Parallel Many–Body Simulations Without All–to–All Communication

Bruce Hendrickson and Steve Plimpton
Sandia National Laboratories
Albuquerque, NM 87185

This work was 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 U.S. Department of Energy under contract No. DE–AC04–76DP00789.

Proposed Running Head: Parallel Many–Body Simulations

Contact Author:
Bruce Hendrickson
Dept. 1422, MS 1110
Sandia National Labs
Albuquerque, NM 87185-1110

Abstract

Simulations of interacting particles are common in science and engineering, appearing in such diverse disciplines as astrophysics, fluid dynamics, molecular physics, and materials science. These simulations are often computationally intensive and so natural candidates for massively parallel computing. Many–body simulations that directly compute interactions between pairs of particles, be they short–range or long–range interactions, have been parallelized in several standard ways. The simplest approaches require all–to–all communication, an expensive communication step. The fastest methods assign a group of nearby particles to a processor, which can lead to load imbalance and be difficult to implement efficiently. We present a new approach, suitable for direct simulations, that avoids all–to–all communication without requiring any geometric clustering. For some computations we find the new method to be the fastest parallel algorithm available; we demonstrate its utility in several parallel molecular dynamics simulations.

Key words. Many–body problem, parallel computation, hypercube.

AMS(MOS) subject classification. 68Q22, 70–08, 70F10

1 Introduction

Simulations of interacting particles, also called many–body or N–body calculations, are common in science and engineering, occurring in such diverse settings as galaxy simulations, molecular dynamics calculations in solid–state and biological physics, and vortex methods in fluid dynamics. A particle in the simulation may represent a cluster of stars, an atom or molecule, or a fluid vortex. These applications share a common computational kernel in which the force acting on each particle is computed as a sum of interactions with some or all of the other particles. Simulations that compute each interaction explicitly use what are known as /em direct methods. Conversely, if the simulation approximates some (usually distant) interactions, it employs *approximate* methods. In either case, realistic simulations often involve large numbers of particles being followed for many timesteps. The consequent computational requirements, combined with the fact that force calculations for different particles can be done independently, make many–body simulations natural candidates for implementation on massively parallel machines.

In this paper, we present a new parallel algorithm for performing many–body computations. It is a direct method which has a different communication complexity than existing direct methods. We call our technique a *force–decomposition* algorithm because, unlike techniques that divide the particles or partition the simulation domain among the processors, we allocate an equal portion of the inter-particle force computations to each processor. This approach was motivated by the block–decomposition and torus–wrap methods now commonly used in parallel linear algebra algorithms [1, 4, 13], and the resulting algorithm has a close analogue in parallel matrix–vector multiplication [12].

In the next section we classify the different kinds of many–body simulations, indicating the kinds of problems for which the force–decomposition algorithm is particularly well–suited. In §3 the algorithm and its scaling properties are detailed, and in §4 various refinements to the basic approach are presented for specific parallel architectures and differing types of many–body problems. In §5, we demonstrate the utility of the algorithm in parallel molecular dynamics simulations, followed by conclusions in §6.

2 Many–body simulations

Many–body simulations can be characterized by the range of the forces being modeled. If the forces are long–range, like gravitational or Coulombic forces, then each particle is affected by all others in the simulation. If the forces are short–range, then each particle is only influenced at each timestep by a limited number of neighboring particles. Forces may be short–range by construction (e.g. van der Waals energies which fall off as the inverse 6th power of distance between two particles) or because longer–range forces are truncated at a cutoff distance to reduce the computational effort in a particular simulation.

As indicated in the introduction, many–body simulations can also be classified by whether they use direct or approximate methods to compute forces. In a long–range force simulation, the computational effort required to compute all the pairwise interactions directly is proportional to N^2, where N is the number of particles. In practice this makes long–range direct simulations unfeasible for large values of N. Various approximate methods have been devised that reduce this computational effort, including particle–mesh algorithms [14] which scale as $f(M)N$ where M is the number of mesh points, hierarchical algorithms [2] which scale as $N\log(N)$, and fast multipole methods [9] which scale as N. For long–range force systems, these approximate methods are the fastest choice for large N, though they are much more complex to implement than direct methods, particularly in parallel. Because of this complexity, approximate methods are typically not faster than direct methods until N reaches a certain threshold value which can be large. In parallel, the performance of approximate methods can suffer further from the fact that the work load can be difficult to balance among processors when the particle density is spatially and/or temporally non–uniform.

For short–range forces systems, the computational effort scales linearly with N, since every particle interacts with only a limited number of nearby particles (assuming no density singularities). For this reason, short–range simulations always use direct methods and compute interactions explicitly. There are two basic methods researchers have used to parallelize such computations. The simplest method is by a *particle–decomposition* of the workload [20]. Each of the P processors is assigned a fixed set of N/P particles and computes all of the forces acting on them for the duration of the simulation. To accomplish this, each processor must know the locations of potentially all the other particles, since the set of nearby particles changes rapidly as the simulation progresses. This requires *all–to–all* communication where each processor shares its updated particle coordinates with all the other processors at every timestep. Various techniques for performing this communication operation on parallel machines have been developed [1, 3, 24, 25], but the fastest can scale no better than $N/\log(P)$. If, as is commonly the case, only single port communication is available or the number of architectural neighbors of each processor is bounded, this is reduced to N, independent of P.

A second method for parallelizing short–range direct simulations is to exploit the geometric locality of the forces by dividing the computational domain into P pieces, one for each processor [20]. Using such a *spatial–decomposition* approach, each processor computes only the forces on particles in its sub–domain. Assuming an equal number of particles per processor, the computation scales as N/P. Communication of particle information is only necessary among processors owning nearby sub-domains. When the cutoff distance is much smaller than each processor's sub–domain this communication cost scales roughly as the surface–to–volume ratio of the sub–domain, namely $(N/P)^{(d-1)/d}$ where d is the geometric dimensionality of the problem.

Because of their scalability, spatial–decomposition methods are the optimal choice for parallelization of short–range direct simulations of large systems. In practice, however, spatial–decomposition methods can suffer from the same difficulties as do the approximate methods in the long–range case. They can be difficult to implement efficiently if, for example, the forces involved are not limited to simple pair–wise interactions (e.g., three–body and four–body forces in the simulation of molecular bonds). And they can be difficult to dynamically load balance if particles are moving rapidly, are non–uniformly dense, or do not fill a simple geometric volume. In addition, if many processors are used so that the sub–domain diameter is not large relative to the extent of the short–range force, then the volume of information that must be communicated across sub–domain boundaries can be large.

