Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library*

Yuan-Shin Hwang† **Raja Das**† **Joel H. Saltz**†
Milan Hodoščček‡ **Bernard Brooks**‡

† *UMIACS and Department of Computer Science*
University of Maryland
College Park, MD 20742
{shin, raja, saltz}@cs.umd.edu

‡ *Laboratory of Structural Biology, DCRT*
National Institutes of Health
Bethesda, MD 20892
{milan, brbrooks}@helix.nih.gov

Abstract

CHARMM (Chemistry at Harvard Macromolecular Mechanics) is a program that is widely used to model and simulate macromolecular systems. CHARMM has been parallelized by using the CHAOS runtime support library on distributed memory architectures. This implementation distributes both data and computations over processors. This data parallel strategy should make it possible to simulate very large molecules on large numbers of processors.

In order to minimize communication among processors and to balance computational load, a variety of partitioning approaches are employed to distribute the atoms and computations over processors. In this implementation, atoms are partitioned based on geometrical positions and computational load by using unweighted or weighted recursive coordinate bisection. The experimental results reveal that taking computational load into account is essential. The performance of two iteration partitioning algorithms, atom decomposition and force decomposition, is also compared. A new irregular force decomposition algorithm is introduced and implemented.

The CHAOS library is designed to facilitate parallelization of irregular applications. This library (1) couples partitioners to the application programs, (2) remaps data and partitions work among processors, and (3) optimizes interprocessor communications. This paper presents an application of CHAOS that can be used to support efficient execution of irregular problems on distributed memory machines.

*This work was sponsored in part by ARPA (NAG-1-1485) and NSF (ASC 9213821).

1 Introduction

Molecular dynamics (MD) simulation programs, such as CHARMM [1], GROMOS [2], and AMBER [3], are useful to study the structural, equilibrium, and dynamic properties of molecules. These programs are very complicated and computationally intensive. Implementing them on massively parallel processing systems not only reduces execution times but also allows users to solve larger problems. In the past, a number of algorithms have been designed and implemented for MIMD architectures [4]. However, certain difficulties arise when trying to achieve high performance with large numbers of processors, due to the irregular computational structure of MD programs. This paper presents an implementation of the CHARMM program that scales well on MIMD distributed memory machines. This program is parallelized using a set of efficient runtime primitives called the CHAOS runtime support library [5].

Several parallel MD algorithms use the replicated approach, i.e. the entire system's coordinates and forces are stored at each processor [4]. The replicated approach eliminates the overhead in maintaining the data distributions, but it does require more memory space. This prohibits users from simulating larger systems. Furthermore, the method is not scalable for large numbers of processors because each processor has to communicate all coordinates and forces to all other processors in order to make them consistent at the end of each time step. In this paper, spatial domain decomposition algorithms have been applied to distribute data and computations over processors based on the coordinates of atoms in the system. This approach provides better locality, load balance, and scalability.

Molecular dynamics simulates the local and global motion of atoms in molecules. Each of the N atoms in the simulation is treated as a point mass. Simulations involve the iterative computation of the total potential energy, forces, and coordinates for each atom in the system in each time step. The potential energy includes bonded energy terms, which consists of bond energy, angle energy, and dihedral angle energy, and non-bonded energy terms, which includes van der Waals and electrostatic potentials. The bonded force calculations consume only a small part of computation time because each atom has few numbers of chemical bonds with other atoms, whereas the non-bonded force calculations consume most of MD simulation time since each atom interacts with all other atoms in the system if no cutoff is imposed. In this paper, two approaches, *data decomposition* and *force decomposition* [6], have been employed to parallelize the non-bonded force calculations. Although the regular force decomposition algorithm partitions atoms evenly over processors, the (non-uniformly sparse) non-bonded list is distributed unevenly and hence severe load imbalance occurs. This paper presents an irregular force decomposition algorithm that maintains good load balance and achieves good performance.

The CHAOS library is designed to facilitate parallelization of irregular applications on distributed memory multiprocessor systems. It is a superset of the PARTI library [7]. This library (1) couples partitioners to the application programs, (2) remaps data and partitions work among processors, and (3) optimizes interprocessor communications. This implementation of parallel CHARMM presents an application of CHAOS that can be used to support efficient execution of irregular problems on distributed memory machines.

In the next section, a brief description of molecular dynamics simulations is given. In Section 3, the design philosophy and functionality of the Maryland CHAOS library is described. Section 4 presents the parallelization approaches and optimizations. The performance of the parallel CHARMM is presented in Section 5. Section 6 briefly describes related work. Conclusions are presented in Section 7.

2 Molecular Dynamics (CHARMM)

CHARMM is a program which calculates empirical energy functions to model macromolecular systems. The purpose of CHARMM is to derive structural and dynamic properties of molecules using the first and second order derivative techniques [1].

The computationally intensive part of CHARMM is the molecular dynamics simulation. This calculation is usually performed when the molecular structure reaches a local stable state (low energy) through energy minimization. It simulates the dynamic interactions among all atoms in the system for a period of time. For each time step, the simulation calculates the forces between atoms, the energy of the whole structure, and the movements of atoms by integrating Newton's equations of motion

$$\frac{\partial^2 x_i}{\partial t^2} = -\frac{\nabla E(x_i)}{m_i}.$$

It then updates the coordinates of the atoms. The positions of the atoms are fixed during the energy calculation; however, they are updated when spatial displacements due to forces are calculated.

The loop structure of molecular dynamics simulation is shown in Figure 1. The physical values associated with atoms, such as velocity, force and displacement, are accessed using indirection arrays (IB, JB, IT, JT, KT, JNB). IB and JB list all the bonds in the system; IB(I) and JB(I) are the atoms connected by bond I. IT, JT, and KT represent all the angles; IT(I), JT(I), and KT(I) are the atoms that construct angle I. JNB is the array that stores the non-bonded lists of all atoms. The energy calculation in the molecular dynamics simulation comprises two types of interactions – bonded and non-bonded.

