it will need to know not only x_z but also positions y_z of atoms owned by processors whose sub–domains are within a cutoff distance r_c of its sub–domain. Similarly, as it computes forces f_z on its atoms, it will compute components of forces g_z on the nearby atoms (Newton's 3rd law). With these definitions, steps (1) and (2) of the algorithm are the computation of non–bonded and bonded forces for interactions involving the processor's atoms. These steps scale as the number of atoms N/P in each processor's sub–domain. In step (3) the g_z forces computed on neighboring atoms are communicated to processors owning neighboring sub–domains. The received forces are summed with the previously computed f_z to create the total force on a processor's atoms. The scaling of this step depends on the length of the force cutoff relative to the sub–domain size. We list it as Δ and discuss it further below. Step (4) updates the positions of the processor's atoms. In step (5) these positions are communicated to processors owning neighboring sub–domains so that all processors can update their y_z list of nearby atoms. Finally, periodically (usually when neighbor lists are created), atoms which have left a processor's sub–domain must be moved in step (6) to the appropriate new processor.

The above description ignores many details of an effective SD algorithm [25], but shows the algorithm's overall scaling is the optimal N/P, so long as the communication costs Δ can be minimized. In the limit of large N/P ratios, Δ scales as the surface–to–volume ratio $(N/P)^{(2/3)}$ of each processor's sub–domain. If each processor's sub–domain is roughly equal in size to the force cutoff distance, then Δ scales as N/P and each processor receives N/P atom positions from each of its neighboring 26 processors (in 3–d). In practice, however, there are several obstacles to minimizing Δ and achieving high parallel efficiencies in MD simulations of molecular systems.

(A) Molecular systems may be simulated in a vacuum or with surrounding solvent that does not uniformly fill a 3–d box. In this case it is non–trivial to divide the simulation domain so that every processor's sub–domain has an equal number of atoms in it. Load–imbalance is the result.

(B) Because of the $1/r$ dependence of Coulombic energies in equation 1, long cutoffs are often used in simulations of organic materials. Thus a processor's sub–domain may be much smaller than the cutoff. The result is considerable extra communication in steps (3) and (5) to acquire needed atom positions and forces, i.e. Δ no longer scales as N/P, but as the cube of the cutoff distance r_c.

(C) As atoms move to new processors in step (6), molecular connectivity information must be exchanged and updated between processors. The extra coding to manipulate the appropriate data structures and optimize the communication performance of the data exchange subtracts from the parallel efficiency of the algorithm.

We discuss several SD implementations [8, 10, 14] further in Section 7.

5 Force–Decomposition Algorithm

We now present the new force–decomposition (FD) algorithm. Its communication cost lies in between that of the RD and SD approaches. The FD method partitions the force matrix F by blocks rather than by rows as in Section 3. Block–decompositions of matrices are common in linear algebra algorithms for parallel machines [5, 19, 22] which sparked our interest in the idea for short–range MD simulations. The assignment of sub–blocks of the force matrix to processors with a calendar ordering of the processors is depicted in Figure 4. We assume for ease of exposition that P is an even power of 2, though it is straightforward to implement the algorithm on any number of processors which can be logically mapped to a square or rectangular grid. As before, we let z index the processors from 0 to $P-1$; processor P_z owns and will update the N/P atoms stored in the sub–vector x_z.

The block–decomposition in Figure 4 is actually of a permuted force matrix F' which is formed by rearranging the columns of the original checkerboarded F in a particular way. If we order the x_z pieces in *row–order* (across the rows of the matrix), they form the usual position vector x which is shown as a vertical bar at the left of the figure. Were we to have x span the columns as in Figure 1, we would form the force matrix as before. Instead, we span the columns with a permuted position vector x', shown as a horizontal bar at the top of Figure 4, in which the x_z pieces are stored in *column–order* (down the columns of the matrix). Thus, in the 16–processor example shown in the figure, x stores each processor's piece in the usual order $(0, 1, 2, 3, 4, ..., 14, 15)$ while x' stores them as $(0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15)$. Now the (ij) element of F' is the force on atom i in vector x due to atom j in permuted vector x'.

The ${F'}_z$ sub–block owned by each processor P_z is of size $(N/\sqrt{P}) \times (N/\sqrt{P})$. To compute the non–bonded forces in ${F'}_z$, processor P_z must know one $N/\sqrt{P}$–length piece of each of the x and x' vectors, which we denote as x_α and ${x'}_\beta$. As these elements are computed they will be accumulated into corresponding force sub–vectors f_α and ${f'}_\beta$. The Greek subscripts α and β each run from 0 to $\sqrt{P}-1$ and reference the row and column position occupied by processor P_z. Thus for processor 6 in the figure, x_α consists of the x sub–vectors $(4, 5, 6, 7)$ and ${x'}_\beta$ consists of the x' sub–vectors $(2, 6, 10, 14)$.

