Molecular Dynamics on a Distributed-Memory Multiprocessor

S. L. Lin
J. Mellor-Crummey
B. M. Pettitt
G. N. Phillips, Jr.

CRPC-TR92295
January 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Appears in Journal of Computational Chemistry (13: 1022-1035).

Molecular Dynamics on a Distributed-Memory Multiprocessor

S.L. Lin,[1,2] J. Mellor-Crummey,[1,3] B.M. Pettitt,[1,4] and G.N. Phillips, Jr.[1,2]*

[1]*The W.M. Keck Center for Computational Biology, Departments of* [2]*Biochemistry and Cell Biology and* [3]*Computer Science, Rice University, Houston, TX 77251, and* [4]*Department of Chemistry, University of Houston, Houston, TX 77204-5641*

Received 22 January 1992; accepted 29 April 1992

Dynamics simulations of molecular systems are notoriously computationally intensive. Using parallel computers for these simulations is important for reducing their turnaround time. In this article we describe a parallelization of the simulation program CHARMM for the Intel iPSC/860, a distributed memory multiprocessor. In the parallelization, the computational work is partitioned among the processors for core calculations including the calculation of forces, the integration of equations of motion, the correction of atomic coordinates by constraint, and the generation and update of data structures used to compute nonbonded interactions. Processors coordinate their activity using synchronous communication to exchange data values. Key data structures used are partitioned among the processors in nearly equal pieces, reducing the memory requirement per node and making it possible to simulate larger molecular systems. We examine the effectiveness of the parallelization in the context of a case study of a realistic molecular system. While effective speedup was achieved for many of the dynamics calculations, other calculations fared less well due to growing communication costs for exchanging data among processors. The strategies we used are applicable to parallelization of similar molecular mechanics and dynamics programs for distributed memory multiprocessors.

INTRODUCTION

Simulation methods for many-particle systems have wide use in theoretical chemistry from material science to rational drug design and protein engineering. Computer simulation often complements analytic theory and experimental methods, and can probe phenomena where other methods have fundamental or technical difficulties. The potential of simulation methods seems only to be limited by the capacity of computers and the quality of the models employed.

Molecular dynamics simulations provide unique information about molecular systems. By following atomic movements in a simulation, actions of molecules can readily be visualized and analyzed. For example, the hinged "lid" motions of some enzymes have been suggested by molecular dynamics simulations to relate to reactivity.[1] More importantly, microscopic trajectories within the phase space of a system implied by a statistical ensemble can be obtained by molecular dynamics simulation; such trajectories yield the macroscopic properties of the system. This capacity, for example, aids in the interpretation of the stability of proteins and the specificity and catalytic capacity of enzymes.[2] Molecular dynamics also finds usage in theoretical issues such as the existence of multiple energy minima in proteins.[3]

Protein dynamics simulations are notoriously computationally intensive. The large computational requirements of such simulations are primarily due to three features. First, simulations typically consist of thousands of protein atoms, often together with even more solvent atoms. Second, accurate dynamics simulation requires multiple types of interactions; among them the long-ranged nonbonded interactions are particularly time-consuming. Third, as a high correlation exists between neighboring atoms, the trajectory of a protein diffuses slowly in phase space, making it difficult to achieve adequate sampling in this statistical ensemble. Even with great simplification in modeling the interactions, it is not uncommon to take hundreds of CPU hours on a CRAY-class vector supercomputer to complete a protein dynamics simulation. Unfortunately, high-performance conventional supercomputers are extremely costly. The high cost of such machines limits their availability for biomolecular research such as protein dynamics simulations.

*Author to whom all correspondence should be addressed.

CCC 0192-8651/92/081022-14$04.00

In recent years, parallel computers have advanced to the point where their performance is competitive with, often exceeding that of conventional vector supercomputers for certain problems. Also, parallel computers are sometimes much more cost-effective than conventional vector supercomputers.[4] Because the performance and price/performance of parallel computers is expected to continue to improve at much faster rates than those of conventional vector supercomputers, parallel computers are generally regarded as the architectures of the future for scientific computing.

The key impediment to the widespread use of parallel computers for scientific computing is their demand for a higher degree of integration of hardware, algorithm, and programming and the difficulty of developing correct, efficient, and portable parallel programs. Although parallel computers can deliver extremely high peak performance, achieving a significant fraction of that performance for a particular application can be extraordinarily difficult requiring machine-dependent optimization in addition to algorithm restructuring. Some of the available parallel machines require their own language and most of them are cumbersome to program because they support only primitive programming models that require users to explicitly manage concurrency. On the other hand, molecular dynamics appears to be an application that can benefit substantially from parallel execution. Movement of atoms in a molecular system is essentially simultaneous and thus lends itself naturally to parallel computation.

The potential for improving turnaround time of molecular dynamics simulations and the challenges of exploiting parallelism effectively in such simulations have prompted recent exploration of ways to implement molecular dynamics on a variety of parallel computers.*[7] Recently, we ported the program CHARMM[8] to the Intel iPSC/860 parallel computer. In this article we discuss the parallelization of molecular dynamics computations that we implemented in CHARMM and its performance.

iPSC/860 AND CHARMM

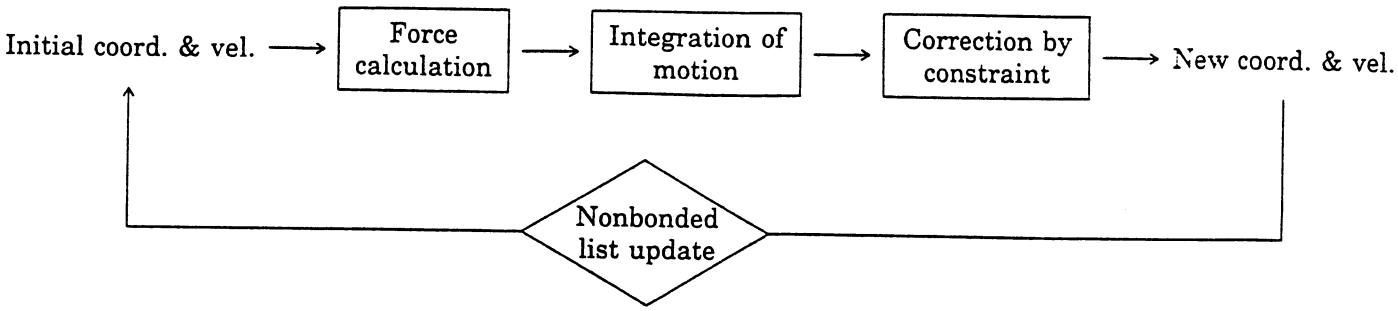
The Intel iPSC/860[9,10] is an MIMD distributed-memory multiprocessor. Each processing node of the system is built around a 40-MHz Intel i860 microprocessor with 8 Mb of local memory that provides space for an application's code and data in addition to a copy of the node operating system. Nodes in the multiprocessor are connected in a hypercube topology. Communication between nodes is accomplished using message passing. The operating system supports routing of messages between arbitrary pairs of nodes and access to attached peripheral storage devices. A 16-MHz PC/386 front-end serves as an interface between the multiprocessor and a network computing environment. The i860 microprocessor has a theoretical computation speed of 80 MFLOPS for single precision arithmetic and 40 MFLOPS for double precision arithmetic. However, such performance is rarely achieved in practice because it is only possible for computations that contain a certain ratio of addition to multiplication and exhibit data reuse patterns that make cache and registers effective.