For these reasons, particle–decomposition methods can be a faster method for parallelizing short–range direct simulations, at least until some threshold value of N is reached [20]. However, since the communication cost of the all–to–all communication step scales at best as $N/\log(P)$, after some point, adding processors to a fixed–size problem does little to speed–up the simulation. An example of this is in the biological molecular dynamics (MD) community where many–body calculations are used to atomistically simulate bonded molecular systems such as polymers and proteins. Several recent parallel implementations of state–of–the–art MD codes [6, 7, 11, 18, 23] have all used some kind of particle–decomposition technique because of the limitations of spatial–decomposition methods discussed above. Unfortunately, these parallel implementations all exhibit poor scaling when P becomes large due to the cost of the all–to–all communication step.

In this paper, we present a new parallel method for short–range direct simulations which we call a *force–decomposition* algorithm. By dividing the work among processors in a new way, we obtain an algorithm that avoids the all–to–all communication bottleneck of particle–decomposition techniques, without requiring geometric clustering or incurring the load imbalance problems associated with spatial–decomposition methods. The communication requirements of the new algorithm scale as $N/\sqrt{P}$, independent of the nature of the forces or the geometric complexity of the domain, without requiring any redundant computation.

The new method is a good replacement for particle–decomposition techniques in any short–range direct many–body simulation in which the communication cost is significant. In these cases, it allows many more processors to be used effectively on a problem. The new method is also a good alternative to spatial–decomposition techniques when communication costs are high or load balancing is difficult. Aside from the advantage of simplicity, if these complexities seriously impact the spatial–decomposition algorithm's performance, the force–decomposition approach can in fact be faster, despite its non–optimal scaling. We return to this point in §5.

The force–decomposition algorithm we propose can also be used in long–range direct simulations. If a direct method is used instead of an approximate method in a long–range simulation (e.g. for a simulation of moderate–sized N), it is typically parallelized by some form of particle–decomposition algorithm since there is no advantage to a spatial–decomposition approach for direct computation of long–range forces. Our new algorithm will reduce the particle–decomposition method's communication cost in such a simulation by a $\sqrt{P}$ factor as in the short–range case. However, since the N^2 computational cost in long–range direct simulations is so high, the communication cost may not be a significant concern. For this reason, it is the short–range class of problems to which we think our new algorithm is most applicable.

3 The force–decomposition algorithm

We begin by considering a generic N–body simulation where force interactions between pairs of particles must be computed at every timestep. We define a position vector x of length N, whose elements denote the location of a particle. These elements consists of d values for a simulation in a d–dimensional geometry. We will consider the set of pairwise forces as an $N \times N$ array F, in which entry F_{ij} denotes the force on particle i due to particle j. If the forces are long–range then F will be dense; if they are short–range it will be sparse. Because of Newton's third law $F_{ij} = -F_{ji}$. The total force on particle i can be expressed as $\sum_j F_{ij}$. This expression is similar to the computation required in forming the product of a matrix with a vector, but there are several important differences. First, when repeatedly forming matrix–vector products the matrix usually stays the same, while in many—body calculations the motion of the particles continually change the values of the force array. This is particularly important in the short–range case where the zero/nonzero structure of the force array changes over time. Second, the skew–symmetric structure imposed by Newton's third law allows for some additional efficiency, as we will see below. And third, knowledge of the locations of two particles is necessary to compute an element of the force array, while only one element of the vector is needed in matrix–vector multiplication. Many–body calculations thus require some additional communication. Despite these differences, an algorithm for matrix–vector multiplication is described in [12] which has a similar flavor to the many–body algorithm we present below.

In particle–decomposition methods, a processor computes all the forces on a subset of particles, which corresponds to assigning entire rows of F to a single processor. We propose instead to assign each processor a square block of F to form a *force–decomposition* as depicted in Fig. 1. We assume for ease of exposition that P is an even power of 2 and that N is a multiple of P, although it is fairly straightforward to relax these constraints. The block owned by each processor is thus square and of size $(N/\sqrt{P}) \times (N/\sqrt{P})$. We will use the Greek subscripts α and β to index the row and column blocks of F running from 0 to $\sqrt{P}-1$. A block of F is denoted as $F_{\alpha\beta}$, and the processor owning this block is $P_{\alpha\beta}$. We note that α and β also index sub–vectors of x of length $N/\sqrt{P}$. We will assume for now that the force between two particles depends only upon their locations and some invariant values like charge or mass, although this assumption will be removed in §4.4. In this case, to compute the array elements in its block, a processor must know the x_α and x_β pieces of the position vector x. We will denote the vector of length $N/\sqrt{P}$ consisting of the net force on particles in the α block due to those in the β block as $f_{\alpha\beta}$, and note that it is computed by processor $P_{\alpha\beta}$.

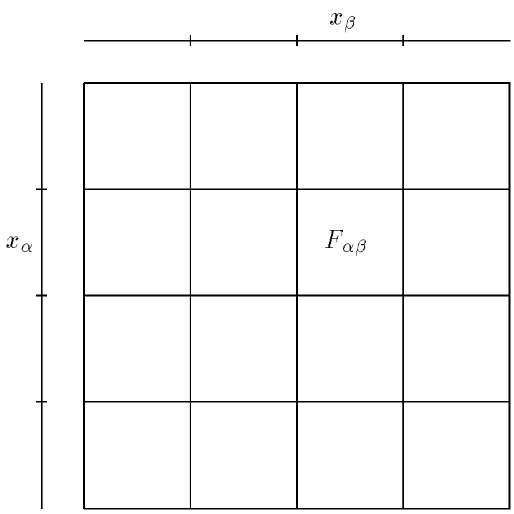

Figure 1. Decomposing the force array among 16 processors.

In addition to generating its block of array elements and computing $f_{\alpha\beta}$, each processor will be responsible for updating the positions of N/P particles, as in the particle–decomposition algorithm. These particles are a sub–vector of x_α; that is, the $\sqrt{P}$ processors in row α divide x_α among them, so each is responsible for a contiguous piece of length N/P. Numbering these pieces with the column index β of the processor, we denote each processor's piece as x_α^β. Which of these pieces is owned by which processor depends on the ordering of the loops in the communication operations discussed below. We denote the vector of *total* forces acting on the particles in x_α^β by f_α^β, elements of which consist of the sum of corresponding values of the $f_{\alpha\beta}$ vectors owned by the processors in a row of the array.

In the force–decomposition algorithm, we will make repeated use of two well–known communication primitives. The first of these is a form of all–to–all communication among all the processors in the same row or column of F. Each processor in a row will share its N/P positions with all the $\sqrt{P}$ processors in the row, so that at the end of the operation, each processor in row α will know all $N/\sqrt{P}$ positions in x_α. This contrasts with the particle–decomposition algorithm, in which all P processors end up with all N positions.