Bonded forces exist between atoms connected by chemical bonds. CHARMM calculates four types of bonded forces – bond potential, bond angle potential, dihedral angle (torsion) potential, and improper torsion. These forces are short-range, i.e. forces existing between atoms that lie close to each other in space. Bonded interactions remain unchanged during the entire simulation process because the chemical bonds that are allowed to change are treated separately. The time complexity of bonded forces

```
L1:    DO N = 1, NSTEPS
         Regenerate non-bonded list if required
         ...
C        Bonded Force Calculations
L2:      DO I = 1, NBONDS
           Calculate bond energy between atoms IB(I) and JB(I)
         END DO
L3:      DO I = 1, NANGLES
           Calculate angle energy of atoms IT(I), JT(I), and KT(I)
         END DO
         ...
C        Non-Bonded Force Calculation
L4:      DO I = 1, NATOMS
           DO J = INBLO(I)+1, INBLO(I+1)
             Calculate non-bonded force between atoms I and JNB(J)
           END DO
         END DO
         ...
         Integrate Newton's equations and update atom coordinates
       END DO
```

Figure 1: Dynamics Simulation Code from CHARMM

calculations is linear to the number of atoms because each atom has a finite number of bonds with other atoms.

Non-bonded forces are due to the van der Waals interactions and electrostatic potential between all pairs of atoms. The time complexity of non-bonded forces calculations is $O(N^2)$ because each atom interacts with all other atoms in the system. When simulating large molecular structures, CHARMM approximates this calculation by ignoring all interactions beyond a certain cutoff range. This approximation is done by generating a non-bonded list, array JNB, which contains all pairs of interactions within the cutoff range. The non-bonded list determines the pattern of data accesses in the non-bonded force calculation loop. The spatial positions of the atoms change after each time step, consequently, the same set of atom interactions may not be appropriate for subsequent time steps. Hence, a new non-bonded list may need to be generated. However, in CHARMM, users have control over non-bonded list regeneration frequency. If users do not specify the non-bonded list regeneration frequency, CHARMM contains a heuristics test that can be used to decide when it is necessary to regenerate the list.

3 CHAOS Runtime Library

The CHAOS library is a set of software primitives that are designed to efficiently handle irregular problems on distributed memory systems. It is a superset of the PARTI library [7]. These primitives have been designed to ease the implementation of computational problems on parallel architecture machines by relieving users of low-level machine specific issues. The design philosophy has been to leave the original (sequential) source codes essentially unaltered, with the exception of the introduction of various calls to the CHAOS primitives which are embedded in the codes at the appropriate locations. These primitives allow the distribution and retrieval of data from the numerous processor local memories.

In distributed memory systems, arrays can be partitioned among local memories of processors. These partitioned arrays are called *distributed arrays*. Users can either partition arrays regularly, e.g. in block or cyclic, or specify irregular mapping of arrays to processors. The distributions of data arrays have significant impact on performance because they determine the patterns of communication between processors. It is frequently advantageous to partition arrays in appropriate irregular distributions. CHAOS provides primitives to map arrays onto processors and uses *translation tables* to describe the distributions of distributed arrays.

CHAOS also provides a set of primitives that carry out optimizations at runtime to reduce both the number of messages and the volume of interprocessor communication. The runtime primitives examine the data accesses made by each processor, calculate what off-processor data have to be fetched and where the data will be stored, and then generate communication schedules to specify the patterns of communication for exchanging data. Once the communication schedules are ready, efficient gather and scatter routines can be used to exchange data between processors.

4 Parallelization

In distributed memory machines the data and the computational work must be partitioned among individual processors. The criteria for this partitioning is to reduce the volume of interprocessor data communication and also to ensure good load balance. This corresponds to partitioning data and iterations, such as atoms, bonded force calculations, and non-bonded force calculations, and then assigning them to processors. Therefore, the parallelization efforts consist of four parts:

- Data Partition – partitioning atoms over processors,
- Iteration Partition – partitioning bonded and non-bonded force calculations,
- Loop Preprocessing – translating indexes from global to local and generating communication schedules, and

```
C     Perform Preprocessing
      Partition atoms (residues) and iterations, and generate a schedule
      ...
C     Calculate Forces
L1:   DO N = 1, NSTEPS
        Regenerate non-bonded list if required
        Generate a new schedule if non-bonded list is updated
        ...
C       Gather Off-Processor Data
        Gather X, Y, and Z
L2:     Calculate bonded forces
        ...
L3:     Calculate non-bonded forces
        ...
C       Scatter Off-Processor Data
        Scatter and accumulate DX, DY, and DZ
C       Calculate New Coordinates
        Integrate Newton's equations and update atom coordinates
      END DO
```

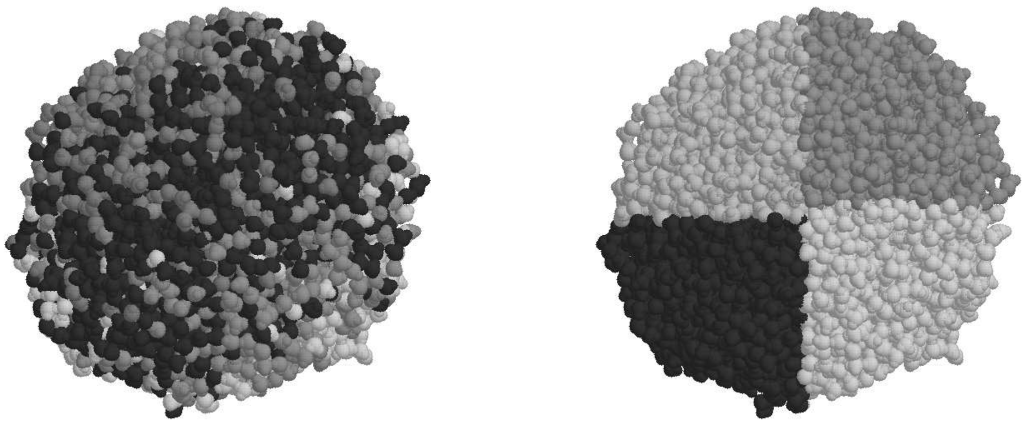
Figure 2: Structure of Parallelized CHARMM

- Communication Optimization – reducing both the number and volume of interprocessor communication messages.