CHARMM is one of several available molecular simulation programs developed primarily for biomolecular systems. The program has a relatively friendly user-interface and has been run on many midsize and mainframe computers. CHARMM (v.20) consists of nearly one hundred thousand lines of FORTRAN (preprocessed from the original Flecs code). For parallelization, it might be desirable to translate the program into a universal parallel language to make it available for any parallel machine; we are exploring the use of FORTRAN D,[11] a machine-independent parallel language for this purpose. However, it is unclear at present that such a language can be compiled to effectively exploit a variety of parallel architecture. As an interim approach, we have rewritten critical portions of CHARMM in message-passing FORTRAN for the Intel iPSC/860 to evaluate the parallelism inherent in its core algorithms and to provide a tool for high-performance molecular dynamics simulation.

DATA STRUCTURE, ALGORITHM, AND PARALLELISM

CHARMM begins a molecular dynamics simulation using an initial set of atomic coordinates and atomic velocities at time $t = t_0$. For each δt time step the simulation cycles through three major stages: the calculation of forces applied to each of the atoms, the integration of equations of atomic motion, and an optional correction to the resulting coordinates to constrain selected interatomic distances to fixed lengths. An enumeration of the pair-wise interactions of nonbonded atoms, called "nonbonded list," is updated in a period of n time steps, where n is usually between 10–100. These four stages constitute the core of the molecular dynamics computation, as shown in the following diagram:

*For parallelization of molecular dynamics, see, for example, ref. 5. For introductory materials on parallel processing, see, for example, ref. 6.



These stages in CHARMM must execute sequentially with respect to each other. Hence parallelism is desirable for implementation at each individual stage. A balanced distribution of the core computations is the major goal of parallelizing the molecular dynamics. Namely, computational work and related data should be divided among the processors as evenly as possible. The parallelization should accomplish this efficiently and coordinate the processors as well as give correct results.

In our prototype parallelization of CHARMM, identical copies of the executable are loaded onto each of the nodes of the iPSC/860. Static data structures used by CHARMM are replicated on all of the nodes. For each of the core computations, work is partitioned and the nodes cooperate in performing the computation. The nodes also generate some distributed data structures that are dynamically allocated by CHARMM using its internal heap and stack architecture. Input and output are handled using a concurrent read, exclusive write strategy, in which the input data stream is fed identically to all of the nodes while only one node writes to external files. Communication is always synchronous using global communication and arithmetic functions provided by a system library for the iPSC/860.

A protein–water sample system is used to examine the performance of the parallelization. The sample system is close to one we have used in realistic molecular dynamics simulations. It consists of a single chain of the globular protein, myoglobin, and a surrounding water shell. The protein has 1541 atoms; 1743 water molecules make up a total of 6770 atoms in the system. The test runs consist of a 1 ps dynamics simulation integrated using the "stochastic boundary" condition,[12] with an 8 Å cutoff range for the nonbonded list and 7.5 Å for the nonbonded force calculations. Stochastic boundary condition was used because of its more general nature and hence complexity. The nonbonded list is updated every 10 fs. The water molecule geometry is fixed at the equilibrium configuration and in the protein the bonds to hydrogens are fixed at their equilibrium lengths. The computation of the sample system spends about 67% of time in the calculation of forces, 8% in the integration of the equations of motion, 11% in the correction of coordinates by constraint, and 13% in the update of the nonbonded list, to total over 99%.

In the following sections we discuss the parallelization of the aforementioned four core computations of the molecular dynamics for the iPSC/860. For each of them we examine the corresponding CHARMM data structure and algorithm, discuss our parallelization strategy, outline the implementation, and present results from measuring simulations of the sample system. The discussion is based on a version of CHARMM-20. For the sake of reporting execution time, the iPSC/860 system library function MCLOCK is used by the program to measure time intervals. All timing measurements as well as data sizes reported in this article were recorded by one processor during a simulation. Checks have been made to ensure there are no significant deviations either among the processors or between repeated tests.

Calculation of Forces

Atoms in a molecular system are driven by time-dependent potential forces. The forces are determined by the relative positions of the atoms. As the atoms move the forces change in tandem. For polyatomic molecules the forces are customarily described as the gradient of a potential force field modeled by multiple energy terms. For an N-atom system when the atoms are at positions $\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N$, CHARMM describes the potential field V as

$$\begin{aligned} V(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) = & \sum_{\text{bonds}} K_b(b - b_0)^2 \\ & + \sum_{\text{valence angles}} K_\theta(\theta - \theta_0)^2 \\ & + \sum_{\text{dihedral}} K_{\phi_p}[1 + \cos(n_p\phi_p - \delta_p)] \\ & + \sum_{\text{improper dihedral}} K_{\phi_i}[1 + \cos(n_i\phi_i - \delta_i)] \\ & + \sum_{\text{nonbonded pairs}} \left(\frac{A}{R^{12}} - \frac{B}{R^6} + \frac{Q_1Q_2}{\varepsilon R}\right) \end{aligned} \quad (1)$$

In eq. (1) b, b_0, and K_b are the bond length, its equilibrium value parameter, and the bond stretching force constant, respectively; θ, θ_0, and K_θ are the bond angle, its equilibrium value parameter, and the angle bending force constant, respectively; ϕ_p, K_{ϕ_p}, n_p, and δ_p are a dihedral angle, its force constant, the symmetric multiplicity, and phase, respectively; ϕ_i, K_{ϕ_i}, n_i, and δ_i are an improper dihedral angle, its force constant, the symmetric multiplicity, and phase, respectively; Q_1 and Q_2 are the charges of two atoms and R is the distance between them, ε is the dielectric constant, and A and B are the coefficients of the 6-12 van der Waals potential.[13]

An atom at position $\mathbf{r}$ is subject to the force $\mathbf{F}(\mathbf{r}) = -\nabla V(\mathbf{r})$. To determine the force for an atom eq. (1) indicates that there are several force terms to be calculated. The first four terms in eq. (1) are framework terms named bonds, angles, and torsions; the latter includes dihedral and improper dihedral terms. The last term in eq. (1) includes the nonbonded force terms of both van der Waals and Coulombic interactions, which are calculated for atom pairs across space. The first four terms scale with the number of atoms and are of order N. The last terms are potentially of order N^2; however, typical approximations reduce this term to order N as well as described below.