The FD algorithm is outlined in Figure 5. As before, each processor has updated copies of the needed atom positions x_α and ${x'}_\beta$ at the beginning of the timestep. In step (1), the non–bonded forces in matrix sub–block ${F'}_z$ are computed. As before, neighbor lists can be used to tag the $O(N/P)$ non–zero interactions in F'_z. As each force is computed, the result is summed into the appropriate locations of both f_α and f'_β to account for Newton's 3rd law. In step (2) each processor computes an N/P fraction of the bonded interactions. Since each processor knows only a subset of atom positions, this must be done differently than in the RD algorithm. We describe this computation in the next section.

In step (3), the force on each processor's atoms is acquired. The total force on atom i is the sum of elements in row i of the force matrix minus the sum of elements in column i', where i' is the permuted position of column i. Step (3a) performs a fold within each row of processors to sum the first of these contributions. There is a key difference between this fold operation and the replicated–data (RD) fold of Figure 2. In this case the vector f_α being folded is only of length $N/\sqrt{P}$ and only the $\sqrt{P}$ processors in one row are participating in the fold. Thus this operation scales as $N/\sqrt{P}$ instead of N as in the RD algorithm. Similarly, in step (3b), a fold is done within each column of F'. The two contributions to the total force are joined in step (3c). In step (4), f_z is used to update the N/P atom positions in x_z. Steps (5a–5b) share these updated positions with all the processors that will need them for the next timestep. These are the processors which share a row or column with P_z. First, in (5a), the processors in row α perform an expand of their x_z sub–vectors so that each acquires the entire x_α. As with the fold, this operation scales as the $N/\sqrt{P}$ length of x_α instead of as N as it did in the RD algorithm. Similarly, in step (5b), the processors in each column β perform an expand of their x_z. As a result they all acquire x'_β and are ready to begin the next timestep.

In the FD method, processors will have equal work to do only if all the matrix blocks F'_z are uniformly sparse. If the atoms are ordered geometrically this will not be the case even for problems with uniform density. This is because such an ordering creates a force matrix with diagonal bands of non–zero elements. As in the RD case, a random permutation of the atom ordering produces the desired effect.

The key feature of the FD method is that the communication operations in steps (3) and (5) now scale as $N/\sqrt{P}$ rather than as N as was the case with the RD algorithm. Likewise, memory costs for position and force vectors are reduced by the same $\sqrt{P}$ factor. When run on large numbers of processors this can significantly reduce the time spent in communication. It is important to note that the communication cost of the algorithm is *independent* of the force cutoff distance, unlike SD methods, whose communication cost grows as the cube (volume) of the cutoff. The FD method also retains the simplicity of the RD technique; it can be implemented using the same expand and fold communication routines.

6 Bonded Forces

We now return to step (2) of the FD algorithm of the preceding section – computation of bonded forces in the MD model. Because each processor only stores two $N/\sqrt{P}$–length portions of the atom positions, we must insure some processor knows the 2, 3, or 4 atom positions required to compute every one of the bonded interactions listed in equation 2. Similar to the randomization process for load–balancing purposes, this can be accomplished as a pre–processing step by a proper ordering of the atoms. We wish to assign atoms to processors in such a way that all bonded interactions can be computed, while preserving load–balance.

We define a *cluster* of atoms as a collection of one or more atoms. They are not necessarily connected within the topology of a molecule. We also define a *group* of processors as all the processors in one row of the force matrix F' of Figure 5. Thus there are $\sqrt{P}$ processors in each of $\sqrt{P}$ groups. Our pre–processing algorithm proceeds in three stages, each of which is computationally simple and can be performed quickly on a workstation, even for very large MD simulations.

(Stage 1) Create clusters.

(Stage 2) Assign clusters to groups.

(Stage 3) Assign atoms to processors within each group.

The first stage is begun by assigning each atom to its own cluster. We then process the list of 4–body interactions in a random order. We demand that each set of 4 interacting atoms be in no more than 2 clusters. If this is not the case we merge the two smallest clusters to form a new single cluster. This is done twice if the 4 atoms were initially in 4 different clusters. When all 4–body interactions have been processed, the list of 3–body interactions is treated similarly. For each set of 3 atoms, clusters are merged as needed to satisfy the constraint. The goal of this stage is to maximize the total number of clusters, with each cluster being of minimal size. Typically clusters will grow no larger than a dozen or two atoms, and there will be many small 1 or 2 atom clusters, particularly if small solvent molecules are included in the model.

The second stage of the algorithm is begun by sorting the clusters by size. Beginning with the largest, the clusters are then assigned one by one to whichever group of processors currently has the least number of total atoms. The goal of this stage is to assign equal numbers of atoms to each group; this is simple in practice since there are typically an order–of–magnitude or more clusters than groups.