All–to–all communication has been studied extensively and a number of algorithms proposed for different parallel architectures [3, 16, 24, 25]. We use a well known algorithm that runs well on most parallel machines, scales optimally, and requires a minimum number of messages be sent and received. This *expand* algorithm is outlined below in Fig. 2. At each step of the expand operation, a processor sends the piece of the vector it currently knows, denoted by y, to another processor in its row and receives a new piece z. It concatenates z with y, denoted by "|" in the figure, doubling the size of the piece it knows, so in a logarithmic number of steps each processor accumulates the entire vector. The total volume of information sent and received by each processor is $N/\sqrt{P} - N/P$. The loop in Fig. 2 selects neighboring processors in increasing bit order, but on a hypercube any order is equivalent. However, on a mesh it is advantageous to send the most voluminous messages the shortest distance, and the ordering in Fig. 2 accomplishes this.

```
y := x_α^β
For k = log_2(√P) − 1, ..., 0
    P' := P_αβ with k^th bit of β flipped
    Send y to processor P'
    Receive z from processor P'
    If bit k of β is 0 Then
        y := y|z
    Else
        y := z|y
x_α := y
```

Figure 2. The expand operation within a row for processor $P_{\alpha\beta}$.

The second communication primitive is essentially the inverse of the expand operation. Processor $P_{\alpha\beta}$ has computed $f_{\alpha\beta}$, its contribution to the forces on all $N/\sqrt{P}$ particles in block α, and these contributions need to be summed over all the processors comprising row block α to generate the total forces. After the summation, processor $P_{\alpha\beta}$ needs to know the summed results for only the N/P particles it is responsible for updating, namely those in x_α^β. These N/P values are the sum of the corresponding elements across all the processors in row block α. This can be accomplished with a *fold* operation [8], as outlined in Fig. 3.

The communication pattern of the fold operation is precisely the reverse of that in the expand operation. At each fold step, a processor sends half of the y vector (y^1 or y^2) to another processor in its row and receives a new half–piece z. It sums z element by element with the half–piece it did not send to create a new y for the next iteration. Like the expand operation, this algorithm is completed in a logarithmic number of steps with y halving in size at every step (instead of doubling as in the expand). The total volume of messages sent and received by each processor is again $N/\sqrt{P} - N/P$ and each processor performs $N/\sqrt{P} - N/P$ floating point operations. We note that both the expand and fold operations require only nearest neighbor communication on hypercubes, and they can also be implemented efficiently on meshes [25].

$y := f_{\alpha\beta}$
For $k = 0, \ldots, \log_2(\sqrt{p}) - 1$
    $y^1 :=$ top half of y vector
    $y^2 :=$ bottom half of y vector
    $P' := P_{\alpha\beta}$ with k^{th} bit of β flipped
    If bit k of β is 0 **Then**
        Send y^2 to processor P'
        Receive z from processor P'
        $y := y^1 + z$
    Else
        Send y^1 to processor P'
        Receive z from processor P'
        $y := y^2 + z$
$f_\alpha^\beta := y$

Figure 3. The fold operation within a row for processor $P_{\alpha\beta}$.

Using the expand and fold primitives as building blocks, we can now describe our force–decomposition algorithm for many–body simulations. Fig. 4 outlines a single timestep of the algorithm for processor $P_{\alpha\beta}$, assuming each processor owns current copies of x_α and x_β at the beginning of the timestep. In step (1) the processor computes all the pairwise forces within its $F_{\alpha\beta}$ block of the force array, summing these values into the $f_{\alpha\beta}$ vector so that there is no need to store $F_{\alpha\beta}$ explicitly. In step (2), a fold operation is performed within rows of processors so that processor $P_{\alpha\beta}$ obtains the total forces on its particles f_α^β. These total forces are used to update the positions of the corresponding N/P particles in step (3). To prepare for the next timestep, these positions must be made known to all processors that share a row or column with $P_{\alpha\beta}$, which is accomplished in steps (4)–(6). First, the positions are shared across each row with an expand in step (4). Then in step (5) each processor exchanges its updated N/P positions with $P_{\beta\alpha}$, the processor in the transpose position in the force array. Finally, vector x_β is acquired via a column expand in step (6). The processors now have all the information they need to begin the next timestep. We will refer to this algorithm as **A1**.

3.1 Exploiting Newton's third law

Algorithm **A1** fails to exploit the skew–symmetric nature of the force array, so each pairwise force calculation is done twice, once for particle i and once for particle j. An algorithm to avoid this duplication can be devised by constructing a modified force array G, defined as follows.

$$G_{ij} = \begin{cases} F_{ij} & \text{if } i+j \text{ even and } i > j, \\ F_{ij} & \text{if } i+j \text{ odd and } i < j, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

(1)	**Compute** $F_{\alpha\beta}$ elements storing results in $f_{\alpha\beta}$
(2)	**Fold** $f_{\alpha\beta}$ within row, yielding f_α^β
(3)	**Update** particle positions in x_α^β using f_α^β
(4)	**Expand** x_α^β within row, result is x_α
(5a)	**Send** x_α^β to $P_{\beta\alpha}$
(5b)	**Receive** x_β^α from $P_{\beta\alpha}$
(6)	**Expand** x_β^α within column, yielding x_β

Figure 4. Single timestep of force–decomposition algorithm **A1** for processor $P_{\alpha\beta}$.

If we imagine the array F to be colored like a checkerboard, G is identical to F except that red squares above the diagonal are set to zero, as are black squares below the diagonal. In this way, each pairwise force is only computed once. Other constructions that achieve this property are possible; this particular one was chosen to preserve load balance, an issue we consider further in §4.1. We can adapt algorithm **A1** to take advantage of Newton's third law by observing that the total force on particle i is the sum of the elements in row i of G minus the sum of the elements in column i of G. This modified algorithm **A2** is depicted in Fig. 5. We denote by $\overline{g}_{\alpha\beta}$ the vector of length $N/\sqrt{P}$ that consists of the sum within rows of all the elements of block $G_{\alpha\beta}$, and by $\underline{g}_{\alpha\beta}$ the vector of length $N/\sqrt{P}$ that consists of the sum within columns of all the elements of block $G_{\alpha\beta}$. As before, we also need the vectors $\overline{g}_\alpha^\beta$ and $\underline{g}_\alpha^\beta$ of length N/P that consist of elements of $\overline{g}_{\alpha\beta}$ and $\underline{g}_{\alpha\beta}$ summed within rows and columns of G respectively.

(1)	**Compute** $G_{\alpha\beta}$, summing results into $\overline{g}_{\alpha\beta}$ and $\underline{g}_{\alpha\beta}$
(2a)	**Fold** $\underline{g}_{\alpha\beta}$ within column, yielding $\underline{g}_\beta^\alpha$
(2b)	**Send** $\underline{g}_\beta^\alpha$ to $P_{\beta\alpha}$
(2c)	**Receive** $\underline{g}_\alpha^\beta$ from $P_{\beta\alpha}$
(2d)	**Fold** $\overline{g}_{\alpha\beta}$ within row, yielding $\overline{g}_\alpha^\beta$
(2e)	$f_\alpha^\beta := \overline{g}_\alpha^\beta - \underline{g}_\alpha^\beta$
(3)	**Update** particle positions in x_α^β using f_α^β
(4)	**Expand** x_α^β within row, result is x_α
(5a)	**Send** x_α^β to $P_{\beta\alpha}$
(5b)	**Receive** x_β^α from $P_{\beta\alpha}$
(6)	**Expand** x_β^α within column, result is x_β

Figure 5. Single timestep of force–decomposition algorithm **A2** for processor $P_{\alpha\beta}$. This algorithm exploits Newton's third law.