CHARMM uses similar data structures and algorithms for each of the bonded terms. They are based upon enumerations of the interactions in each term. Taking the valence angle potential interactions described by the term

$$\sum_{\text{valence angles}} K_\theta(\theta - \theta_0)^2 \qquad (2)$$

as an example, the data structure consists of a set of lists of size proportional to the number of angular interactions. Three lists each enumerate one of the three atoms forming the angle θ for each interaction. Another list specifies (for each interaction) an index into a parameter table containing constants K_θ and θ_0. Algorithm 1 shows pseudo-code for calculating the angular forces from the individual interactions.

Algorithm 1

loop over angles
  fetch $\mathbf{r}$*'s of the three atoms*
  calculate θ *from* $\mathbf{r}$*'s*
  look up K_θ *and* θ_0 *at the parameter table*
  calculate $\mathbf{F}(\mathbf{r})$ *at* $\mathbf{r}$*'s*
end loop

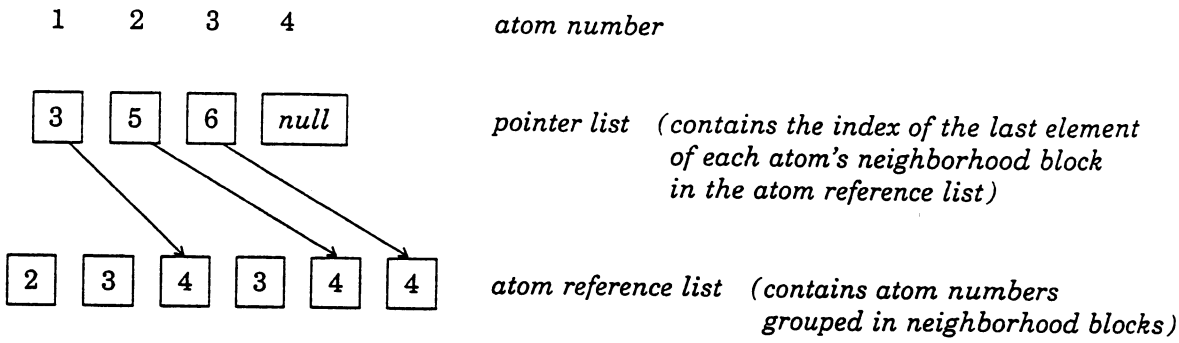
In bonded-term calculations such as the one shown in Algorithm 1, there are no dependences between loop iterations other than the accumulation of the forces on each atom that result from multiple bonds (this is why such terms also work well on vector machines). Therefore, a straightforward partitioning of loop iterations among the processors will obtain parallelism without major difficulty; communication will be required to sum up the elementary forces computed by each of the processors. The atom reference lists and parameter reference list can be distributed among the processors before the dynamics starts, and remain unchanged during an entire dynamics session. For each bonded term, the atom reference lists and the parameter reference lists together have size equal to

(the number of interactions)
× (1 + the number of atoms in the interaction) (3)

The number of interactions is of the same order of magnitude as the number of atoms. Because of differences in the data and loop size for different bonded term calculations, each calculation may require a different loop and data partition.

The nonbonded force term considers only pairwise interactions between atoms across space. If there are N atoms in the system there can be up to $N(N-1)/2$ nonbonded interaction pairs, minus some bonded pairings that are omitted. Because of the large number of interactions, computation of this term typically has the largest requirements for memory and CPU time. A common strategy to reduce the $O(N^2)$ burden is to limit the range over which nonbonded interactions will be considered using a range cutoff or interaction taper. Such strategies reduce the number of interactions to be considered to $O(N \cdot \text{cutoff}^3)$ and require the use of a data structure to trace the pairings that will be considered. Even with physically reasonable cutoff values, the number of interactions in practice still tends to be tens to hundreds of times larger than N. To minimize memory requirements, CHARMM constructs the nonbonded data structure differently from those of the bonded terms.

The CHARMM nonbonded data structure enumerates a list of pairing partners for each atom. An atom's pairing partners form a neighborhood block; all blocks are concatenated contiguously into an atom reference list. An array of pointers maps each atom to its neighborhood block in the atom reference list. The atom reference list and the pointer list together make up the data structure called the nonbonded list. The following example illustrates a nonbonded list that describes four atoms fully connected in six pairs:



The pairs of atoms are 1-2, 1-3, 1-4, 2-3, 2-4, and 3-4 in the above example.

The nonbonded list has a size of

(the number of nonbonded pairs) + (total number of atoms) (4)

This representation saves a factor of two over a naive enumeration of interaction pairs. Supplementary to the nonbonded list there are also structures for the access to the parameter table for the force field constants A, B, Q_1, and Q_2 in eq. (1). The size of these structures is on the order of the number of the atoms.

Corresponding to the differences in data structure, the CHARMM algorithm for calculation of the nonbonded forces differs from those of the bonded terms. Algorithm 2 shows pseudo-code for calculation of the nonbonded forces. Algorithm 2 enumerates the nonbonded interactions through double loops instead of through a single loop as featured in the bonded-term algorithms.

Algorithm 2

```
loop over atoms
    fetch r of the atom
    loop over the atom's partners
        fetch the partner's r
        calculate R and look up relevant
          parameters
        calculate F(r) at r's
    end loop
end loop
```

Due to the large number of nonbonded interactions, calculation of the nonbonded force is the most time-consuming computation in the molecular dynamics simulation. Therefore distributing the nonbonded force calculation is important. Although the bonded and nonbonded algorithms differ in form, they use essentially the same logic. Both algorithms iterate over all interactions enumerated in their respective data structures and calculate the forces exerted on each atom. Therefore partitioning methods like those we used for the bonded terms can also be applied to the calculation of the nonbonded terms, except for the specifics of how to manipulate the unique nonbonded data structure and the double loops in Algorithm 2. However, our strategy for partitioning the calculation of the nonbonded forces is merged with the strategy for parallelizing the generation and update of the nonbonded list. The strategy we use is presented in the next section, which discusses the generation and update of the nonbonded list.