In the final stage, the atoms in each group are assigned to individual processors within the group. This is done randomly so that atoms within a cluster end up spread across the $\sqrt{P}$ processors of the group and so that each processor ends up with roughly the same number of total atoms. The only exception to this rule is as follows. First, the list of 4–body interactions is scanned in a random order once more. If two of the 4 atoms are in one group and two in another, then both the atoms in one of the sets are assigned to the same (random) processor within their group.

If SHAKE [28] or RATTLE [2] is being used to constrain bond lengths, these stages can be modified to treat small subsets of coupled atoms as one unit. This insures all the atoms in a particular subset are assigned to one processor so that computation of the constraint can be parallelized over subsets.

When the stages are completed, each processor will have nearly equal numbers of atoms assigned to it. The randomness used in the stages also insures each processor will have roughly equal numbers of non–bonded forces to compute. Moreover we have guaranteed every bonded computation will be computable by one or more processors. This is automatic for the 2–body interactions, since some processor will own atom i in its x_α and atom j in its x'_β. For 3–body interactions, at least two of the 3 atoms will be in a particular

x_α, since they were assigned to the same cluster. Some processor in the α group will own the third atom in its x'_β. A similar argument holds for the 4–body interactions. The purpose of the final–stage restriction on two of the 4 atoms being assigned to the same processor (not just the same group) was to insure both atoms will be in the same x'_β.

Once atoms are assigned to processors, the final pre–processing step is to create lists of bonded interactions that will be computed by each processor in the MD simulation. When all the 2, 3, or 4 atoms of an interaction are in one x_α (or x'_β), any processor in that row (or column) of F' can compute the interaction. These can be assigned so as to tune the overall load–balance of the bonded computations.

7 Results

We have implemented both RD and FD algorithms in a parallel MD code we have written called ParBond. It is similar in concept (though not in scope) to the widely–used commercial and academic macromolecular codes CHARMM, AMBER, GROMOS, and DISCOVER. In fact, ParBond was designed to be CHARMM–compatible in the sense that it uses the same force equations as CHARMM [6]. Since the RD and FD methods both use the same communication primitives, ParBond simply has a switch that partitions the force matrix either by rows or sub–blocks as in Figures 1 and 4.

Brooks et al. [7] have done extensive benchmarking with CHARMM on a variety of machines with a large prototypical protein simulation. A 2534–atom myoglobin molecule (with an adsorbed CO) is surrounded by a shell of solvent water molecules for a total of 14,026 atoms. The resulting ensemble is roughly spherical in shape. The benchmark is a 1000–timestep simulation performed at a temperature of 300° K with a non–bonded force cutoff of 12.0 Å. Neighbor lists are created every 25 timesteps with a 14.0 Å cutoff; a total of 6.7 million atom pairs are stored in the neighbor lists. Timing results for the benchmark simulation run on different numbers of processors are shown in Figure 6.

All of the solid symbols in the figure are timings due to Brooks et al. [7]. The single processor Cray Y–MP timing of 3.64 secs/timestep is for a version of CHARMM they have optimized for vector processing. They have also developed a parallel version of CHARMM [7] using a RD algorithm similar to that discussed in Section 3. Timings with that version on an Intel iPSC/860 and the Intel Delta at CalTech are shown in the figure, as are timings for ParBond using the RD algorithm of Section 3, running on the Intel Paragon at Sandia. Taking into account that the i860XP floating point processors in the Paragon are about 30% faster than the i860XR chips in the iPSC/860 and Delta and that inter–processor communication is significantly faster on the Paragon, the two sets of RD timings are quite similar. Both curves show a marked roll–off in parallel efficiency above 64–128 processors due to the poor scaling of the all-to–all communication steps.

The timing results for ParBond using the FD algorithm are shown as open circles in the figure. The performance falls off less rapidly as processors are added; it is running 1.3 times faster than its RD counterpart on 256 processors (0.265 secs/timestep vs. 0.347) and 2.1 times faster on 1024–processors (0.0913 secs/timestep vs. 0.189). The 1024–processor timing is about 40 times faster than the single Y–MP processor timing. The dotted line in the figure represents perfect speed–up or 100% parallel efficiency for the ParBond code. The benchmark requires too much memory to run on a single processor so we estimated the ParBond one–processor timing by summing the 64–processor computation times for bonded and non–bonded forces and neighbor list construction only, not communication or load–imbalance times. (The one–processor Intel iPSC/860 timing for CHARMM is also an estimate [7].) The FD algorithm still has a relatively high parallel efficiency in ParBond of 61% on 1024 processors, as compared to 30% for the RD timing.