Figure 2 shows the parallelized version of the loop structure of CHARMM, as shown in Figure 1, after the parallelizing steps.

4.1 Data Partition

The way in which the atoms are numbered frequently does not have a useful correspondence to the interaction pattern of the molecules. Naive data distributions such as BLOCK or CYCLIC may result in a high volume of communication and poor load balance. In molecular dynamics programs, bonded interactions occur only between atoms in close proximity to each other and non-bonded interactions are excluded beyond a certain cutoff range. Therefore, it is reasonable to assign atoms that are close to each other to the same processor. Additionally, the amount of computation associated with an atom depends on the number of atoms with which it interacts. Hence, data partitioners such as recursive coordinate bisection (RCB) and recursive inertial bisection (RIB), which use spatial information as well as computational load, are good candidates to partition atoms over processors. RCB and RIB divide atoms into two subdomains based on positions of atoms and moment of inertia of the simulated system respectively, and the same strategy is applied to the subdomains recursively until $P = 2^k$ subdomains is



(a) BLOCK Distribution (b) Irregular Distribution

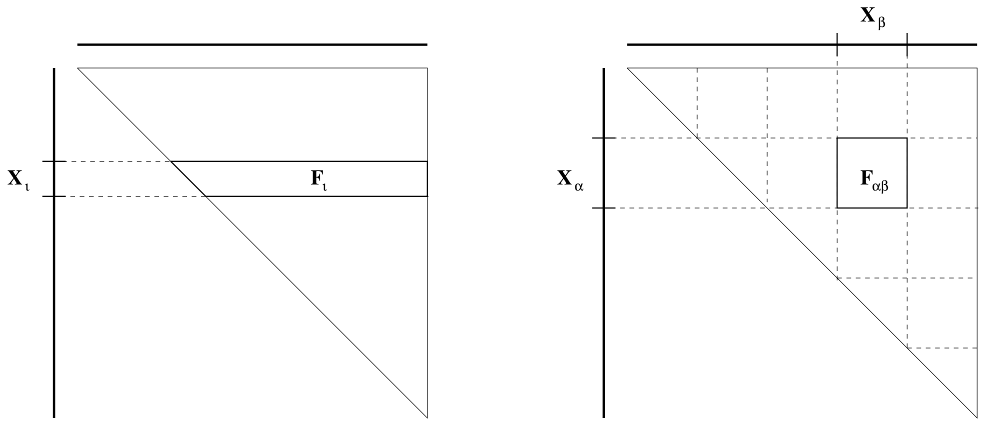
Figure 3: Distributions of Hydrated Carboxy-Myoglobin (MbCO) on 4 Processors

obtained. Figure 3 depicts two distributions of hydrated carboxy-myoglobin (MbCO) on 4 processors: (a) a BLOCK distribution and (b) an irregular distribution generated by a RCB partitioner. However, since CHARMM groups sets of atoms interconnected by bonds into electrostatic groups (usually neutral) and provides functions to calculate interactions between groups, the finest granularity of data partitioning is the group instead of the atom as shown in Figure 3.

Bonded force calculations consume about 1% of the total execution time of energy calculation, while non-bonded calculations consume over 90%. Hence, balancing the computational load due to non-bonded calculations is of primary concern. The computation work of non-bonded force calculation for each atom is roughly proportional to the size of the non-bonded list for the atom. Therefore, the size of the non-bonded list for each atom is a good estimate of workload. Partitioners examine the position and the estimated workload of each atom and distribute atoms onto processors so that interprocessor communication is minimal and load is balanced. Once the distribution of atoms is determined, all data arrays that are associated with atoms are distributed identically.

4.2 Iteration Partition

Once atoms are partitioned, the distribution is used to decide how loop iterations of energy calculation are partitioned over processors. Each loop index of bonded force calculations is assigned to the processor that owns the majority of data array elements accessed in that iteration. If the choice of processor is



(a) Atom Decomposition

(b) Force Decomposition

Figure 4: Decomposition of Non-Bonded Force Matrix F

not unique, the processor with the lowest computational load is chosen to maintain load balance. The partitioning of loop interactions for bonded force calculations does not impact performance much, but the partitioning for non-bonded force calculations impacts significantly.

Since non-bonded forces are the interactions between all pairs of atoms, the interactions can be represented by an $N \times N$ force matrix F, where N is the number of atoms in the system. Every non-zero element F_{ij} of the force matrix represents the force on atom i due to atom j. This matrix F is skew-symmetric due to Newton's third law, $F_{ij} = -F_{ji}$. Therefore, only the upper triangular matrix is required to calculate the non-bonded forces. Since the non-bonded force calculations can be represented by force matrix F, partitioning iterations of non-bonded force calculations is equivalent to partitioning force matrix F. Currently, two different iteration partitioning approaches, *atom decomposition* and *force decomposition*, have been implemented.

4.2.1 Atom Decomposition

In the force matrix F, row i represents the list of atoms that interact with atom i, i.e. row i is the non-bonded list of atom i. Hence, one straightforward way to partition iterations of non-bonded force calculations is to assign the non-bonded list of each atom to the processor that owns the atom. That is, each row F_i of the force matrix F is assigned to the processor that owns atom i. This is equivalent to distributing the rows of F onto processors. Figure 4(a) shows the partitioning of force matrix F. Row F_i is assigned to processor p that owns atom X_i. Processor p requires the coordinates of atom X_i and atoms X_{i+1} to X_N to compute the non-bonded force f_i.

Each processor has N/P atoms where N is the number of atoms and P is the number of processors. Since the non-bonded forces are long-range interactions, each atom interacts with all other $N-1$ atoms. Hence, each process has to access close to $N - N/P$ off-processor atoms, i.e. the order of the communication cost for each processor is $O(N)$. Therefore, this row-based decomposition will cause $O(P \times N)$ off-processor accesses. The communication overhead increases when the number of processors increases. A better decomposition scheme is required to reduce the communication overheads.