We again assume that processor $P_{\alpha\beta}$ knows x_α and x_β at the beginning of the timestep. The first step of **A2** is similar to the first step of **A1**, except that the elements of G are summed within both rows and columns, yielding $\overline{g}_{\alpha\beta}$ and $\underline{g}_{\alpha\beta}$. Steps (2a)–(2e) replace step (2) of **A1**. First, in step (2a), the partial forces $\underline{g}_{\alpha\beta}$ are folded within each column resulting in $\underline{g}_{\beta}^{\alpha}$ being known by $P_{\alpha\beta}$. In steps (2b) and (2c), each processor exchanges this component with the processor in the transpose position, storing the received vector as $\underline{g}_{\alpha}^{\beta}$. A fold of $\overline{g}_{\alpha\beta}$ is then performed across the rows in step (2d) to generate $\overline{g}_{\alpha}^{\beta}$. Now in step (2e) the total forces on the particles owned by processor $P_{\alpha\beta}$ are computed by subtracting $\underline{g}_{\alpha}^{\beta}$ from $\overline{g}_{\alpha}^{\beta}$. Since processor $P_{\alpha\beta}$ now knows f_{α}^{β} it can update the locations of its particles, and steps (3)–(6) are identical to those of **A1**.

3.2 Scalability

The communication stages of the algorithms **A1** and **A2** occur in steps (2) and (4)–(6). Algorithm **A1** requires one fold, two expands and one exchange with the transpose processor. These operations require each processor to send and receive $\frac{3}{2}\log_2(P)+1$ messages. The total volume of fold and expand messages sent and received by each processor is $3(N/\sqrt{P}-N/P)$, and the volume of the transpose message is N/P. Algorithm **A2** includes two fold, two expand and two transpose communications, implying a total of $2\log_2(P)+2$ sends and receives for each processor. The corresponding fold and expand volume for each processor is $4(N/\sqrt{P}-N/P)$, and that for the transpose communication is $2N/P$.

The fold and expand operations require only nearest neighbor communication on hypercubes, and they can also be efficiently implemented on mesh architectures [25]. The transpose operation may involve communication between distant pairs of processors, but this is unlikely to affect the overall scaling of the algorithm. Since the message volume in the transpose operation is about $\sqrt{P}$ smaller than that for the folds and expands, the transposes can tolerate a delay of $O(\sqrt{P})$ before they dominate the communication time. In §4.2 we will address this issue further, and show how to implement a contention–free transpose on hypercubes.

Assuming the transmission time dominates the $\log(P)$ message startup time, the overall communication cost of the force–decomposition algorithms scales as $N/\sqrt{P}$, which compares favorably to the $N/\log(P)$ communication cost required for particle–decomposition methods.

The computation portion of the algorithms occurs in steps (1)–(3). We will charge the additions in the fold operation to the communication cost, in which case all computations are in steps (1) and (3). The updates of particle locations in step (3) are perfectly balanced across processors, so all potential load imbalance occurs in step (1), where the load on processor $P_{\alpha\beta}$ is proportional to the number of nonzero entries in $F_{\alpha\beta}$ (or $G_{\alpha\beta}$). If forces are long–range, then $F_{\alpha\beta}$ will be dense, and the load will be perfectly balanced. If the forces are short–range, some load imbalance is possible; we propose a method for minimizing it in §4.1.

Algorithm **A2** exploits the skew–symmetry in the force matrix, computing a minimal number of pairwise interactions, so it requires the minimal number of floating point operations. By contrast, step (1) of **A1** involves computing each pairwise interaction twice, so it is not optimally efficient. For this reason, we have found in practice that algorithm **A2** is generally faster than **A1**, despite the extra communication in **A2**. Assuming the load is balanced, the computational time for both algorithms is $O(N^2/P)$ for long–range forces and $O(N/P)$ for short–range forces.

Algorithm **A1** requires each processor to store two position vectors and one force vector of length $N/\sqrt{P}$, while algorithm **A2** needs an additional force vector. An additional work vector of half this length is probably needed for the fold operation, depending on precisely how the parallel machine handles communication. With current generation parallel machines, very large problems can be run within these memory constraints.

4 Algorithmic details and special cases

In this section we discuss refinements to the algorithms presented in §3 to improve their performance. In order, the issues discussed in each subsection are reducing load imbalance, implementing the transpose communication efficiently, overlapping computation with communication, and extending the force–decomposition algorithms to handle non–pairwise forces.

4.1 Short–range forces and load imbalance

If the forces being modeled are long–range then the force array will be dense, and load balance is assured. However, in the short–range case the array will be sparse. If the density of nonzeros is nonuniform, some processors may have more interactions to compute than others, resulting in load imbalance and longer run times. For example, in the common situation where particles are numbered according to their geometric locations, then the diagonal blocks of the force array will have a larger number of nonzeros than the off-diagonal blocks.

This problem can be avoided by randomly permuting the ordering of the particles at the beginning of the simulation. This produces a random sparsity pattern in F or G so that each processor has roughly equal amounts work to do. A random permutation has the advantage that the balance is likely to persist as the particles move during the simulation. Since it need only be generated once, the cost of the permutation is not critical; we have implemented a simple random permutation generator due to Knuth [17] for the molecular dynamics simulations discussed in §5. A detailed analysis of the load balance implications of random matrix permutations for the related problem of multiplying a vector by a sparse matrix is presented in [19].

4.2 Transposition on parallel computers

The expand and fold primitives used in the force–decomposition algorithm are most efficient on a parallel computer if rows and columns of the force array can be mapped to subsets of processors that allow for fast communication. On a hypercube a natural subset is a subcube, while on a 2–D mesh rows, columns or submeshes are possible. Unfortunately, such a mapping can make the transpose operation inefficient since it requires communication between processors that are architecturally distant. Even though all modern parallel computers use *cut–through routing* so that a single message can be transmitted between non–adjacent processors at nearly the same speed as between adjacent ones, if multiple messages are simultaneously trying to use the same wire, one of them will be delayed. The potential occurrence of such *congestion* depends on the message routing algorithm employed by the machine.