In our prototype parallelization, we have chosen to parallelize the calculation of forces, as illustrated in Algorithms 1 and 2, by distributing the data structures and partitioning loop iterations among the processors in the manner discussed above. For the bonded terms, distribution is essentially perfect in the sense that the loads on the processors do not differ by more than one interaction. For the nonbonded term, the algorithm we use does not guarantee perfect distribution; however, it achieves near perfect balance. The method is expected to give a speedup nearly proportional to p on the partitioned loops, with computational overhead from the non-partitioned parts of the code. Communication between processors occurs after all the terms have been calculated. The iPSC/860 library function GDSUM is called to synchronize the processors, pass the scattered forces around, and perform an arithmetic summation to accumulate the forces and disseminate the results among the processors. The communication costs are approximately proportional to $N \log_2 p$.[10]

Table I lists the timing measurements of all the force term calculations against the number of processors p. Also listed are the time for an extra perturbation term, the total computation time, the communication time, and the total time of computation and communication. The perturbation term is calculated when a thermodynamic perturbation simulation, the free energy calculation, is requested. This term comprises two passes of the bonded terms and five passes of the nonbonded terms, operating on a subset of the data structures.

Table I shows that the single-term times halve as

Table I. Timing on the calculations of forces in a simulation of the sample system (in seconds).

	p					
	1	2	4	8	16	32
Bond	206	113	67	35	17	9
Angle	660	332	167	84	43	22
Dihedral	149	75	39	20	11	6
Improper	159	80	40	21	11	6
Nonbonded	9581	4778	2351	1180	583	290
Perturbation	3270	1722	949	555	360	263
Computation total	14,054	7130	3643	1924	1054	627
Communication	0	220	603	824	1075	1324
Total	14,054	7350	4246	2747	2130	1951

p doubles, within a small deviation, indicating computational overhead is small. The perturbation time scales less efficiently, which can be traced to the more significant contribution of sequential code in the corresponding procedure. The total computation time, which is the sum of the times spent on all of the force calculations, added with the extra "housekeeping" expenditure in the handling procedure, improves by nearly a factor of 2 for each doubling of p, up to $p = 32$, the maximum number of processors used in the timing test. Unfortunately, communication time rises when p increases, and roughly proportional to $\log_2 p$. When $p \geq 8$, the net speedup starts to dwindle as additional processors are added. Going from $p = 16$ to $p = 32$ the increase in communication overhead essentially offsets the gain from faster computation of the forces. It is expected that using more than 32 processors with the current iPSC architecture and the $\log_2 p$ communication tools would result in no further gain and probably a deterioration in the force calculations.

Generation and Update of the Nonbonded List

CHARMM generates the nonbonded list before the dynamics starts and updates it at given time intervals. The generation and update share an essentially identical procedure. The kernel of the procedure is a two-stage list refinement. At the first stage, the neighboring relationship of atom clusters, which are predefined by CHARMM as "groups",[8] are examined and the pairs of groups that are not separated beyond the cutoff range are picked out. At the second stage, the atoms in these pairs are examined and those that satisfy the cutoff criterion are then put into the nonbonded list. The pseudo-code for this procedure is shown in Algorithm 3.

Algorithm 3

```
loop over groups
    define a rectangular box enveloping the
      group
end loop
loop over groups
    loop over groups with larger or equal
      indices
        if the boxes of the two groups are
          within cutoff range
        then pair the groups up
    end loop
end loop
loop over groups
    loop over atoms in the group
        loop over pairing groups
            loop over atoms in the pairing
              group
                if the two atoms are within
                  cutoff range
                then pair up the atoms into
                  the nonbonded list
            end loop
        end loop
    end loop
end loop
```

The two-stage strategy greatly enhances the efficiency of nonbonded list generation and update. We have compared the algorithm with one that does not prune group pairing preemptively, namely, simply to examine all the $N(N-1)/2$ pairs of atoms. Sequential executions on the sample system demonstrated a 10-fold difference in their speeds (data not shown). This step scales essentially as the order of N^2 (we should note that other more complex strategies that offer better asymptotic efficiency by sorting groups by position in space and only considering pairings of nearby groups may not seem warranted for molecular systems of the size we considered).

An efficient parallel algorithm should achieve an even distribution of both the computation load and the final nonbonded list structure. Algorithm 3 indicates determination of the size of the most demanding computation segment, namely, the last major loop, has to be delayed until after the finish of the first loops. Furthermore, the size of the nonbonded list cannot be ascertained until the procedure is completed. Based on the information flow, we derive a strategy for generation of the nonbonded list described as follows.

The strategy uses loop partitioning for parallelism, which Algorithm 3 suggests to be the natural choice. The crucial piece of the strategy is a *preprocessing* step, after which the loops are partitioned and the processors work on a distributed

nonbonded list. For clarity, we call the groups referred to in the outermost loops in Algorithm 3 the master groups, in contrast to the pairing groups in the nested loops.

1. When an initial generation of the nonbonded list is requested before the dynamics starts, a *preprocessing* step is activated. Each processor performs a one-time mock execution of Algorithm 3 to assess the size of the nonbonded list and how the master groups contribute to the product. This does everything but allocate storage space for the list.
2. Once assessment is done, the master groups are divided as evenly as possible with regard to their contributions among the number of processors. The outermost loops in Algorithm 3 are then partitioned accordingly. This step establishes for each processor a local subset of the master groups that overlook all pairing groups in the subsequent allocation and update procedures.
3. Subsequently, allocation is accomplished by using the partitioned Algorithm 3. Each processor only executes its own partition of loops and allocates a partial nonbonded list. This step completes the generation procedure.
4. The update procedure is equivalent to the above step: Processors execute the partitioned Algorithm 3 to operate on their local partition of the data structure.
5. A new mechanism, *repartition* of the nonbonded list, is introduced into the CHARMM utility, to allow the user to force the generation procedure in the middle of the dynamics. This may be used to rebalance the loads periodically.
6. In the phases between generation, update, and repartition, the distributed nonbonded list is stable and is used for calculating the nonbonded forces. Each processor computes only the interaction pairs registered in its partial list.

The scheme is relatively simple and has been very effective compared to alternatives. It achieves remarkably well balanced distribution of data and computation as well as no communication costs. As can be seen in Table II, a sample run demonstrates that computation speedup is nearly proportional to the number of processors, p, as expected.

Table II. Timing on the update of the nonbonded list (in seconds).

	p					
	1	2	4	8	16	32
Nonbonded-list update	2818	1663	929	474	248	138
Communication	0	0	0	0	0	0

The generation procedure results in a balanced distribution of the nonbonded list, which accounts for over 500,000 atom pairs in the sample. During the 1 ps dynamics the nonbonded list distribution deviated slightly by the 100 updates, while no repartition was performed. The balance of the nonbonded list distribution can be appreciated in the performance of the nonbonded force calculation. The nonbonded force calculation follows very closely to a proportional speedup, as Table I has shown, indicating distribution of the nonbonded list remains nearly perfect throughout the sample simulation.

The nonbonded list data structures account for the bulk of the dynamically allocated storage used by CHARMM. CHARMM uses an array HEAP to represent dynamically allocated storage. In our parallelization, each processor node has its own private HEAP. Distributing the nonbonded list data structures among the processors rather than replicating them saves significant amounts of storage space. Table III shows the peak occupancy of HEAP for the testing dynamics simulations using 1–32 processors. The decrease in HEAP occupancy as the number of processors increases results chiefly from distribution of the nonbonded list.