4.2.2 Force Decomposition

Force decomposition is a block-wise decomposition of the non-bonded force matrix rather than a row-wise decomposition as used in the atom decomposition scheme. The communication cost for each processor reduces from $O(N)$ to $O(N/\sqrt{P})$. The assignment of sub-blocks of force matrix F to processors is depicted in Figure 4(b). The sub-block $F_{\alpha\beta}$ of F is assigned to one processor, say $P_{\alpha\beta}$, where X_α and X_β are subsets of atoms in the system. Processor $P_{\alpha\beta}$ will require coordinates of atoms X_α and X_β to calculate the partial non-bonded force $f_{\alpha\beta}$. The non-bonded force f_α can then be calculated by combining the partial forces $f_{\alpha i}$, where i = 1, number of sub-blocks in row α.

Figure 4(b) presents a regular force decomposition of force matrix F. Even numbers of atoms are assigned to processors. This regular decomposition will cause load imbalance, since some processors have only half of the computations and some are idle. In order to improve load balance, an irregular force decomposition algorithm is implemented in two steps.

Step 1 Processors are divided into groups and a spatial partitioner is applied to distribute atoms irregularly to the processor groups so that interprocessor communication is minimized and computation load is balanced. This step is equivalent to applying the atom decomposition algorithm to partition force matrix F over processor groups.

Step 2 Each subset of the force matrix is then partitioned evenly over processors in the same group.

Table 1 compares the partitioning results of regular and irregular force decomposition schemes for the benchmark case, MbCO + 3830 water molecules (total 14026 atoms), that will be used in Section 5. The size of the non-bonded list is 6700818 at the beginning of simulation. Table 1 lists the maximum and average sizes of non-bonded lists assigned to processors and the number of idle processors.

Figure 5 depicts an irregular force decomposition of matrix F of 16 atoms over four processors. At step 1, processors are divided to two groups, processor 1 and 2 in group P_{12} and processors 3 and 4 in P_{34}. The force matrix F in Figure 5(a) is partitioned irregularly to two sub-matrices, F_{12} and F_{34}. Then sub-matrix F_{12} is assigned to processor group P_{12}, and F_{34} to P_{34}, as shown in Figure 5(b). Step 2

Table 1: Regular vs. Irregular Force Decomposition

		Regular				Irregular			
Processors		16	32	64	128	16	32	64	128
Non-Bonded	Max.	1203882	1003341	704616	473341	420214	210490	106916	54344
List	Avg.	418801	209400	104700	52350	418801	209400	104700	52350
Idle Processors		6	12	28	56	0	0	0	0

further partitions sub-matrices F_{12} and F_{34} and assigns them to processors. Figure 5(c) shows the final result of the irregular force decomposition.

4.3 Loop Preprocessing

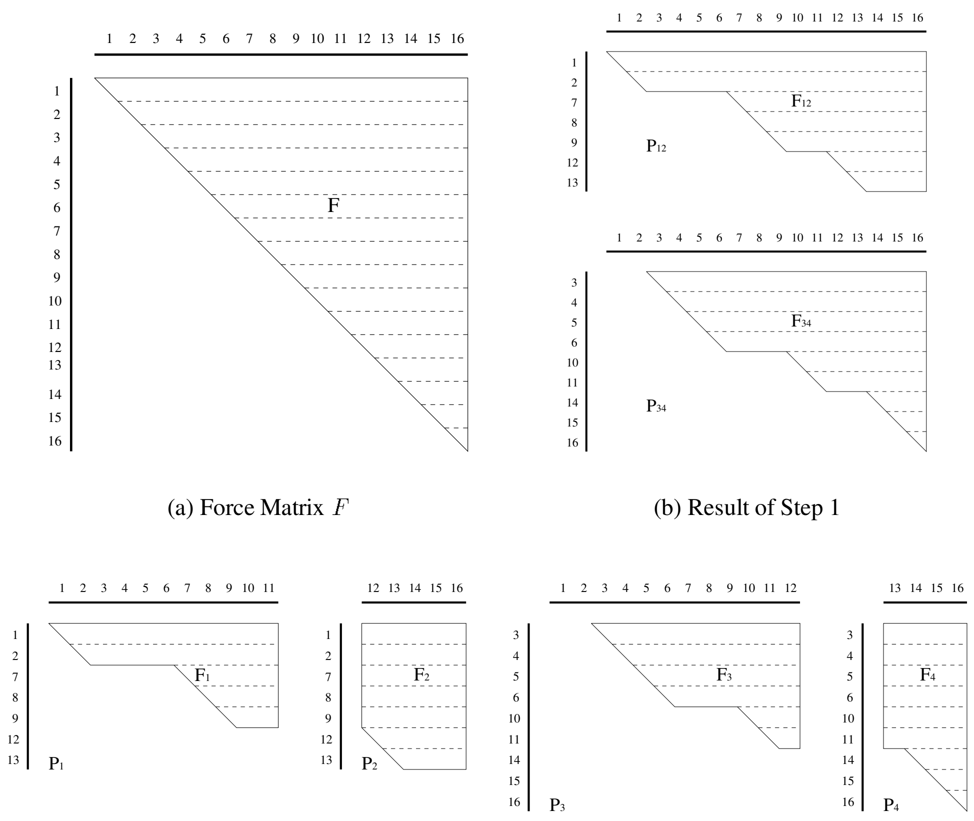
Since arrays are distributed over processors, preprocessing has to be carried out to calculate what off-processor elements need to be fetched and where the elements will be stored once they are received. The preprocessing (1) examines the indirection arrays of both bonded and non-bonded force calculations, (2) translates the indexes from global to local, and (3) generates a communication schedule for gathering and scattering off-processor data.

Indirection arrays for bonded force calculation loops remain unchanged while the non-bonded list adapts during computation. Hence, preprocessing for bonded force calculation loops need not be repeated, whereas it must be repeated for non-bonded force calculation loops whenever the non-bonded list changes. CHAOS provides *hash tables* and *stamps* to handle codes with adaptive indirection arrays. While building schedules, each indirection array is hashed with a unique time stamp. The hash table is used to remove any duplicate off-processor references. When the non-bonded list is regenerated, non-bonded list entries in the hash table are cleared with the corresponding stamp. Then the stamp can be reused and the new non-bonded list entries are hashed with the reused stamp.