The routing strategy used in a parallel machine is often rigidly determined by the operating system. On a hypercube the scheme for routing a message is usually to compare the bit addresses of the sending and receiving processors and flip the bits in a fixed order until the two addresses agree. On the nCUBE 2 and Intel iPSC/860 hypercubes the order of comparisons is from lowest bit to highest, a procedure known as *dimension order routing*. Thus a message from processor 1001 to processor 0100 will route from 1001 to 1000 to 1100 to 0100. Now consider what occurs during the transpose if the processors in the hypercube are assigned to the force array in a calendar ordering as in the left diagram of Fig. 6, where the low–order bits of the processor address are its column number and the high–order are its row number. For the transpose operation each of the processors in row α will pair with a processor in column α to exchange messages. But with the routing scheme just described, all of column bits will be flipped first, so all these messages will route through the diagonal processor $P_{\alpha\alpha}$, producing congestion on its wires. The same congestion occurs on mesh machines which typically route a message first along a row (or column) of processors followed by routing within a column (or row) to arrive at the destination processor.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	1	4	5
2	3	6	7
8	9	12	13
10	13	14	15

Figure 6. Left: Calendar mapping of processors to force array.
Right: Interleaved mapping to eliminate transpose contention.

We know of no solution to this congestion problem on a mesh, but using a calendar ordering, the maximum congestion that can occur during the transpose operation is a factor of $\sqrt{P}$, the number of processors in a row or column. Anticipating this, algorithms **A1** and **A2** exchange a minimum amount of data in their transpose, so as discussed in §3.2, even with a worst case $\sqrt{P}$ slow–down, the overall scaling of the communication operations is not affected.

On a hypercube, a different mapping of processors to the force array can avoid transpose congestion altogether. Consider a d–dimensional hypercube where the address of each processor is a d–bit string, where for simplicity we assume that d is even. The row block number α of the force array is a $d/2$–bit string as is the column block number β. For fast fold and expand operations, we require that the processors in each row and column form a subcube, which is assured if any set of $d/2$ bits in the d–bit processor address encode the row number and the other $d/2$ bits encode the column number. The ordering in the left diagram of Fig. 6 has this property, but as shown above it produces congestion.

Now consider a mapping where the bits of the row and column indices of the force array are interleaved in the processor address. For a 64–processor hypercube (with 3–bit row and column addresses for the 8×8 blocks of the force array) this means the 6–bit processor address could look like $r_2c_2r_1c_1r_0c_0$ where the three bits $r_2r_1r_0$ encode the row index and $c_2c_1c_0$ encodes the column index. An assignment of processors to the force array using this mapping for a 16–node hypercube is shown in the right diagram of Fig. 6.

Note that in this mapping each row and column of the force array is still a sub–cube of the hypercube, so that the expand and fold operations can be performed optimally. However, the transpose operation is now contention–free as demonstrated by the following theorem. Although the proof assumes a routing scheme where bits are flipped in order from lowest to highest, a similar contention–free mapping is possible for any fixed routing scheme as long as row and column bits are forced to change alternately. This result was discovered independently by Johnsson and Ho [15], and generalized by Boppana and Raghavendra [5].

Theorem 4.1 *Consider a hypercube using dimension order routing, and map processors to elements of an array in such a way that the bit–representations of a processor's row number and column number are interleaved in the processor's bit–address id. Then the wires used when each processor sends a message to the processor in the transpose location in the array are disjoint.*

Proof: Consider a processor P with bit–address $r_b c_b r_{b-1} c_{b-1} \cdots r_0 c_0$, where the row number is encoded with $r_b \cdots r_0$, and the column number with $c_b \cdots c_0$. The processor P^T in the transpose array location will have with bit–address $c_b r_b c_{b-1} r_{b-1} \cdots c_0 r_0$. Under dimension order routing, a message is transmitted in as many stages as there are bits, flipping bits in order from right to left to generate a sequence of intermediate patterns. After each stage, the message will have been routed to the intermediate processor denoted by the current intermediate bit pattern. The wires used in routing the message from P to P^T are those that connect two processors whose patterns occur consecutively in the sequence of intermediate patterns. After $2k$ stages, the intermediate processor will have the pattern $r_b c_b \cdots r_k c_k c_{k-1} r_{k-1} \cdots c_0 r_0$. The bits of this intermediate processor are a simple permutation of the original bits of P in which the lowest k pairs of bits have been swapped. Also, after $2k-1$ stages, the values in the bit positions $2k$ and $2k-1$ are equal.

Now consider another processor $P' \neq P$, and assume that the message being routed from P' to P'^T uses the same wire employed in step i of the transmission from P to P^T. Denote the two processors connected by this wire by P_1 and P_2. Since they differ in bit position i, P_1 and P_2 can only be encountered consecutively in the transition between stages $i-1$ and i of the routing algorithm. Either $i-1$ or i is even, so a simple permutation of pairs of bits of P must generate either P_1 or P_2; say P_*. Similarly, the same permutation applied to P' must also yield either P_1 or P_2; say P'_*. If $P_* = P'_*$ then $P = P'$ which is a contradiction. Otherwise, both P_1 and P_2 must appear after an odd number of stages in one of the routing sequences. If i is odd then bits i and $i+1$ of P must be equal, and if i is even then bits i and $i-1$ of P are equal. In either case, $P_1 = P_2$ which again implies the contradiction that $P = P'$. ∎

If a contention–free transpose can be implemented by this, or any other mapping on a particular machine, then a modified version of algorithms **A1** and **A2** is slightly faster. Each occurrence of an N/P–volume transpose followed by an expand or fold can be replaced by an $N/\sqrt{P}$–volume transpose which eliminates the need for the expand or fold. In particular, steps (5) and (6) of **A1** and **A2** can be replaced by:

(5a) **Send** x_α to $P_{\beta\alpha}$

(5b) **Receive** x_β from $P_{\beta\alpha}$.

A similar change is possible for steps (2a)–(2c) of **A2**. Although these modifications do not change the overall message volume, they reduce the number of messages and the corresponding startup costs.

4.3 Overlapping computation and communication

If a processor is able to both compute and communicate simultaneously, then algorithms **A1** and **A2** have the shortcoming that a processor has nothing to do after sending a message while waiting for its neighbor's message to arrive. This can be alleviated in the first fold operation by interspersing the computation of step (1) with the logarithmic loop of the fold. For instance, in **A1**, instead of computing all the elements of $f_{\alpha\beta}$ before beginning the fold operation, at each step in the fold loop $P_{\alpha\beta}$ can compute only those elements of $f_{\alpha\beta}$ that are about to be sent. The elements to be sent in the next step can be generated before issuing a receive for the current step, effectively reducing or eliminating the time spent waiting for the current step's message to arrive. In this way, the total run time is reduced on each pass through the fold loop by the minimum of the transmission time for the message and the time to compute the next set of elements of $f_{\alpha\beta}$.