We have compared the performance of the new FD algorithm to the results in reference [7], not only because Brooks et al. have provided a standardized benchmark, but because their timings exhibit the best scaling of any RD implementation we have seen in the literature (62% parallel efficiency on 128 Intel iPSC/860 processors). These include a parallelization of CHARMM for a 32–processor Intel iPSC/860 [23], of a CHARMM–like code with long–range forces for a 24–processor transputer machine [18], of AMBER for a 128–processor nCUBE [12] and 512–processor Fujitsu AP1000 [29], of GROMOS for a 64–processor nCUBE and 128–processor Intel iPSC/860 [11], and of general molecular simulation codes for a 64–processor Intel iPSC/860 [32] and IBM workstation cluster [21]. All of these efforts show reduced parallel efficiencies as more processors are used due to the scaling problems inherent in the RD approach. Depending on the parallel machine and its communication characteristics the reported efficiencies fall as low as 10–15% on a few dozens to hundreds of processors and in some cases the overall speed–up is even reduced as more processors are added. The implementation of Sato et al. [29] is a notable exception with parallel efficiencies of 32 and 44% on 512 processors for their two benchmark calculations.

A comparison of ParBond performance across parallel machines for the RD and FD algorithms is shown in Table I. These are results from simulations of a 3x3x3 array of 250–atom liquid–crystal molecules (6750 total atoms) done in collaboration with Wright Patterson AFB researchers to study atomistic effects on macroscopic structure in thin film geometries [24]. The cutoffs for forces and neighbor lists used in this model were 8.0 and 10.0 Å. The performance was tested on three parallel machines: a 1024–processor nCUBE 2 and Intel Paragon at Sandia and a 512–processor Cray T3D at Cray Research. As in the previous figure, the difference in total run time (2nd number in each entry) between the two algorithms grows larger as the number of processors increases for all three machines. The listed communication times (1st number in each entry) are the key reason for the difference in performance.

It is worth re–emphasizing that the performance of both the RD and FD algorithms scale linearly with N, the number of atoms. Thus the FD advantage of 1.3–3.3x speed–up on 256–1024 Paragon processors in Figure 6 and Table I should hold for similar simulations of any sized system. Because of the lower memory requirements for the FD algorithm we can model systems with as many as 2 million atoms in the ParBond liquid–crystal simulations (8.0 Å cutoff) discussed above on 1024 processors of the Paragon (16 Mybtes/processor). By contrast, the largest liquid–crystal system we can model on that many processors with the RD option in ParBond is about 125,000 atoms. Similarly, Brooks et al. report their 14026–atom myoglobin benchmark (12.0 Å cutoff) is nearly the largest simulation they can perform using a RD technique

P	nCUBE 2		Intel Paragon		Cray T3D	
	RD	FD	RD	FD	RD	FD
64	21.8/62.7	5.04/48.5	3.05/12.1	.884/9.51	2.11/8.55	.745/7.28
128	22.0/43.8	4.04/27.5	3.16/8.30	.754/5.52	2.16/5.60	.667/4.19
256	22.2/34.1	2.89/15.6	3.55/6.96	.681/3.32	2.22/4.19	.586/2.60
512	22.2/29.2	2.35/9.67	3.68/6.13	.623/2.24	2.38/3.75	.547/2.13
1024	22.3/26.8	1.76/6.41	3.96/5.98	.651/1.78	---	---

Table I: *ParBond CPU timings (seconds per 100 timesteps) for replicated–data (RD) and force–decomposition (FD) algorithms on varying numbers of processors (P) on three parallel machines for a 6750–atom liquid–crystal simulation. The first number in each entry is for communication; the second is total time.*

on the 512–processor Intel Delta (about 12 usable Mbytes/processor) with parallel CHARMM [7].

Finally, while we have not written a spatial–decomposition code suitable for molecular systems, we contrast the performance of all three parallel algorithms in an atomic MD simulation. In this benchmark (described fully in reference [25]), individual atoms interact via only a Lennard Jones potential, analogous to the non–bonded component of a molecular simulation. Figure 7 shows the timing results for simulations of a 3–d box containing 10976 atoms in a liquid state (reduced density $\rho^* = 0.8442$) on the nCUBE 2 at Sandia. For a force cutoff of 2.5σ (55 neighbors/atom), the SD algorithm is faster than the other two for more than 32 processors. However, for a force cutoff of 5.0σ (440 neighbors/atom) more typical of the longer–range cutoffs used in molecular systems with Coulombic effects, the FD algorithm is the fastest choice on any number of processors. This is because its communication costs are independent of the cutoff length, whereas the SD algorithm must perform more communication as the cutoff length is increased. The performance of the SD code would be further reduced if atoms were not uniformly distributed within a regular 3–d box as in this benchmark. If N were increased (for fixed P), there would come a crossover point at which the SD code becomes faster [25] even for the longer cutoff problem, but these results indicate there is a regime (up to potentially many tens of thousands of atoms) where the FD approach will be the fastest algorithmic choice.