4.4 Communication Optimization

In CHARMM the same data are accessed by several consecutive loops. The coordinates of atoms are used in both bonded and non-bonded force calculation loops. The atom coordinates are only updated at the end of each time step. Hence, all of the off-processor coordinates needed for the calculation can be obtained at the beginning of the time step. This makes it advantageous to perform computations without reading the same data more than once.

The CHAOS primitives can be used to track and reuse off-processor data copies. The primitives use hash tables to omit duplicate off-processor data references. In this implementation, one single



(a) Force Matrix F (b) Result of Step 1

(c) Result of Irregular Force Decomposition

Figure 5: Irregular Force Decomposition

communication schedule was generated for the loops in the dynamic energy calculation routine based on the content of the hash table. This schedule was used to gather off-processor data at the beginning and scatter the results at the end of each time step of energy calculation.

5 Experiments and Results

This section describes the experiments performed and presents the corresponding results. A number of different experiments were performed for data and iteration partitioning. The results are presented to show how the performance scales from small to large numbers of processors. All the timing results, except the load balance index, are presented in seconds.

The performance of molecular dynamics simulations was studied with a benchmark case, MbCO + 3830 water molecules (total 14026 atoms), on the Intel iPSC/860. It ran for 1000 steps with 40

Table 2: Communication Overheads

Number of Processors	16	32	64	128
BLOCK	–[1]	–	293.9	309.9
CYCLIC	324.4	363.5	365.5	396.4
Weighted BLOCK	425.8	436.3	497.5	572.6
RCB	–	155.2	166.8	198.8
Weighted RCB	147.1	159.8	181.1	219.2

non-bonded list updates. The cutoff for non-bonded list generation was 14 Å. The non-bonded list was regenerated 40 times during the simulation. The recursive coordinate bisection (RCB) partitioner was used to partition atoms.

5.1 Performance

In this section, the impact on performance of different data and iteration partitioning approaches is presented.

Communication Overheads

Distribution of data arrays has significant impact on performance because it determines the patterns of communication between processors and, possibly, load balance. Several regular and irregular data partitioning methods have been implemented to compare the communication overheads. Table 2 presents average communication times of different data partitioning methods from 16 to 128 processors. Atom decomposition was used as the iteration partitioning algorithm.

Both BLOCK and CYCLIC are regular distributions and partition arrays evenly over processors. BLOCK divides an array into contiguous chunks of size N/P and assigns one block to each processor, whereas CYCLIC specifies a round-robin division of an array and assigns every P^{th} element to the same processor. Table 2 shows that both BLOCK and CYCLIC do not exploit locality and, therefore, cause higher communication overheads. Weighted BLOCK divides an array into contiguous chunks with different sizes so that each chunk would have the same amount of computational work. The communication cost of weighted BLOCK is higher than that of BLOCK distribution, but weighted BLOCK distribution has better load balance. RCB and weighted RCB are irregular distributions that partition atoms based on coordinates. Both distributions exploit the locality of data accesses, and hence have lower communication costs.

Load Balance

[1]Not available due to memory limitation of iPSC/860

Table 3: Load Balance Indexes

Number of Processors	16	32	64	128
BLOCK	–	–	3.13	3.08
CYCLIC	1.08	1.12	1.21	1.28
Weigh. BLOCK	1.05	1.07	1.10	1.12
RCB	–	1.66	2.35	2.38
Weigh. RCB	1.01	1.04	1.06	1.08

Table 3 presents the load balance indexes of different data partitioning algorithms. The *load balance index* is calculated as

$$\frac{(\max_{i=1}^{n} computation\ time\ of\ processor\ i) \times (number\ of\ processors\ n)}{\sum_{i=1}^{n} computation\ time\ of\ processor\ i}$$

Neither BLOCK nor CYCLIC distributions consider computation load of each element when distributing data arrays over processors, and hence both cause severe load imbalance. Weighted BLOCK divides arrays into different sizes of blocks based on the computation load of each element so that good load balance is obtained. The irregular distribution generated by weighted RCB achieves good load balance because the weighted version of RCB assigns nearly the same amount of computation load to each processor.

Performance

Table 4: Performance of Parallel CHARMM (Atom Decomposition)

Number of Processors	1	16	32	64	128
Execution Time	74595.5[2]	4356.0	2293.8	1261.4	781.8
Computation Time	74595.5	4099.4	2026.8	1011.2	507.6
Communication Time	0	147.1	159.8	181.1	219.2
Load Balance Index	1.00	1.03	1.05	1.06	1.08

Table 4 and Table 5 present the performance of CHARMM when atom decomposition and force decomposition algorithms are applied for iteration partitioning, respectively. The execution time includes the energy calculation time and communication time of each processor. The computation time is the average of the computation time of dynamics simulations over processors; the communication time is the average communication time. As predicated in Section 4.2, force decomposition has lower communication overheads than atom decomposition. The results also show that CHARMM scales well

[2]Estimate (based on model systems due to memory limitations with 1 node) [4]

Table 5: Performance of Parallel CHARMM (Force Decomposition)

Number of Processors	1	16	32	64	128
Execution Time	74595.5	4433.6	2296.7	1239.0	750.4
Computation Time	74595.5	4240.0	2096.8	1039.8	518.9
Communication Time	0	134.8	122.3	137.5	188.1
Load Balance Index	1.00	1.01	1.04	1.06	1.08

and that good load balance is maintained up to 128 processors.

5.2 Preprocessing Overheads

Data and iterations partitioning, remapping, and loop preprocessing, must be done at runtime. Preprocessing overheads of the simulation are shown in Table 6. The data partition time is the execution time of RCB. All arrays associated with atoms have to remapped and redistributed when the new distribution of atoms is known. The remapping and preprocessing time shows the overhead. After partitioning atoms, the non-bonded list is regenerated. This non-bonded list regeneration was performed because atoms were redistributed over processors and it was done before dynamics simulation occurred. In table 6, the regeneration time is shown as non-bonded list update time.