4.4 Non–pairwise forces

The discussion in the preceding sections assumed that the forces involved pairs of particles and that they could be computed using only particle positions. In some simulations, computing forces requires additional information like velocities or vorticities which change at each timestep. Updated values for these quantities can be communicated as needed in a force–decomposition algorithm by mimicking the expand of position vectors in **A1** and **A2**. Quantities such as electric charge or mass that are needed to compute forces but do not change over time can be shared once at the beginning of the simulation and stored by each processor in vectors of size $N/\sqrt{P}$.

In addition, forces are sometimes computed in many–body simulations which require information about more than two particles. These are computable within the force–decomposition framework without extra communication if there is a processor that knows the positions within its x_α and/or x_β vectors of all the particles needed to compute a particular force. For example, a three–body force, like the angular force in a molecule, can be computed if there is a processor that knows the positions of all 3 particles. This is guaranteed if the particles are ordered (a pre–processing step) so that at least two of the particles are in the same sub–vector x_α. The third particle will then be in some x_β so that processor $P_{\alpha\beta}$ knows all three. As this simple example indicates, many–body force computations require additional care in the ordering of particles. Unfortunately, these restrictions can conflict with the desire to balance the computational load. We have addressed these issues in [22] for bonded molecular dynamics simulations where three– and four–body forces are used.

5 Results

In this section we demonstrate the utility of the force–decomposition algorithm **A2** in several short–range force molecular dynamics (MD) simulations. Where available, we compare it to particle– and spatial–decomposition algorithms on the same benchmark calculations. Implementation details for all three parallel algorithms as used in MD simulations and more specifics on the benchmarks are given elsewhere [20, 22]; here we highlight the performance of the force–decomposition algorithm. Our implementation of **A2** included a random ordering of the atoms as discussed in §4.1, but no effort to overlap communication with computation as in §4.3.

The first benchmark is a simulation of N atoms in three dimensions interacting via a Lennard–Jones potential. The density and temperature are chosen so that a liquid is modeled. The force interaction is truncated in range so that, on average, each atom interacts with about 55 others at every timestep; this is typical of cutoff lengths used in statistical and solid–state physics simulations. The benchmark has periodic boundary conditions and the particles fill a 3–D parallelepiped, which makes it particularly well suited for a spatial–decomposition algorithm. Neighbor lists are used in all three algorithms for efficient computation of the inter–particle forces. This is a data structure which minimizes the number of nearby particles that must be checked at every timestep to see if they are close enough to interact with a given particle. The computational cost of building the neighbor lists every few timesteps is included in the timing results presented.

The machines used in this study were a 1024–processor nCUBE 2 (MIMD hypercube with 4 Gbytes of memory) and a 1024–processor Intel Paragon (MIMD 2–D mesh with 16 Gbytes of memory). We believe our algorithms are appropriate for any parallel machine which supports a MIMD message–passing protocol.

In Fig. 7, the number of CPU seconds required per simulation timestep for the first benchmark problem is shown for all three parallel algorithms running on the 1024–processor nCUBE for several hundred timesteps. For comparison purposes, single processor Cray Y-MP timings are also shown for our implementation of the fastest serial algorithm in the literature [10]. The Cray algorithm vectorizes fully; these results are the fastest Y-MP/1 timings for this benchmark that have been reported. Fig. 7 shows the linear scaling of all four algorithms with large N, as expected for short–range force MD simulations. It also shows that the force–decomposition algorithm is faster than the particle–decomposition approach for all problem sizes. The spatial–decomposition algorithm is fastest once N reaches several thousands of atoms, but is more inefficient on this many processors for smaller sizes. We note that this benchmark is actually a best–case scenario for a spatial–decomposition algorithm because the particles are of uniform density and completely fill a rectangular 3–D volume which eliminates potential load imbalance. Notwithstanding, the threshold size where a spatial–decomposition method becomes the fastest has been increased from a few hundred atoms for the particle–decomposition case to several thousand. Even for very large N, the force–decomposition timings are within a factor of 3 of the spatial–decomposition results. The particle–decomposition code and Cray code were unable to run the largest problems due to memory limitations.

In Fig. 8, timing results are shown for the same benchmark problem where the number of atoms is kept fixed at 10976 while varying the number of processors on the nCUBE 2. Two sets of simulations were run with each of the three decomposition algorithms. The solid symbols represent calculations with the same cutoff used in Fig. 7, while the open symbols are for a longer cutoff encompassing around 400 neighbors. The latter case is more typical of a simulation, for example, of organic molecules whose individual atoms are charged and so require a longer cutoff to accurately capture longer–range Coulombic effects. For both sets of results one–processor timings and perfect speed–up lines are shown, representing the best performance a parallel algorithm could potentially achieve. For the shorter cutoff all three algorithms perform similarly on small numbers of processors. As P increases, the parallel efficiency of the particle–decomposition algorithm degrades markedly due to the $O(N)$ scaling of its all–to–all communication step. The loss of efficiency in the force–decomposition algorithm is not so severe due to its $\sqrt{P}$ factor reduction in communication cost. Although the spatial–decomposition algorithm outperforms it for all P, it is by only a small amount for this problem size.

The timing data represented by the open symbols tell a different story. The communication cost in the particle–

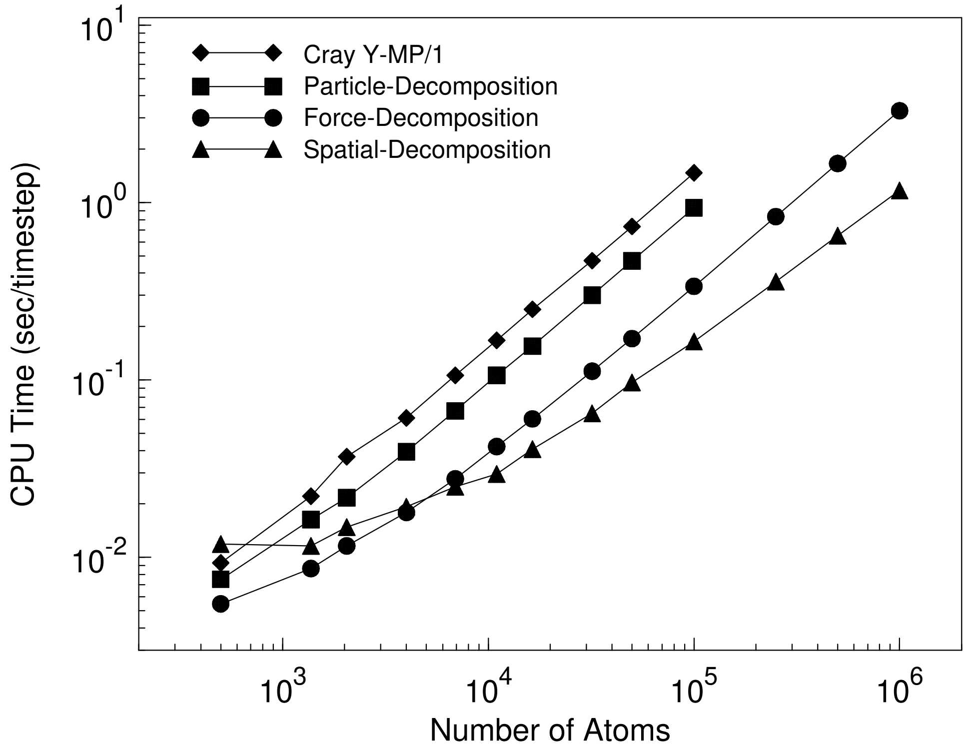

Figure 7. Performance of the three parallel algorithms on a 1024–processor nCUBE 2 for a molecular dynamics benchmark as a function of problem size. Single processor Cray Y–MP performance is also shown for a fully vectorized algorithm.