Spatial–decomposition implementations of molecular simulations on parallel machines with hundreds of processors are discussed in references [8, 10, 14]. Esselink and Hilbers developed their linear–chain polymer model for a 400–processor T800 Transputer machine. They partition the simulation box in 2–d columns with the 3rd dimension owned wholly on processor. For a 1400–particle benchmark system with cutoffs that encompassed 200 neighbors/atom they achieve a 50% parallel efficiency on 400 processors [14]. Brown et al. [8] implemented a SD algorithm with 3–d sub–domains for linear–chain polymers in rectangular boxes. They report an efficiency of about 40% on a 64000–atom simulation on 512 processors of the Fujitsu AP1000. The recently developed EulerGROMOS code of Clark et al. [10] is designed for general bio–molecules (with arbitrary connectivity patterns) and simulations of irregular–shaped domains. EulerGROMOS outperforms the earlier parallel replicated–data implementation of GROMOS [11] for 128 or more processors of the Intel Delta at CalTech. By using a hierarchical decomposition each processor in EulerGROMOS ends up with a rectangular–shaped sub–domain of variable size which may not align with its neighbors. This allows irregular–shaped domains to be partitioned across processors in a load–balanced fashion at the cost of extra communication overhead. They report a parallel efficiency of roughly 15% (without load–balancing) on 512 processors of the Delta for a 10914–atom simulation of solvated myoglobin with a 10.0 Å cutoff in a uniformly filled 3–d box and have simulated systems with up to 131,663 atoms at 25% parallel efficiency [10].

8 Conclusions

We have proposed a new strategy, force–decomposition, for parallelizing short–range MD simulations and illustrated its effectiveness on several parallel supercomputers. Like replicated–data and spatial–decomposition algorithms, the FD method's computational cost scales optimally as N/P. Its communication and memory costs scale as $N/\sqrt{P}$, in between the N scaling of RD and N/P scaling of SD.

The new FD algorithm is a particularly well–suited for simulations of molecular systems where long force cutoffs, computation of bonded forces, and irregular simulation geometries can degrade the parallel efficiency of SD implementations. Like RD techniques, the FD method is geometry–independent, so that even irregular or dynamically–changing problems are automatically load–balanced. The key advantage of the FD method over RD is its reduced memory costs and improved communication scaling which enables larger numbers of processors to be used more effectively to simulate a given problem, though in practice the performance difference between RD and FD methods may not become significant until more than a few dozen processors are used. While SD algorithms are clearly the method–of–choice for very large MD simulations due to their optimal N/P scaling, the relatively high parallel efficiencies and overall simplicity of the FD approach make it a fast option for many of the simulations currently being performed on massively parallel computers.

9 Acknowledgments

We thank Richard Judson and Grant Heffelfinger of Sandia for fruitful discussions regarding MD simulations of macromolecular systems and for suggesting improvements to a preliminary version of this manuscript. We also thank Grant for helping write the ParBond code and Ron Oldfield of Sandia for coding the bonded–force partitioning algorithm presented in Section 6. We thank Bernie Brooks of the National Institutes of Health for providing us with input files for the myoglobin benchmark problem. The liquid–crystal simulations discussed in the same section were performed in collaboration with Wright Patterson AFB researchers Ruth Pachter and Soumya Patnaik. The Cray T3D runs were performed on a machine at Cray Research with the assistance of Barry Bolding. Finally, the bonded–force routines (equation 2) in ParBond were written using a public–domain MD code authored by Andreas Windemuth (now at Columbia University) as a guide.

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Clarendon Press, Oxford, 1987.

[2] H. C. Andersen. RATTLE: A velocity version of the SHAKE algorithm for molecular dynamics calculations. *J. Comp. Phys.*, 52:24–34, 1983.

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

[4] M. Barnett, L. Shuler, S. Gupta, D. Payne, R. van de Geijn, and J. Watts. Building a high–performance collective communication library. In *Proc. Supercomputing '94*, pages 107–116. IEEE Computer Society Press, 1994.

[5] R. H. Bisseling and J. G. G. van de Vorst. Parallel LU decomposition on a transputer network. In G. A. van Zee and J. G. G. van de Vorst, editors, *Lecture Notes in Computer Science, Number 384*, pages 61–77. Springer–Verlag, 1989.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4:187–217, 1983.

[7] B. R. Brooks and M. Hodošček. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News*, 7:16–22, 1992.

[8] D. Brown, J. H. R. Clarke, M. Okuda, and T. Yamazaki. A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers. *Comp. Phys. Comm.*, 83:1–13, 1994.