Integration of Equations of Motion

CHARMM integrates the Newton's equation of motion by the Verlet algorithm.[14] The algorithm derives the next position, $\mathbf{r}(t + \delta t)$ of an atom of mass m and subject to force $\mathbf{F}(\mathbf{r})$, from its present position $\mathbf{r}(t)$ and previous position $\mathbf{r}(t - \delta t)$:

$$\mathbf{r}(t + \delta t) = 2\mathbf{r}(t) - \mathbf{r}(t - \delta t) + \delta t^2 \frac{\mathbf{F}(\mathbf{r}(t))}{m} \quad (5)$$

The atom's velocity $\mathbf{v}(t)$ is approximated using the central difference formula

$$\mathbf{v}(t) = \frac{\mathbf{r}(t + \delta t) - \mathbf{r}(t - \delta t)}{2\delta t} \quad (6)$$

CHARMM also provides an option for the integration of Langevin's equations, to provide a mechanism to couple the simulated system to a heat bath. This option supplements a frictional force $\mathbf{f}(\mathbf{r}(t))$ randomly generated for the atoms at the time step; any Cartesian component f of the random force f satisfies statistical properties

$$\langle f(t) \rangle = 0, \qquad \langle f(t), f(0) \rangle = 2k_B T \beta m \delta(t) \quad (7)$$

where k_B is the Boltzman's constant, T is the temperature of the heat bath, β is the frictional coefficient, and m the mass of the atom at $\mathbf{r}$.[12] The integration of the Langevin's equations is then

Table III. Effect of distributing nonbonded list on the HEAP occupancy (in bytes).

	p					
	1	2	4	8	16	32
Peak HEAP occupancy	2,357,952	1,369,840	868,752	618,400	492,464	429.952

$$\mathbf{r}(t+\delta t) = \frac{2\mathbf{r}(t) - (1 - \frac{1}{2}\beta\delta t)\mathbf{r}(t-\delta t) + \delta t^2(\mathbf{F}(\mathbf{r}(t)) + \mathbf{f}(\mathbf{r}(t))/m)}{1 + \frac{1}{2}\beta\delta t} \quad (8)$$

The code for integration by Langevin's equations can be simplified as shown in Algorithm 4.

Algorithm 4

generate random forces
loop over atoms
    calculate $\mathbf{r}(t+\delta t)$ *according to* eq. (8)
    calculate $\mathbf{v}(t)$ *according to* eq. (6)
    let $\mathbf{r}(t-\delta t) = \mathbf{r}(t)$
    let $\mathbf{r}(t) = \mathbf{r}(t+\delta t)$
end loop

The time complexity of this integration step is $O(N)$, where N is the number of atoms.

The procedure in CHARMM that implements integration also clusters other miscellaneous operations that must be done at the same time step such as calculating the system temperature, making preparations for the next force calculation and constraint corrections. Many of these operations are in the form of loops, which can be partitioned across the processors.

The sample simulation was executed with the stochastic boundary conditions[12] for which Algorithm 4 is used. A nonzero frictional coefficient β was applied to the molecular system, except for an internal spherical volume of a 10 Å radius, where β was set equal to zero, resulting in simple Newtonian behavior for the atoms in this region. Testing results of the integration procedure are listed in Table IV. They show that the computation time at the integration stage is not very sensitive to the number of processors. Close examination reveals that the readily parallelizable work in the integration procedure is only about 10% of the total work; thus the parallelization is far from thorough for this procedure. Communication costs are contributed chiefly from a library function GCOLX, a global concatenation function to distribute the results.

Table IV. Time spent in the procedure of motion integration (in seconds).

	p					
	1	2	4	8	16	32
Motion integration, etc.	1714	1630	1583	1562	1550	1545
Communication	0	101	129	154	165	176
Total	1714	1731	1712	1717	1716	1721

Coordinate Correction Due to Constraints

The atomic movement of macromolecules as big as a protein covers a wide range of frequencies. The highest frequency determines the maximum time step allowed in a molecular dynamics simulation. Some internal vibrations of higher frequencies, especially those involving hydrogen atoms, are readily decoupled from lower-frequency motions. Removing them enables the use of larger time steps that save computer time. CHARMM facilitates the removal of selected bond-stretching and angle-bending, by fixing corresponding interatomic distances. Fixed-distance constraints are applied to the equations of motion by the method of Lagrangian multipliers. They result in a correction to atomic coordinates obtained from the unconstrained equations, eq. (5). For atom i, the correction $\delta\mathbf{r}_i$ is calculated by

$$\delta\mathbf{r}_i = \sum_u g_{iu}\frac{\mathbf{r}_{iu}(t)}{m_i} \quad (9)$$

where the sum is over i's bonded neighbors and the gs are to be solved by the simultaneous equations of the form

$$(\mathbf{r}_{ij}(t+\delta t) + \delta\mathbf{r}_i - \delta\mathbf{r}_j)^2 = d_{ij}^2 \quad (10)$$

for all constraints, where d_{ij} designates the constant distance between atoms i and j (for details, see ref. 15). Equation (10) presents a nonlinear problem. CHARMM employs the SHAKE algorithm,[15] which linearizes eq. (10) to the form

$$2g_{ij}\left(\frac{1}{m_i} + \frac{1}{m_j}\right)(\mathbf{r}_{ij}(t+\delta t)\cdot\mathbf{r}_{ij}(t)) = d_{ij}^2 - \mathbf{r}_{ij}^2(t+\delta t) \quad (11)$$

which is solved iteratively, followed by application of Newton's third law to correct $\mathbf{r}_i(t+\delta t)$ and $\mathbf{r}_j(t+\delta t)$ by

$$\delta\mathbf{r}_i = g_{ij}\frac{\mathbf{r}_{ij}(t)}{m_i}, \qquad \delta\mathbf{r}_j = -g_{ij}\frac{\mathbf{r}_{ij}(t)}{m_j} \quad (12)$$

The iteration continues until all the constraints ex-

pressed by eq. (10) are satisfied within a given tolerance. The number of constraints scales as order N.

CHARMM has a data structure for the constraints similar to that of the bonded interaction. Developed along the constraints, there are two lists enumerating the atoms in pair whose distances are to be fixed, and a list of the designated distance constants. The following pseudo-code describes the process of correcting the atom coordinates by constraint.