and force–decomposition algorithms is unaffected by the cutoff length. However the longer cutoff induces significant extra communication for the spatial–decomposition algorithm. Now the force–decomposition algorithm offers the best performance for all P from 16 to 1024. The fall off in its and the particle–decomposition algorithm's performance as P increases is now less dramatic as compared to the shorter–cutoff case since communication now requires a smaller fraction of the total run time.

If N were increased in the benchmark of Fig. 8, eventually spatial decomposition would become faster due to its superior large N scalability. However, in other MD simulations the performance of the spatial–decomposition algorithm could be further degraded by particle density non–uniformities. We illustrate this point with timing results from a different kind of MD calculation, using a parallel code we have written for simulating organic systems [21, 22]. The chief difference in such a code is that bonded forces within the topology of the molecules must now be computed in addition to non–bonded pairwise forces. In our code both a particle– and force–decomposition are implemented. Timing results for simulations of a 6750–atom ensemble of liquid–crystal molecules are shown in Fig. 9 on varying numbers

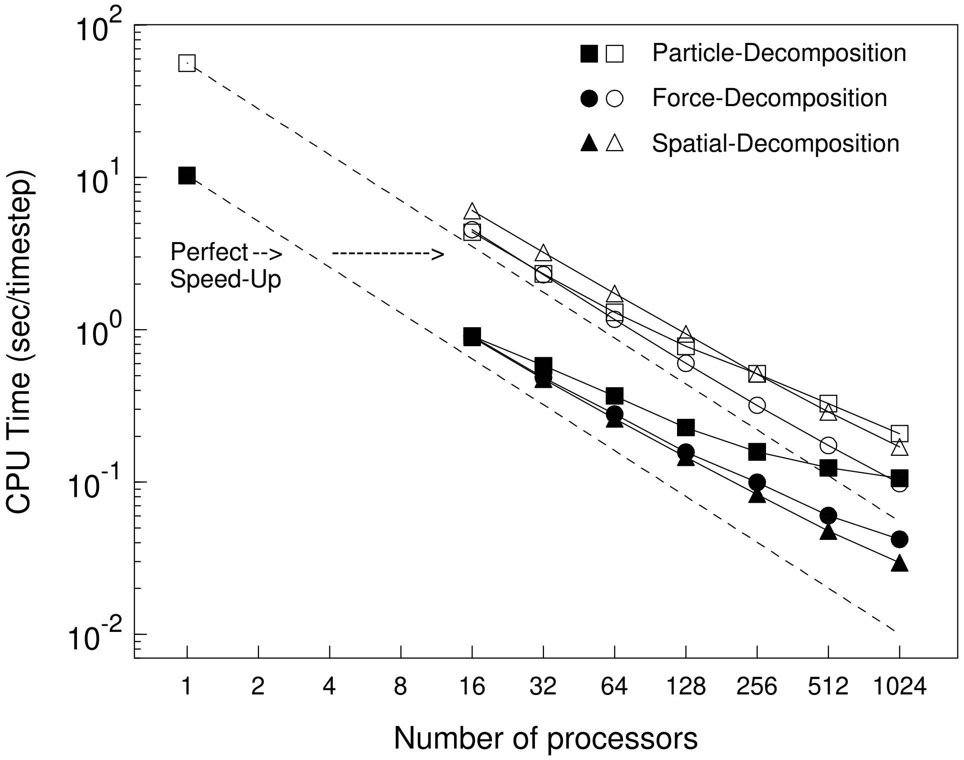

Figure 8. Performance of the three parallel algorithms on a molecular dynamics simulation of 10976 atoms for varying numbers of nCUBE 2 processors. The solid symbols are for a shorter cutoff distance; the open symbols are for a longer cutoff. Perfect speed–ups extrapolated from single–processor timings are shown by the dashed lines.

of processors of both the nCUBE 2 and Intel Paragon. On both machines the force–decomposition algorithm runs about 1.7 times faster on 256 processors and 3–3.5 times faster on 1024 processors than does the particle–decomposition algorithm. As mentioned in §2, the dramatic roll–off in the particle–decomposition timings is typical of the performance degradation seen in other parallel implementations of this kind of MD simulation [6, 7, 11, 18, 23] as P increases and the communication portion of the algorithm begins to dominate. The loss of efficiency in both algorithms is more pronounced on the Intel Paragon because of its 2–D mesh architecture.

As discussed in §2, systems like the liquid–crystal simulation presented here are difficult to parallelize with a spatial–decomposition approach. The simulation is run with a long force cutoff so that each atom interacts with hundreds of neighbors. And the atoms form a roughly spherical cluster in vacuum, typical of molecular problems that do not have a natural crystalline periodicity. Both of these characteristics would create serious efficiency problems for a spatial–decomposition algorithm running with 6 or 7 atoms per processor on a 1024–processor parallel machine. In fact, no spatial–decomposition timings are included in Fig. 9 because we do not have (nor to our knowledge has anyone written) a general–purpose parallel organic MD code using spatial–decomposition techniques that will run effectively on hundreds to thousands of processors for such a problem. It is a good example of a simulation for which we believe the force–decomposition algorithm is currently the fastest choice available. In our case, the new algorithm has made a significant difference in the speed at which we can perform several hundred thousand timestep simulations of the system to study the individual molecule's conformational properties.