[9] S. Chynoweth, U. C. Klomp, and L. E. Scales. Simulation of organic liquids using pseudo–pairwise interatomic forces on a toroidal transputer array. *Comp. Phys. Comm.*, 62:297–306, 1991.

[10] T. W. Clark, R. V. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proc. Scalable High Performance Computing Conference–94*, pages 95–102. IEEE Computer Society Press, 1994.

[11] T. W. Clark, J. A. McCammon, and L. R. Scott. Parallel molecular dynamics. In *Proc. 5th SIAM Conference on Parallel Processing for Scientific Computing*, pages 338–344. SIAM, 1992.

[12] S. E. DeBolt and P. A Kollman. AMBERCUBE MD, Parallelization of AMBER's molecular dynamics module for distributed–memory hypercube computers. *J. Comp. Chem.*, 14:312–329, 1993.

[13] H. Q. Ding, N. Karasawa, and W. A. Goddard III. Atomic level simulations on a million particles: The cell multipole method for Coulomb and London interactions. *J. Chem. Phys.*, 97:4309, 1992.

[14] K. Esselink and P. A. J. Hilbers. Efficient parallel implementation of molecular dynamics on a toroidal network: II. Multi–particle potentials. *J. Comp. Phys.*, 106:108–114, 1993.

[15] D. Fincham. Parallel computers and molecular simulation. *Molec. Sim.*, 1:1–45, 1987.

[16] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors: Volume 1.* Prentice Hall, Englewood Cliffs, NJ, 1988.

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

[18] H. Heller, H. Grubmuller, and K. Schulten. Molecular dynamics simulation on a parallel computer. *Molec. Sim.*, 5:133–165, 1990.

[19] B. Hendrickson and D. Womble. The torus–wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15:1201, 1994.

[20] B. A. Hendrickson and S. J. Plimpton. Parallel many–body simulations without all–to–all communication. *J. Par. and Dist. Comp.*, 27:15, 1995.

[21] J. F. Janak and P. C. Pattnaik. Protein calculations on parallel processors: II. Parallel algorithm for forces and molecular dynamics. *J. Comp. Chem.*, 13:1098–1102, 1992.

[22] J. G. Lewis and R. A. van de Geijn. Distributed memory matrix–vector multiplication and conjugate gradient algorithms. In *Proc. Supercomputing '93*, pages 484–492. IEEE Computer Society Press, 1993.

[23] S. L. Lin, J. Mellor-Crummey, B. M. Pettit, and G. N. Phillips Jr. Molecular dynamics on a distributed–memory multiprocessor. *J. Comp. Chem.*, 13:1022–1035, 1992.

[24] S. S. Patnaik, R. Pachter, S. J. Plimpton, and W. W. Adams. Molecular dynamics simulation of a cyclic siloxane based liquid crystalline material. In *Electrical, Optical, and Magnetic Properties of Organic Solid State Materials*, volume 328, pages 711–716. Materials Research Society Symposium Proc., Fall 1993.

[25] S. J. Plimpton. Fast parallel algorithms for short–range molecular dynamics. *J. Comp. Phys.*, 117:1–19, 1995.

[26] S. J. Plimpton and B. A. Hendrickson. Parallel molecular dynamics with the embedded atom method. In *Materials Theory and Modeling*, volume 291, pages 37–42. Materials Research Society Symposium Proc., Fall 1992.

[27] S. J. Plimpton, B. A. Hendrickson, and G. S. Heffelfinger. A new decomposition strategy for parallel bonded molecular dynamics. In *Proc. 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 178–182. SIAM, 1993.

[28] 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.

[29] H. Sato, Y. Tanaka, H. Iwama, S. Kawakika, M. Saito, K. Morikami, T. Yao, and S. Tsutsumi. Parallelization of AMBER molecular dynamics program for the AP1000 highly parallel computer. In *Proc. Scalable High Performance Computing Conference–92*, pages 113–120. IEEE Computer Society Press, 1992.

[30] H. Schreiber, O. Steinhauser, and P. Schuster. Parallel molecular dynamics of biomolecules. *Parallel Computing*, 18:557–573, 1992.

[31] W. Smith. Molecular dynamics on hypercube parallel computers. *Comp. Phys. Comm.*, 62:229–248, 1991.

[32] W. Smith and T. R. Forester. Parallel macromolecular simulations and the replicated data strategy: I. The computation of atomic forces. *Comp. Phys. Comm.*, 79:52–62, 1994.

[33] A. Windemuth. Advanced algorithms for molecular dynamics simulations: The program PMD. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry*, page 151. American Chemical Society, 1995.

[34] W. S. Young and C. L. Brooks, III. Dynamic load balancing algorithms for replicated data molecular dynamics, 1995.