During simulation, non-bonded list was regenerate periodically. When the non-bonded list was updated, the schedule must be regenerated. The schedule regeneration time in Table 6 gives the total schedules regeneration time for 40 non-bonded list updates. This overhead was included in the computation times in Table 4 and Table 5. By Comparing these numbers to those in Table 5, it can be observed that the preprocessing overhead is relatively small compared to the total execution time.

Table 6: Preprocessing Overheads

Number of Processors	16	32	64	128
Data Partition	0.27	0.47	0.83	1.63
Non-bonded List Update	7.18	3.85	2.16	1.22
Remapping and Preprocessing	0.03	0.03	0.02	0.02
Schedule Generation	1.31	0.80	0.64	0.42
Schedule Regeneration ($\times$40)	43.51	23.36	13.18	8.92

5.3 Scalability

The scalability of the parallel CHARMM was tested on Intel Delta. Table 7 shows the performance of CHARMM from 64 to 512 processors. The execution times and load balance indexes in the table

Table 7: Performance on Intel Delta

	Atom Decomposition				Force Decomposition			
Number of Processors	64	128	256	512	64	128	256	512
Execution Time	1226.2	712.3	504.6	458.3	1187.8	687.3	460.7	397.4
Load Balance Index	1.08	1.11	1.20	1.48	1.05	1.07	1.09	1.15

Table 8: Performance on Cray T3D (PVM)

Number of Processors	16	32	64	128
Execution Time	2973.1	1572.3	790.6	544.5
Computation Time	2882.8	1449.5	637.6	318.6
Communication Time	52.6	69.6	113.4	204.3
Load Balance Index	1.01	1.04	1.06	1.06

demonstrate that CHARMM can achieve high performance on large numbers of processors and maintain good load balance.

Table 7 also demonstrates that the irregular force decomposition outperforms the atom decomposition by an increasing margin as the number of processors increases. Force decomposition is 3 percent faster than the atom decomposition on 64 processors, whereas force decomposition is 15 percent faster than atom decomposition on 512 processors.

5.4 Portability

The CHAOS library has been ported to several distributed memory systems, such as Cray T3D, IBM SP-2, and Intel Paragon, etc. Using these libraries, the parallel CHARMM code has been executed on the Cray T3D, IBM SP-2, and Intel Paragon. The performance results of (irregular force decomposition) CHARMM on Cray T3D, IBM SP-2, and Intel Paragon are presented in the Table 8, Table 9, and Table 10, respectively. For communication, Cray T3D uses the Parallel Virtual Machine (PVM) library, IBM SP-2 uses the Message Passing Library (MPL), and Intel Paragon uses the Paragon OSF/1 Message-Passing System Calls. The computation time on IBM SP-2 is only about half of the computation times on Intel Paragon and Cray T3D. In addition, the communication costs on Intel Paragon and Cray T3D are slightly worse than that on IBM SP-2 for comparable numbers of processors. The net effect is that IBM SP-2 is substantially faster when compared to the execution times on Intel Paragon and Cray T3D.

CHAOS can also be ported to distributed shared memory systems. For instance, a subset of the CHAOS runtime library has been ported to Cray T3D by using the SHMEM shared memory library[3]. Data

[3] A shared memory access library released by Cray Research, Inc. as part of the CRAFT compiler environment

Table 9: Performance on IBM SP-2 (MPL)

Number of Processors	16	32	64	128
Execution Time	1414.1[4]	759.4	413.1	338.1
Computation Time	1332.6	674.8	312.2	165.3
Communication Time	48.4	51.4	76.3	146.9
Load Balance Index	1.02	1.05	1.08	1.16

Table 10: Performance on Intel Paragon (Massage-Passing System Calls)

Number of Processors	16	32	64	128
Execution Time	3472.2	1863.4	972.5	579.1
Computation Time	3229.4	1629.2	778.9	391.1
Communication Time	180.3	165.4	154.5	165.6
Load Balance Index	1.02	1.04	1.05	1.06

on remote processors can be referenced directly via support circuitry on each node and interconnect network. The latency of each remote access is 1 to 2 microseconds. The performance of CHARMM on Cray T3D using SHMEM library is shown in Table 11. The low latency of remote access makes Cray T3D scale very well. The communication overhead goes down when the number of processors increases, since each processor accesses fewer remote data. Furthermore, the communication overhead on (shared-memory) Cray T3D is dramatically small when compared to the overheads on message-passing systems, which usually have large message startup latencies.

Table 11: Performance on Cray T3D (SHMEM Library)

Number of Processors	16	32	64	128
Execution Time	2647.7	1357.8	698.4	353.0
Computation Time	2591.1	1297.2	647.0	321.9
Communication Time	21.0	14.3	10.9	8.8
Load Balance Index	1.01	1.04	1.06	1.07

Figure 6 and Figure 7 compare the execution times and average communication times on various systems, respectively. Figure 6 depicts that IBM SP-2 is significantly faster than other systems on small numbers of processors. However, the computational performance of SP-2 is offseted by the communication overhead on 128 processors. The execution time of SP-2 on 128 processors is comparable to that

[4] All SP-2 timings were measured on the SP-2 with Power1 nodes (located at Argonne National Laboratory)