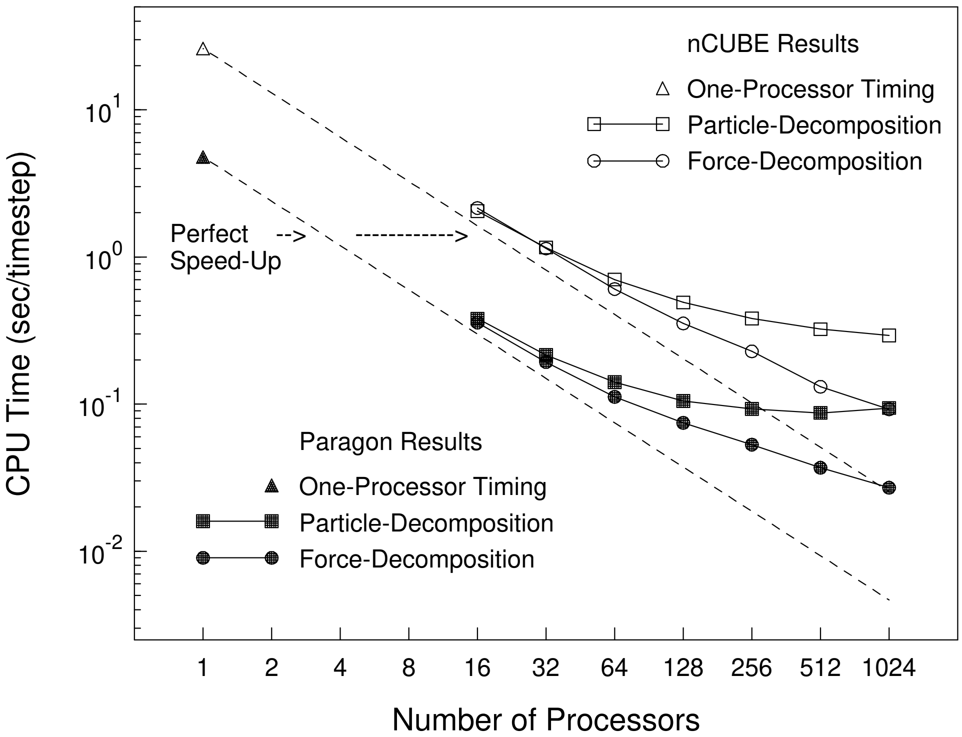

Figure 9. Performance of two parallel algorithms on two machines in a molecular dynamics simulation of a 6750–atom liquid–crystal system.

6 Conclusions

We have presented a parallel force–decomposition algorithm for many–body calculations that avoids the cost of all–to–all communication. It is most appropriate for short–range direct simulations though it could also be useful in long–range simulations that employ direct methods. The communication cost of our algorithm scales as $O(N/\sqrt{P})$ instead of the

$O(N/\log(P))$ required by particle–decomposition algorithms. In molecular dynamics benchmark calculations, the force–decomposition algorithm proved faster than a particle–decomposition scheme for all problem sizes considered. For large problems, a spatial–decomposition algorithm is faster, but for small and intermediate sized problems the new algorithm is best. The crossover size is a function of load–balance and cutoff length but can easily be in the tens–of–thousands of particles. Additionally, because the force–decomposition algorithm does not need to exploit any geometric structure present in the physical domain, it is much simpler to implement and load balance on a wider class of problems than spatial–decomposition methods.

The mapping of array elements to processors we describe for hypercubes is likely to be of independent interest. This mapping ensures that rows and columns of the array are owned by subcubes, and that a transpose operation can be performed without any message contention on machines with cut–through routing. This mapping has already been applied to matrix–vector multiplication [12] and could be useful in other linear algebra algorithms.

Finally, we note that considerable effort has been expended to develop clever algorithms for all–to–all communication on various machine topologies [3, 16, 24, 25]. The examples most commonly cited as needing this form of communication are dense linear algebra calculations and many–body problems. Recent work has shown that the fastest dense linear algebra algorithms can avoid all–to–all communication [1, 4, 13]. It is our contention that the current work does the same for many–body problems.

Acknowledgements

We are indebted to David Greenberg for assistance in developing the hypercube transposition algorithm in §4.2, and general discussions about routing algorithms on parallel architectures.

References

[1] E. Anderson, A. Dongarra, J. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. LAPACK for distributed memory architectures: progress report. In *Proc. 5th SIAM Conf. Parallel Processing for Scientific Computing*, Houston, March 1991.

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

[3] D. P. Bertsekas, C. Özveren, G. D. Stamoulis, P. Tseng, and J. N. Tsitsiklis. Optimal communication algorithms for hypercubes. *J. Parallel Distributed Comp.*, 11:263–274, 1991.

[4] 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.

[5] Rajendra Boppana and C. S. Raghavendra. Optimal self–routing of linear–complement permutations in hypercubes. In *Proc. Fifth Distributed Memory Computing Conf.*, pages 800–808. IEEE, 1990.

[6] R. R. Brooks and M. Hodoscek. Parallelization of CHARM for MIMD machines. *Chem. Des. Auto. News*, 7:16, December 992.

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

[8] 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.

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

[10] G. S. Grest, B. Dunweg, and K. Kremer. Vectorized link cell Fortran code for molecular dynamics simulations for a large number of particles. *Comp. Phys. Comm.*, 55:269–285, 1989.

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

[12] B. Hendrickson, R. Leland, and S. Plimpton. A parallel algorithm for matrix–vector multiplication. *Intl. J. High Speed Comput.* To appear.

[13] B. Hendrickson and D. Womble. The torus–wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.* To appear.

[14] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles.* Adam Hilger, New York, NY, 1988.

[15] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on Boolean n–cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[16] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computing*, 38(9):1249–1268, September 1989.

[17] D. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison–Wesley, Reading, MA, 1973.

[18] 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.

[19] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Sci. Stat. Comput.*, 14(3):519–530, May 1993.

[20] S. J. Plimpton. Fast parallel algorithms for short–range molecular dynamics. Technical Report SAND 91–1144, Sandia National Laboratories, Albuquerque, NM, 1993.

[21] S. J. Plimpton and B. Hendrickson. Parallel molecular dynamics simulations of organic materials. *J. of Modern Physics C.* To appear.

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

[23] 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 Conf.*, pages 113–120. IEEE, 1992.

[24] Donald M. Topkis. All–to–all broadcast by flooding in communication networks. *IEEE Trans. Computing*, 38(9):1330–1333, September 1989.

[25] R. van de Geijn. Efficient global combine operations. In *Proc. 6th Distributed Memory Computing Conf.*, pages 291–294. IEEE Computer Society Press, 1991.

Bruce Hendrickson received a BS in Mathematics and an MS in Physics from Brown University in 1982, followed by a PhD in Computer Science from Cornell University in 1990. He is a Senior Member of the Technical Staff in the Applied and Numerical Mathematics Department at Sandia National Laboratories in Albuquerque, New Mexico. His interests include scientific computation, parallel algorithms and combinatorics.

Steve Plimpton received a BS in Physics from Brigham Young University in 1982, and a Ph.D. in applied and engineering physics from Cornell University in 1989. He is a Senior Member of the Technical Staff in the Parallel Computational Sciences Department at Sandia National Laboratories in Albuquerque, NM, where he has worked on a variety of computational physics problems using parallel computers. His chief interests are in using parallel particle methods for materials modeling (molecular dynamics, Monte Carlo) and for simulating low-density fluid flows.