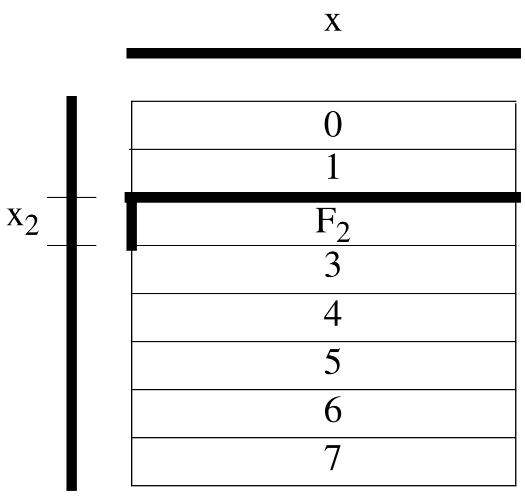

Figure 1: *The division of the force matrix among 8 processors in a replicated–data algorithm. Processor 2 is assigned N/P rows of the matrix and the corresponding x_2 piece of the position vector. In addition, it must know the entire position vector x (shown spanning the columns) to compute the non–bonded forces in F_2.*

(1)	**Compute** non–bonded forces in F_z, doubly summing results into local copy of f	$\frac{N}{P}$
(2)	**Compute** $1/P$ fraction of bonded forces, summing results into local copy of f	$\frac{N}{P}$
(3)	**Fold** f across all processors, result is f_z	N
(4)	**Update** atom positions in x_z using f_z	$\frac{N}{P}$
(5)	**Expand** x_z among all processors, result is x	N

Figure 2: *Single timestep of the replicated–data algorithm for processor P_z.*

(1)	**Compute** non–bonded forces in D_z, summing results into f_z and g_z	$\frac{N}{P}$
(2)	**Compute** bonded forces in D_z, summing results into f_z and g_z	$\frac{N}{P}$
(3)	**Share** g_z with neighboring processors, summing received forces into my f_z	Δ
(4)	**Update** atom positions in x_z using f_z	$\frac{N}{P}$
(5)	**Share** x_z with neighboring processors, using received positions to update y_z	Δ
(6)	**Move** atoms to new processors as necessary	Δ

Figure 3: *Single timestep of the spatial–decomposition algorithm for processor P_z.*

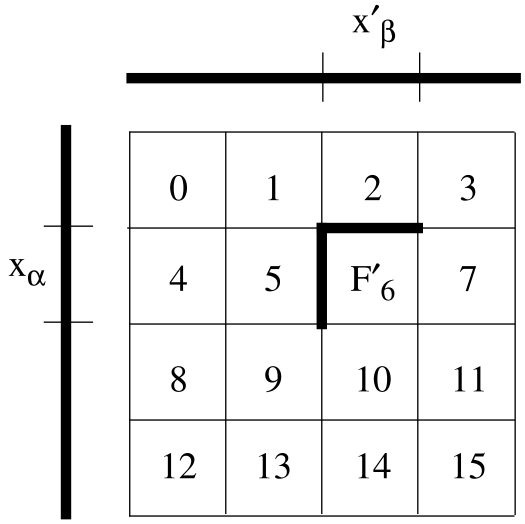

Figure 4: *The division of the permuted force matrix F' among 16 processors in the force–decomposition algorithm. Processor P_6 is assigned a sub–block ${F'}_6$ of size $N/\sqrt{P}$ by $N/\sqrt{P}$. To compute the non–bonded forces in ${F'}_6$ it must know the corresponding $N/\sqrt{P}$-length pieces x_α and ${x'}_\beta$ of the position vector x and permuted position vector x'.*

(1) **Compute** non–bonded forces in F'_z,	
storing results in f_α and f'_β	$\frac{N}{P}$
(2) **Compute** $1/P$ fraction of bonded forces,	
storing results in f_α and f'_β	$\frac{N}{P}$
(3a) **Fold** f_α within row α, result is f_z	$\frac{N}{\sqrt{P}}$
(3b) **Fold** f'_β within column β, result is f'_z	$\frac{N}{\sqrt{P}}$
(3c) **Subtract** f'_z from f_z, result is total f_z	$\frac{N}{P}$
(4) **Update** atom positions in x_z using f_z	$\frac{N}{P}$
(5a) **Expand** x_z within row α, result is x_α	$\frac{N}{\sqrt{P}}$
(5b) **Expand** x_z within column β, result is x'_β	$\frac{N}{\sqrt{P}}$

Figure 5: *Single timestep of the force–decomposition algorithm for processor P_z.*

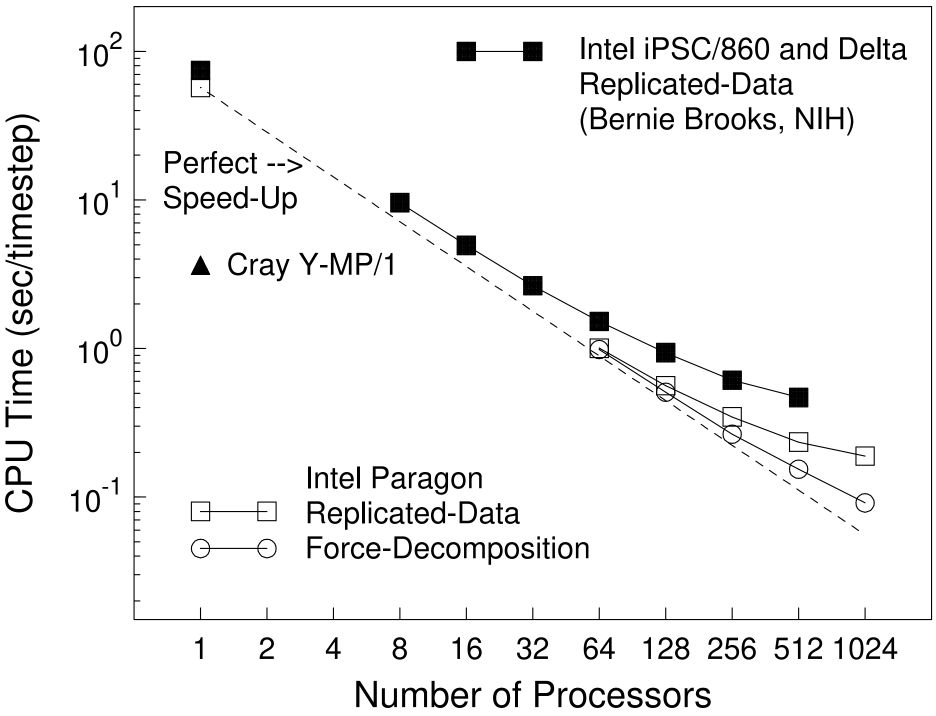

Figure 6: *CPU timings (seconds/timestep) for the replicated–data and force–decomposition algorithms on different numbers of processors for a 14026–atom myoglobin benchmark. Timings for the solid symbols are from reference [7].*

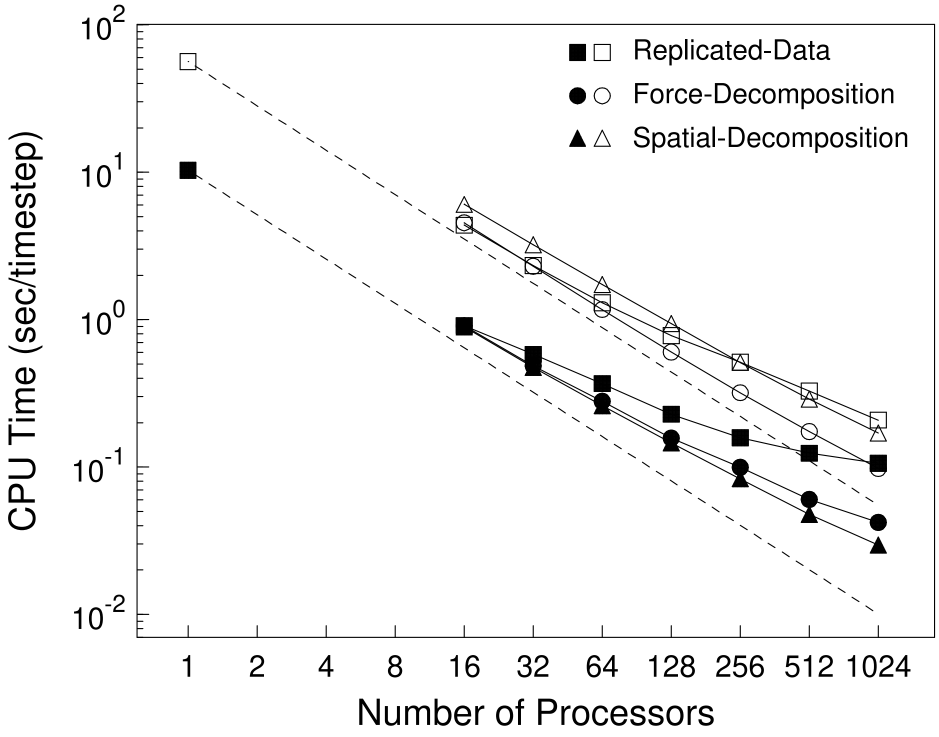

Figure 7: *CPU timings (seconds/timestep) for replicated–data, force–, and spatial–decomposition algorithms for a 10976–atom Lennard Jones benchmark on different numbers of nCUBE 2 processors. The solid symbols are for a 2.5σ cutoff; the open symbols are for a 5.0σ cutoff.*