Algorithm 5

repeat
  for all constrained atom pairs
    solve eq. (11)
    evaluate eq. (12)
    update the coordinates of the two atoms
until all constraints satisfied

In terms of solving a system of thousands of simultaneous nonlinear equations, SHAKE is quick. The power of the SHAKE algorithm lies in two aspects of its design. First, the constraint equations are usually highly sparse: The number of variables in any one of the constraint equations is seldom more than half a dozen for a realistic molecular system. By linearization SHAKE actually keeps only one variable for each equation. Iterative algorithms work particularly well with sparse systems.[16] The second factor is the immediate update of rs after an equation is solved. With the use of the new rs in upcoming equations, immediate update often, though not necessarily, provides better starting atom positions for a later constraint, hence resulting in quick convergence.

However, with regard to decomposing the algorithm for parallel execution the immediate updates of the atom's coordinates cause serious interdependency of the coordinates across the loop, resulting in a high communication overhead for any naive loop partitioning strategy. However, immediate update of coordinates only benefits evaluation of other constraint equations involving the same atom. In typical biomolecular applications of SHAKE the constraints are localized so that the groups of dependent equations are small. Moreover, these dependences can be identified in advance by inspecting the molecular topology because dependences only occur for the constraints of joint bonds. Therefore by separating dependent and independent equations, SHAKE can be envisioned as a *two-dimensional* algorithm, one dimension defined by an axis along which the independent constraints are placed, and the other dimension along an axis where the dependent constraints are strung together. We then have the expression

$$\text{constraints} = \sum_{\substack{\text{along axis of} \\ \text{independence}}} \sum_{\substack{\text{along axis of} \\ \text{dependence}}} (\text{single constraint}) \quad (13)$$

Update of the atomic coordinates is required only along the axis of dependence, and along the axis of independence the constraints are readily partitionable. The distinction between the two dimensions leads to our "*2-D SHAKE*" parallel algorithm.

The 2-D SHAKE algorithm includes two stages. The first is a one-time construction of a partitioned constraint data structure. The construction collects dependent constraints into sets and scatters the sets across the processors evenly, with all members of each set contained completely in individual processors.[17] At the regular stage of performing constrained coordinate corrections during the dynamics, eq. (13) expands to the following algorithm.

Algorithm 6

repeat
  loop over sets of dependent constraints
    for all constraints in the set
      solve eq. (11)
      evaluate eq. (12)
      update the coordinates of the two constrained atoms
  end loop
until all constraints satisfied
broadcast new coordinates

The 2-D algorithm can potentially speed up SHAKE by a factor near p, owing to the even distribution of the constraint sets achieved at the construction stage. The speedup also depends on the convergence of the constraints on individual processors. Dynamic balance schemes that are not discussed here can be used to even out the works, if the problem is considered significant. For the sample system under study, convergence was quite homogeneous, requiring approximately 20 repetitions on any of the processors. The timing measurements shown in Table V indicate that except for approximately 5% sequential overhead

Table V. Timing on SHAKE, the coordinate correction by constraints (in seconds).

	p					
	1	2	4	8	16	32
SHAKE computation	2197	1199	676	405	268	203
Communication	0	127	258	388	518	643
Total	2197	1326	933	792	787	846

that remained constant, the speedup of the remainder of the computation was proportional to the number of processors.

Communication is generally necessary in Algorithm 6 after all constraints have converged, to broadcast the corrections around the processors. We used GDSUM to sum up corrections for all of the atomic coordinates. Table V shows that communication expenditure in the test increases approximately proportional to $\log_2 p$. The sum of computation and communication times reaches minimum at around $p = 16$, giving a maximum parallel efficiency of about 3 over the single-processor execution.

It should be mentioned that for highly connected constraints for which degree of independence is low or balance is bias, different algorithms that may require more communication should be more suitable.[17]

Summary

We summarize the testing results in Table VI as the speedup ratios for using multiple processors against using a single processor, except that for the communication time the ratio is against using two processors. The total times and the percentage expenditures in the use of 1 and 32 processors are also listed in Table VI.

To evaluate the performance of the parallelization we introduce the following expression for the dependence of execution time, T, on the number of processors, p:

$$T(p) = a/p + b + c \ln p \qquad (14)$$

where the coefficients a and b describe the costs of the computations having been and not being parallelized, respectively, and c describes the communication costs. For the computations a reciprocal function is a good approximation, as justified by the discussions in the above subsections. The communication costs asymptotically follow $\ln p$ by using the iPSC library communication routines exclusively.[10]

Scalability of the parallelization against p can be conveniently discussed by evaluating p at the zero point of the derivative dT/dp to yield

$$p_c = a/c \qquad (15)$$

where p_c is the critical number of processors at which the speedup ratio $R \equiv T(1)/T(p)$ reaches maximum

$$R_c = \frac{a + b}{c(1 + \ln(a/c)) + b} < R_m \qquad (16)$$

where

$$R_m = \frac{a/c}{\ln(a/c)} \qquad (17)$$

$R_c < R_m$ is valid when $a > c$ and R_m is the limit of R_c when $a \gg b$ and $\ln(a/c) \gg 1$. R_c and p_c characterize how the parallel performance scales with the use of multiple processors.

What can we learn from these formulae?

First, we see that the cost ratio a/c uniquely defines both p_c and the limit of R_c. This quantifies what one's instinct would have realized that the more of the code being parallelized and the higher the communication efficiency, the better the parallel performance. A subtler point is that additional parallelizing with communication may or may not improve overall performance, depending on the

Table VI. Speedup ratios, percentage execution times in the use of 1 and 32 processors, and total execution times measured for the sample simulation.

	p						(%) p	
Speedup	1	2	4	8	16	32	1	32
Bond	1.0	1.8	3.1	5.9	12.1	22.9	1.0	0.2
Angle	1.0	2.0	4.0	7.9	15.3	30.0	3.2	0.5
Dihedral	1.0	2.0	3.8	7.5	13.5	24.8	0.7	0.1
Improper	1.0	2.0	4.0	7.6	14.5	26.5	0.8	0.1
Nonbonded	1.0	2.0	4.1	8.1	16.4	33.0	45.9	6.0
Perturbation	1.0	1.9	3.4	5.9	9.1	12.4	15.7	5.4
Force total	1.0	2.0	3.9	7.3	13.3	22.4	67.3	12.9
Nonbonded-list update	1.0	1.7	3.0	5.9	11.4	20.4	13.5	2.8
SHAKE	1.0	1.8	3.3	5.4	8.2	10.9	10.5	4.2
Integration procedure	1.0	1.1	1.1	1.1	1.1	1.1	8.2	31.8
Other operations	1.0	1.0	1.0	1.0	1.0	1.0	0.5	4.7
Computation total	1.0	1.8	3.1	4.7	6.4	7.7	100.0	55.8
Communication total	—	1.0	0.45	0.33	0.26	0.21	0.0	44.2
Total	1.0	1.7	2.6	3.6	4.2	4.3	100.0	100.0
Total time (h)	5.80	3.37	2.19	1.62	1.39	1.35		

cost ratio. Clearly, communication efficiency is the center piece of the issue. This is why we must conduct a parellelization with low communication. If faster communication were available, besides a gain directly attributable to a smaller c, we could have wider choice of algorithms and could make parellelization more thorough.