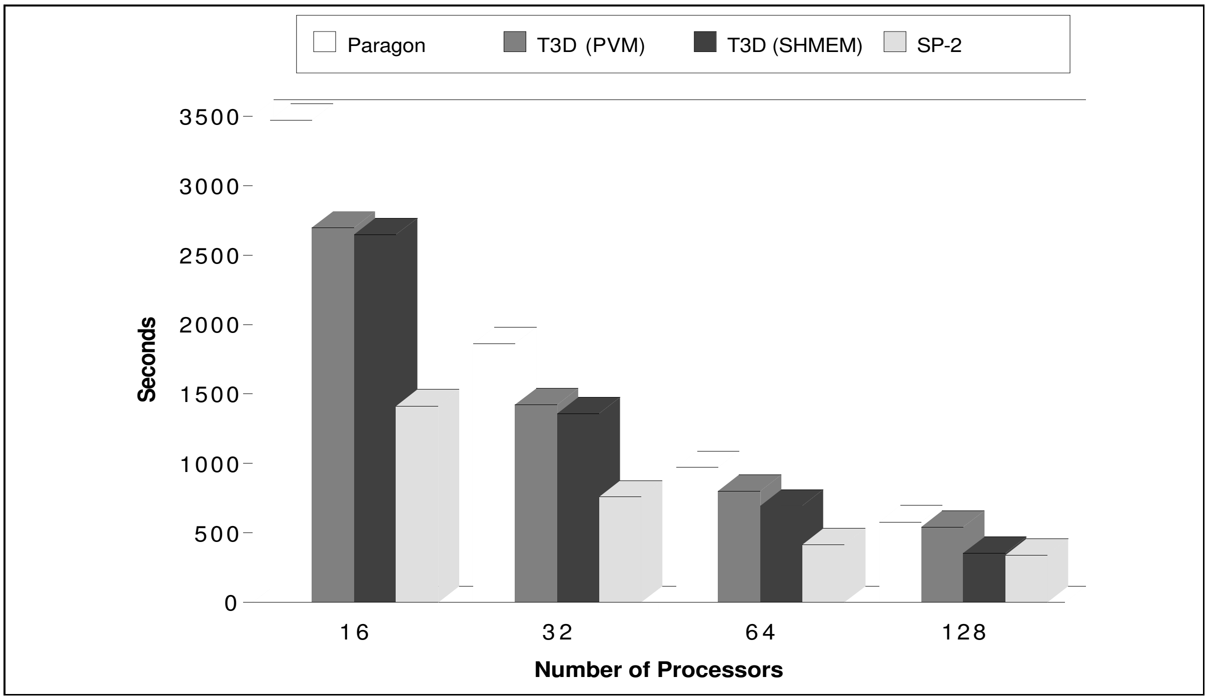


Figure 6: Execution Times

of (distributed shared-memory) Cray T3D. Similarly, Cray T3D (PVM) has better execution time than Intel Paragon on 16 processors, whereas the difference is only about 6% on 128 processors since Cray T3D suffers high communication latencies of PVM on large numbers of processors. Figure 7 shows how communication performance of each system scales from 16 to 128 processors. Intel Paragon has nearly the same communication times on different numbers of processors. Both IBM SP-2 and Cray T3D (PVM) incur more communication overheads as the number of processors increases, whereas the communication time of Cray T3D using SHMEM library goes down. Consequently, Cray T3D (SHMEM) outperforms other systems in communication performance by a greater margin as the number of processors increases. The communication overhead on T3D is slightly less than half of that on SP-2 on 16 processors, and it reduces to only 6% as the number of processors increases to 128 processors.

5.5 Comparison with Optimized Full Communication Methods

In comparing the distributed data versions with the full communication replicated data version of CHARMM [4], it is clear that distributed data versions can achieve somewhat better overall performance and that much larger systems can be simulated due to the reduced memory requirements of the distributed data methods. Nevertheless, the constraints and difficulties in developing a large and widely distributed computer program preclude the abandonment of the straightforward replicated data approach. The goals of these parallelization efforts have been to:

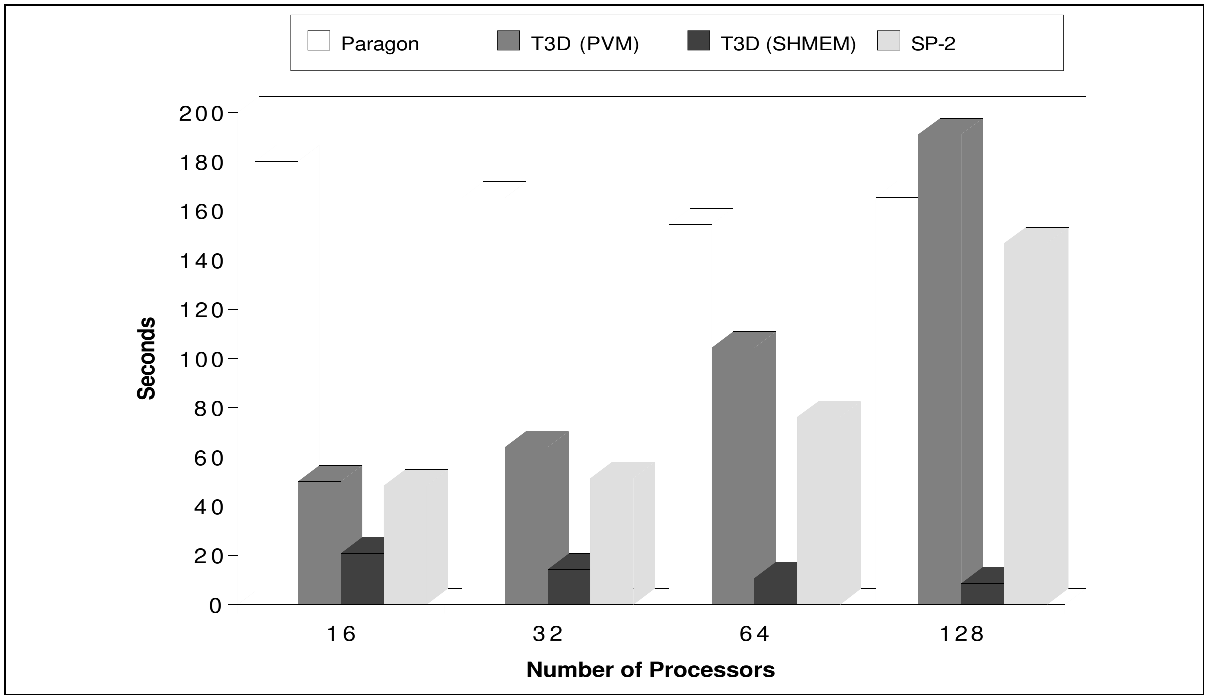


Figure 7: Communication Times

- develop a version of CHARMM where all features are functional, but not necessarily run in parallel,
- provide general and efficient parallelization for common tasks that are computationally intensive (molecular dynamics),
- remain input script compatible with CHARMM on scalar machines,
- remain source code compatible with CHARMM on scalar machines, and
- be able to function on a wide variety of hardware platforms.