Second, the size of the molecular system, represented by the number of atoms, N, does not explicitly appear in the formulae. How the parallelization scales against system size N is determined by how a, b, and c depend on N. We can express a, b, and c in power series of N to include only the prominent terms expected for realistic values of N by

$$a = a_1N + a_2N^2$$
$$b = b_0 + b_1N + b_2N^2$$
$$c = c_0 + c_1N \qquad (18)$$

where the expansion coefficients are positive. From eqs. (15) and (18) one can easily find that p_c is an increasing function of N. From eqs. (16) and (17) one can also show that when $a > c$, a sufficient condition for R_c to be an increasing function of N can either be (1) $a_2/b_2 > R_m - 1$ or (2) $b_0 \geq b_2N^2$. The two conditions are independent. We note that b_0 is contributed chiefly from the sequential portion of the code in the procedures of integration, SHAKE, and force calculations, which are executed every time step; a_2N^2 and b_2N^2 are both from the work in the middle loop of Algorithm 3, but respectively for pairing groups in the nonbonded list update and generation, and are reciprocally proportional to the periods of update and repartition, respectively. Therefore condition (1) means a distribution of the $O(N^2)$ terms that accords with the limit speedup ratio, R_m, suffices scalability with regard to N. Regulating the update and repartition periods changes the distribution. Condition (2) conveys that the $O(1)$ term, whose weight in total computation diminishes as N increases, apparently helps scalability. Note that although the b_1N does not appear in condition (2), one can re-express the condition to include the term. It is conceivable that R_c can be non-scalable; however conditions are too convolved to formulate as concise.

Table VI shows p_c is around 32 and R_c is around 4.3. Least square fit to the total execution times gives $b/a = 0.114$, $c/a = 0.034$, $p_c = 29.5$, and $R_c = 4.24$. The moderate values of p_c and R_c can be attributed to several elements. The first and most significant is the communication costs. The value of c/a indicates rather high relative communication costs. The costs grow with p to consume 44% of the total execution time at $p = 32$. Second, the parallelization is not thorough. Although for the $N = 6770$ sample system about 90% of the work has been in parallel computation, a maximum speedup ratio of only 10 is expected for this system even if there is no communication. The most costly portion of the sequential code lies within the integration procedure, which accounts for 32% of the total execution time at $p = 32$. If the integration procedure were perfectly scalable, the fitted p_c would increase slightly to 32 and R_c would increase to 6.1, while the maximum speedup without communication would be 33 for the sample system. Third, judging by the speedup ratios of individual computations, sequential work in corresponding procedures that set up the partitioned iterations are not negligible for the sample system, except for the calculation of nonbonded forces. Fourth, the simulation conditions were set up for even presence of individual computational components. For example, the weight of the nonbonded force calculation is only 46% at $p = 1$, using a short cutoff distance (7.5 Å) upon which the calculation has cubic dependence, while a percentage well over 75% is more customary and over 90% is quite often in applications and testings.[5] Change of simulation conditions would affect all the numerical predictions presented above.

DISCUSSION

In this article we have examined the problems associated with parallelizing existing molecular dynamics algorithms on a distributed-memory multiprocessor. We used the Intel iPSC/860 as the platform for parallelization of the program CHARMM. The modification of the program is still ongoing; however, the shape of the strategies has been molded and the performance can be analyzed to reveal the pivotal elements in such kinds of parallelizations. Our strategy focuses on partitioning the workload of the core loops of the molecular dynamics computation. Quite different effects are observed in the four main stages, for a variety of reasons.

In the calculation of all force terms, nearly proportional speedup is achieved for up to 32 processors (Table VI). This performance may be attributed to the characteristics of the force fields and the algorithms for computing them. Except for a short setup stage, all the calculations are nested in loops. The computational work of the loops dominates the setup when the molecular system is not exceptionally small. However, the $O(N \log_2 p)$ communication overhead eclipses quickly the speedup of computation that is $O(N/p)$, to the extent that no further overall improvement is expected by using significantly more processors.

Our parallelization of the nonbonded list generation and update was quite successful. Computational speedup is close to proportional. More importantly, it gives rise to an evenly distributed nonbonded list, which results in a significant saving in time and space throughout the entire dynamics simulation (Tables I–III). Communication is not a factor so there is virtually no limit on how many processors can be used. A couple of factors contribute to the success of our strategy. First, we have taken advantage of a characteristic embedded in the pairing algorithm and data structure. This characteristic is the reducibility of the map of pairing from a complete graph to a union of directed subgraphs. The union can be distributed without interference between its members. Therefore one needs chiefly to be concerned with the criteria of partitioning. Second, we assume that the fluctuation of the atom pairing during the dynamics is not large enough to seriously impair the balance among the partitions. We also assume if there is a systematic drift in the geometry of the molecular system to induce a drift in the balance, the drift is slow enough to be remedied by infrequent repartition of the nonbonded list (clearly, problems on proteins involving partial unfolding might require regular repartition). In practice, the aforementioned assumptions of small fluctuation and slow drift in the conformation of molecular systems seem to hold.

For the computations at the stage of motion integration, loop partitioning shows a negligible effect, even though communication time remains low (Table IV). As already explained, this is because most of the computational load is outside the partitioned loops. The integration procedure whose computation time was measured contains many miscellaneous functionalities besides the integration of motion, necessary for updating all relevant informations after a time step. Obviously, other approaches must be investigated for this complex procedure, including optimizing the outside-loop computations.

For the correction of coordinates by constraint we have converted the SHAKE algorithm to a two-dimensional one. Taking advantage of the sparseness of the simultaneous equations of constraints and their locality in typical applications, the parallel algorithm separates dependent and independent constraints. Scattering of data structure and computation then becomes possible and nearly proportional speedup was observed (Table V). The general algorithm requires broadcast of corrections at the end of the computation that offsets the overall speedup as with the force calculations.

The efforts we made and the effects of the parallelism present a classical case of the interdependency of hardware, algorithm, and programming in parallel computation. Parallelization by loop partitioning is highly efficient for an MIMD machine, as we have already shown, while it is not a strategy suitable for an SIMD machine. Distribution of data structures such as the constraints and the nonbonded list is important for a distributed-memory computer, but is less significant for shared-memory machines. These two characteristics justify our choice of using loop partitioning and data distribution for a multiprocessor in the core molecular dynamics computations where loops dominate, for example, in the calculation of the nonbonded forces. However, the high communication costs of the chosen Intel iPSC/860 hardware and software limit the overall speedup. We have derived a no-communication strategy for the update of the nonbonded list, and a low-communication algorithm for the correction of coordinates by constraint. Even so, imbalance between communication speed and computation speed of the machine remains a bottleneck of the parallel performance and limits the algorithms that can be effectively exploited. Adopting hardwares with higher communication bandwidth, such as the newer Intel multiprocessors with 2-D mesh architecture, and softwares with higher communication efficiency such as the $O(N)$ algorithms[7] to replace corresponding $O(N \log p)$ algorithms provided by the current Intel library, will free the program from communication-bound and open up more possibilities for performance enhancement. To a certain extent, our choice to keep the infra-structure and user-interface of CHARMM intact also affected the strategies of parallelization.

We have demonstrated the usefulness and limitations of exploring parallelism in molecular dynamics by modifying an existing program. We have shown that it is realistic and promising to use multiprocessors for computations in molecular dynamics simulation. Most of the core computations have been shown to be parallelizable. The strategies we have used can be applicable to other programs for molecular mechanics and dynamics computations and to other MIMD machines. The features of our parallelization point to further gains from the next generation of distributed computers. Elements that effect the parallel performance have been analyzed and provide direction for further explorations.

APPENDIX

The CHARMM algorithms for the calculation of forces represent two kinds of basic design: an interaction-first approach, as in the bonded al-

gorithms, and an atom-first approach, as in the nonbonded algorithm. To account for all interactions, the interaction-first approach iterates over the interactions directly, using a data structure that enumerates all participating atoms for each of the interactions. On the contrary, the atom-first approach iterates over the atoms to calculate interactions associated with the atom, and hence adopts a data structure that enumerates for each atom the associated atoms involved in the interactions. The two approaches require different data structures of different sizes, and lead to different routes to complete the calculation, as demonstrated by eqs. (3) and (4) and Algorithms 1 and 2.

Because atoms and interactions are the prime entities in molecular dynamics, one might consider swapping the two approaches for the two classes of interactions. Memory considerations for the Intel iPSC/860 rule out the interaction-first approach for nonbonded terms, for it needs to almost double the size of the already quite large nonbonded list. For the bonded terms, size consideration is not as crucial: The change would be relatively small, and could be either upward and downward. An atom-first structure could be recast for the CHARMM bonded data structure, to form one that resembles the nonbonded data structure. One of the atom reference lists would be replaced by a pointer list. Sorted into appropriate blocks, the remaining atom reference lists and the parameter reference list would be accessed by the pointers. The atom-first scheme would allow all bonded terms to nest under the same outer loop, together with the nonbonded terms, as shown in the following algorithm.

Algorithm 7

loop over atoms
  fetch **r** *of the atom*
  execute the loop over the bonds led by the atom
  execute the loop over the angles led by the atom
  execute the loop over the dihedrals led by the atom
  execute the loop over the improper dihedrals led by the atom
  execute the loop over the nonbonded pairs led by the atom
end loop

Compared to separate bonded and nonbonded calculations, this scheme saves repeated references to the leading atoms. In terms of parallelization, it also makes it possible to unify the loop partitions. However this algorithm would not be suitable for vectorization, due to the short inner loops that would iterate few times for any realistic problem.

CHARMM constructs data structures according to system topology. If we construct the data structures by sorting atomic coordinates spatially and partition them among the processors accordingly, lower communication demand is possible owing to locality of the atoms. To obtain satisfactory locality a prerequisite is for each processor to hold a subsystem whose dimension is greater than the cutoff distance. For a typical biomolecular system of 10,000 atoms whose diameter is around 50 Å, a typical cutoff distance of 15 Å will allow the use of a moderate number of processors. The spatially sorted data structures should be the choice for much larger systems.

In future development of the parallel molecular dynamics, we will explore these alternative approaches further.

This research was supported by NSF Grant DMB 87-16507, the Robert A. Welch Foundation, and the W.M. Keck Center for Computational Biology.

References

1. D. Joseph, G.A. Petsko, and M. Karplus, *Science*, **249**, 1425 (1990).
2. W.F. van Gunsteren, *Protein Eng.*, **2**, 5 (1988).
3. R. Elber and M. Karplus, *Science*, **235**, 318 (1987).
4. K.C. Bowler, A.D. Bruce, R.D. Kenway, G.S. Pawley, and D.J. Wallace, *Physics Today*, **40(10)**, 40 (1987).
5. T.W. Clark and J.A. McCammon, *Comp. Chem.*, **14**, 219 (1990); T.W. Clark, J.A. McCammon, and L.R. Scott, In *Proceedings of the Fifth SIMA Conference on Parallel Processing for Scientific Computing*, March 1991, Houston, TX; C.J. Craven and G.S. Pawley, *Comp. Phys. Comm.*, **62**, 169 (1991); H. Heller, H. Grubmüller, and K. Schulten, *Mol. Sim.*, **5**, 133 (1990); M.R.S. Pinches and D.J. Tildesley, *Mol. Sim.*, **6**, 51 (1991); D.C. Rapaport, *Comp. Phys. Comm.*, **62**, 217 (1991); W. Smith, *Comp. Phys. Comm.*, **62**, 229 (1991); O. Teleman, B. Svensson, and B. Jönsson, *Comp. Phys. Comm.*, **62**, 307 (1991); A. Windemuth and K. Schulten, *Mol. Sim.*, **5**, 353 (1991).
6. T. Axford, *Concurrent Programming*, John Wiley & Sons, New York, 1989.
7. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Computers*, Prentice Hall, Englewood Cliffs, NJ, 1988.
8. B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, *J. Comp. Chem.*, **4**, 187 (1983).
9. *iPSC/2 iPSC/860 User's Guide*, Intel Corporation, Beaverton, OR, 1991.
10. L. Bomans and D. Roose, In *Hypercube and Distributed Computers*, F. Andé and J.P. Verjus, Eds., Elsevier Science Publishers B.V., North-Holland, Amsterdam, 1989.
11. G. Fox, S. Hiranandadi, K. Kennedy, C. Koelbel, U.

Kremer, C.-W. Tseng, and M.-Y. Wu, Fortran D language specification, Technical Report TR90-141, Department of Computer Science, Rice University, Houston, TX, December 1990, revised April 1991.
12. A. Brünger, C.L. Brooks III, and M. Karplus, *Chem. Phys. Lett.*, **105**, 495 (1984).
13. C.L. Brooks III, M. Karplus, and B.M. Pettitt, *Proteins: A Theoretical Perspective of Dynamics, Structure, and Thermodynamics*, John Wiley & Sons, New York, 1988.
14. L. Verlet, *Phys. Rev.*, **159**, 98 (1967).
15. J.-P. Ryckaert, G. Ciccotti, and H.J.C. Berendsen, *J. Comp. Phys.*, **23**, 327 (1977).
16. M.J. Maron, *Numerical Analysis: A Practical Approach*, Macmillan Publishing, New York, 1982.
17. S.L. Lin et al., in preparation.