For the myriad of CHARMM tasks that do not dominate the computational costs, the simple replicated data approach allows adequate performance with minimal development costs. The replicated data approach uses a minimal set of communication routines (vector global sum, vector distributed global sum, distributed vector global broadcast, read from node zero and broadcast, and simple timing routines) which can be highly optimized for each hardware system. The overall general development approach has been to allow for a variety of different parallel methods within a single working program, and to choose the most suitable approach for each task or sub-task.

6 Related Work

There has been considerable effort by researchers to exploit parallelism of molecular dynamics computations on various parallel architectures. Several parallel algorithms have been proposed and implemented on MIMD systems [4]. It has also been proposed and observed that spatial domain decomposition algorithms can be applied to achieve good load balance [8]. More recently there have been scalable algorithms created for large MIMD multiprocessor systems [6, 9].

Several widely used MD programs have been parallelized on MIMD computers. Clark et al. [9] applied spatial decomposition algorithms to develop EULERGROMOS, a parallelization of the GROMOS MD program. Brooks and Hodoščček implemented an efficient parallel version of CHARMM using the replicated data approach [4]. The parallelization effort of CHARMM done by Das and Saltz [10] was based on the distributed data approach. All data arrays associated with atoms were partitioned over processors in BLOCK distribution. This implementation did not achieve good performance and scalability because BLOCK distribution did not exploit locality.

Plimpton proposed the regular force decomposition algorithm for parallel short-range MD computation [6]. The algorithm was adapted in this paper to decompose force matrices irregularly in order to achieve good load balance.

CHAOS has been used to parallelize irregular scientific computation application programs, such as computational chemistry and computational fluid dynamics (CFD). CHAOS has also been used by several parallel compilers to handle irregular computations on distributed memory systems [11, 12].

7 Conclusions

This paper presented an implementation of the CHARMM program that scaled well on MIMD distributed memory machines. This implementation was done by using a set of efficient runtime primitives called the CHAOS runtime support library. Both the data decomposition and force decomposition algorithms were implemented and compared.

Although the original force decomposition algorithm partitioned atoms regularly over processors, the non-bonded list was distributed unevenly and hence severe load imbalance occurred. An irregular force decomposition algorithm was introduced and implemented in this paper. Processors were divided into groups and the atom decomposition algorithm was applied to distributed force matrix F onto processor groups. Each subset of the force matrix was then partitioned evenly over processors in the same group. This paper showed that the irregular force decomposition algorithm could achieve good performance and good load balance.

The performance results showed that the irregular force decomposition algorithm had lower communication overheads and better load balance on large numbers of processors. The irregular force decomposition outperformed the atom decomposition by an increasing margin as the number of processors increased. Force decomposition is 3 percent faster than the atom decomposition on 64 processors, whereas force decomposition is 15 percent faster than atom decomposition on 512 processors. The load balance indexes of the irregular force decomposition and atom decomposition algorithms on 512 processors were 1.15 and 1.48, respectively.

Data partitioning is important for parallelization because it determines the patterns of communication among processors and load balance. Appropriate data partitioning schemes can exploit the data locality and reduce interprocessor communication overheads.

The CHAOS library provides a set of runtime primitives that (1) couples partitioners to the application programs, (2) remaps data and partitions work among processors, and (3) optimizes interprocessor communications. The runtime primitives can efficiently handle both regular and irregular distributions of data and iterations. The implementation of the atom decomposition and irregular force decomposition algorithms was straightforward – simply embedding CHAOS runtime primitives in the original code at the appropriate locations. This implementation of parallel CHARMM presented an application of CHAOS that can be used to support efficient execution of irregular problems on distributed memory machines.

Acknowledgements

The authors thank Robert Martino and CBEL/DCRT for the use of the Intel iPSC/860 at NIH. The authors would like to thank Paul Messina and the Concurrent Supercomputing Consortium for the access of Intel Paragon and Touchstone Delta System, and Jet Propulsion Laboratory for the access of Cray T3D. The authors also thank Rick Stevens and Argonne National Laboratory for the use of IBM SP-2. Access to these facilities, except the NIH iPSC/860, was provided by the Center for Research on Parallel Computation. The authors would like to gratefully acknowledge Donna Meisel for careful proofreading of this manuscript.

References

[1] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[2] W. F. van Gunsteren and H. J. C. Berendsen. Gromos: Groningen molecular simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[3] P. K. Weiner and P. A. Kollman. Amber:assisted model building with energy refinement. a general program for modeling molecules and their interactions. *Journal of Computational Chemistry*, 2:287, 1981.

[4] B. R. Brooks and M. Hodoscek. Parallelization of charmm for mimd machines. *Chemical Design Automation News*, 7:16, 1992.

[5] Joel Saltz, Ravi Ponnusamy, Shamik D. Sharma, Bongki Moon, Yuan-Shin Hwang, Mustafa Uysal, and Raja Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and UMIACS, March 1995.

[6] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. Technical Report SAND91-1144, Sandia National Laboratories, May 1993.

[7] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[8] Scott B. Baden. Programming abstractions for run-time partitioning of scientific continuum calculations running on multiprocessors. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Los Angeles, California, December 1987. Also Tech Report LBL-24643, Mathematics Dept., Lawrence Berkeley Laboratory.

[9] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelizing molecular dynamics using spatial decomposition. In *Proceedings of the '94 Scalable High Performance Computing Conference*, pages 95–102, Knoxville, Tennessee, May 1994.

[10] R. Das and J. Saltz. Parallelizing molecular dynamics codes using the Parti software primitives. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 187–192. SIAM, March 1993.

[11] Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions in FORTRAN 90D/HPF compilers. Technical Report CS-TR-3268 and UMIACS-TR-94-57, University of Maryland, Department of Computer Science and UMIACS, May 1994. To appear in IEEE Parallel and Distributed Technology, Spring 1995.

[12] R. v. Hanxleden, K. Kennedy, and J. Saltz. Value-based distributions in Fortran D – a preliminary report. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993. Submitted to Journal of Programming Languages - Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines.