Experiences with Parallel N-Body Simulation

Pangfeng Liu, *Member*, *IEEE*, and Sandeep N. Bhatt

Abstract—This paper describes our experiences developing high-performance code for astrophysical *N*-body simulations. Recent *N*-body methods are based on an adaptive tree structure. The tree must be built and maintained across physically distributed memory; moreover, the communication requirements are irregular and adaptive. Together with the need to balance the computational work-load among processors, these issues pose interesting challenges and tradeoffs for high-performance implementation. Our implementation was guided by the need to keep solutions simple and general. We use a technique for implicitly representing a dynamic global tree across multiple processors which substantially reduces the programming complexity as well as the performance overheads of distributed memory architectures. The contributions include methods to vectorize the computation and minimize communication time which are theoretically and experimentally justified. The code has been tested by varying the number and distribution of bodies on different configurations of the Connection Machine CM-5. The overall performance on instances with 10 million bodies is typically over 48 percent of the peak machine rate, which compares favorably with other approaches.

Index Terms—N-body simulations, parallel processing, Barnes-Hut algorithm, adaptive tree structure, Peano-Hilbert space filling curve.

1 Introduction

Computational methods to track the motions of bodies which interact with one another, and possibly subject to an external field as well, have been the subject of extensive research for centuries. So-called "*N*-body" methods have been applied to problems in astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics. In this paper, we restrict attention to gravitational *N*-body simulation.

The problem is stated as follows: Given the initial states (position and velocity) of N bodies, compute their states at time T. The most common, and simplest, approach is to iterate over a sequence of small time steps. Within each time step, the acceleration on a body is approximated by the instantaneous acceleration at the beginning of the time step. The instantaneous acceleration on a single body can be directly computed by summing the contributions from each of the other $N-1$ particles. While this method is conceptually simple, vectorizes well, and is the algorithm of choice for many applications, its $O(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of particles.

Beginning with Appel [3] and Barnes and Hut [5], there has been a flurry of interest in faster algorithms. Experimental evidence shows that heuristic algorithms require far fewer operations for most initial distributions of interest, and within acceptable error bounds. Indeed, while there are pathological bad inputs for both algorithms, the number of operations per time step is $O(N)$ for Appel's method, and $O(N \log N)$ for the Barnes-Hut algorithm when the bodies are uniformly distributed in space, provided that certain control parameters are appropriately chosen.

Greengard and Rokhlin [16] developed the fast multipole method with $O(N)$ arithmetic complexity, which is accurate to any fixed precision. Sundaram [31] subsequently extended this method to allow different bodies to be updated at different rates; this reduces the arithmetic complexity over a large time period. Thus far, however, because of the complexity and overheads in the fully adaptive three-dimensional multipole method, the algorithm of Barnes and Hut continues to enjoy application in astrophysical simulations.

Several parallel implementations of the algorithms mentioned above have been developed over the years. Board et al. [12], [13] implemented the 3D adaptive fast multipole method on shared memory machines including the KSR. Zhao and Johnsson [38] describe a nonadaptive 3D version of Greengard's algorithm on the Connection Machine CM-2, and Singh et al. [29] implemented the adaptive method in two dimensions on the experimental DASH machine at Stanford. Finally, Nyland et al. implemented a 3D adaptive FMM with data-parallel methodology in a Proteus, an architecture-independent language [21], [22]. Pringle [24] implemented the FMM in both 2D and 3D on the Meiko Computer Surface, CS-1, a distributed memory parallel computer with explicit message-passing paradigm.

Salmon [26] implemented the Barnes-Hut algorithm, with multipole approximations, on message passing architectures including the NCUBE and Intel iPSC. Warren and Salmon [34], [35] report impressive performance from extensive runs on the 512 node Intel Touchstone Delta. Bhatt et al. applied the Barnes-Hut method to a multifilament fluid dynamic problem [15], [10], [9]. Singh et al. [28], [30], [29] also implemented the Barnes-Hut algorithm for the experimental DASH prototype. Warren et al. implemented the tree code using 16 Intel Pentinum Pro

- *P. Liu is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Ciayi, Taiwan, R.O.C. E-mail: pangfeng@cs.ccu.edu.tw.*
- *S.N. Bhatt is with Akamai Technologies, Inc., Cambridge, MA 02139.*

Manuscript received 29 Aug. 1996; revised 1 Feb. 2000; accepted 7 Feb. 2000. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100284.

```
For each time step:
  1. Build the BH tree.
  2. Compute centers-of-mass bottom-up.
  3. For each body
       • Start a depth-first traversal of the tree, truncating the
         search at internal nodes where the approximation is
         applicable.
       • Update the contribution of the node to the acceleration of
         the body.
  4. Update the velocity and position of each body.
```

Fig. 1. The Barnes-Hut algorithm.

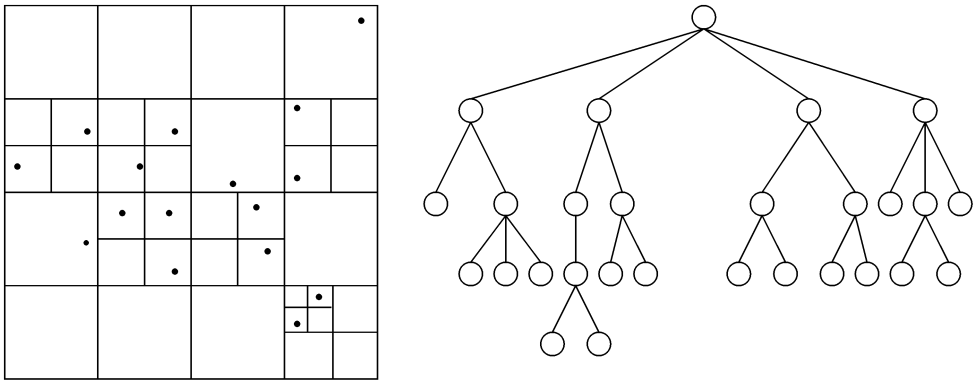

Fig. 2. A recursive partition in two dimension and its corresponding BH tree.

processors, and reported sustained performance in excess of one Gigaflop [33]. Blelloch [11] implemented and compared the Barnes-Hut algorithm, FMM, and the Parallel Multipole Tree Algorithm (PMTA) in NESL, a parallel programming language developed in CMU.

More recently, the Tree-Code-Particle-Mesh method (TPM) [37] combines multiple tree-code with a particle-mesh algorithm to achieve better solution resolution with low computation costs. For regions of dense particle distribution the tree code is used to achieve good resolution, and a fast particle-mesh algorithm is used in other regions for better computation performance. The code is written in PVM and is thus portable to a variety of computing platforms, and has been used for slices of a CDM universe in an 80 Mpc/h volume with a force softening length of 5 Kpc/h [37].

The remainder of this paper is organized as follows: Section 2 reviews the Barnes-Hut algorithm, the issues in parallel implementation, and recent related work. Section 3 describes our implementation and the reasons behind our design choices. Section 4 discusses experimental results from simulations, and Section 5 concludes.

2 The Barnes-Hut Algorithm

All tree codes exploit the idea that the effect of a cluster of bodies at a distant point can be approximated by a small number of initial terms of an appropriate power series. The Barnes-Hut algorithm uses a single-term, center-of-mass, approximation.

To organize a hierarchy of clusters, the Barnes-Hut algorithm, sketched in Fig. 1, first computes an oct-tree (BH-tree) partition of the three-dimensional box (region of space) enclosing the set of bodies. The partition is computed recursively by dividing the original box into eight octants of equal volume until each undivided box contains exactly one body. Fig. 2 is an example of a recursive partition in two dimensions. Alternative tree decompositions have been suggested [2], [14], [25]; the Barnes-Hut algorithm applies to these as well.

Each internal node of the BH-tree represents a cluster. Once the BH-tree has been built, the centers-of-mass of the internal nodes are computed in one phase up the tree, starting at the leaves. Step 3 computes accelerations; each body traverses the tree in a depth-first manner starting at the root. For any internal node, if the distance D from the corresponding box to the body exceeds the quantity R/θ, where R is the side-length of the box and θ is an accuracy parameter, then the effect of the subtree on the body is approximated by a two-body interaction between the body and a point mass located at the center-of-mass of the tree node. The tree traversal continues, but the subtree is bypassed.

Once the accelerations on all the bodies are known, the new positions and velocities are computed in Step 4. The entire process, starting with the construction of the BH-tree, is repeated for the desired number of time steps.

For convenience, we refer to the set of nodes which contribute to the acceleration on a body as the *essential* nodes for the body. Each body has a distinct set of essential nodes which changes with time.

One remark concerning distance measurements is in order. There are several ways to measure the distance between a body and a box. Salmon [26] discusses several alternatives in some detail. For consistency, we measure distances from bodies to the perimeter of a box in the L_∞ metric. This is a conservative choice, and for sufficiently small θ avoids the problem of "detonating galaxies" [26]. In our experiments, we use $\theta = 1$; this corresponds to $\theta = 0.5$ for the original Barnes-Hut algorithm.

The overhead in building the tree and traversing it while computing centers-of-mass and accelerations is negligible in sequential implementations. With 10,000 particles, more than 90 percent of the time is devoted to arithmetic operations involved in computing accelerations. Less than 1 percent of the time is devoted to building the tree. Thus, it is reasonable to build the BH-tree from scratch at each iteration.

2.1 Issues in Parallel Implementation

The Barnes-Hut algorithm provides sufficient parallelism; all bodies can, in principle, traverse the tree simultaneously. However, a good implementation must resolve a number of issues. To begin with, the bodies cannot all be stored in one node of a distributed-memory machine. With the bodies partitioned among the processors, the costs of building and traversing the BH-tree can increase significantly. In contrast, the time for arithmetic operations will, essentially, decrease linearly as the number of processors increases. This tension between the communication overhead and computational throughput is of central concern to both applications programmers and architects.

The challenges to developing high-performance code can be summarized as follows:

1. The BH-tree is irregularly structured and dynamic; as the tree evolves, a good mapping must change adaptively.
2. The data access patterns are irregular and dynamic; the set of tree nodes essential to a body cannot be predicted without traversing the tree. The overhead of traversing a distributed tree to find the essential nodes can be prohibitive unless done carefully.
3. The sizes of essential sets can vary tremendously between bodies; the difference often ranges over an order of magnitude. Therefore, it is not sufficient to map equal numbers of bodies among processors; rather, the work must be equally distributed among processors. This is a tricky issue, since mapping the nodes unevenly can create imbalances in the work required to build the BH-tree.

Finally, our aim is not simply to develop an efficient implementation of one algorithm. Rather, we seek techniques which apply generally to other N-body algorithms as well as other applications involving distributed tree structures.

2.2 Related Work

We sketch the important aspects of Salmon's thesis [26], which motivated us initially, as well as the more recent reports of Warren and Salmon [34], [35], and of Singh et al. [28], [30]. We also point out the differences of our techniques from these approaches.

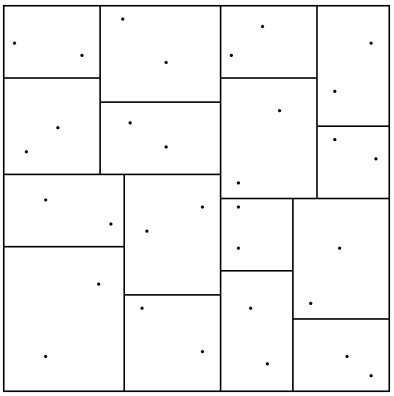

Fig. 3. A two-dimensional orthogonal recursive bisection.

Salmon [26] and Warren and Salmon [34] weight each body by the number of interactions in the previous time step. The volume enclosing the bodies is then recursively decomposed by orthogonal hyperplanes into regions of equal total weight. Fig. 3 shows the resulting decomposition, often called the *orthogonal recursive bisection*, ORB for short. When bodies move across processor boundaries, or their weights change, work imbalances can result. The ORB is recomputed at the end of each time step.

Each processor builds a local tree for its set of bodies which is later extended into a *locally essential tree*. The locally essential tree for a processor contains all the nodes of the global tree that are essential for the bodies contained within that processor. Once the locally essential trees have been built, the rest of the computation requires no further communication. Both implementations use quadrupole moments for higher accuracy.

The global tree is neither explicitly nor implicitly built. The process of building the locally essential trees requires nontrivial bookkeeping and synchronization. The bookkeeping is complicated by the "store-and-forward" nature of the process: when a processor receives information, it sifts through the data to retrieve any information that is locally essential, figure out what information must be forwarded, and discards the rest. The flow of information follows the dimension order of the hypercube.

We too use the ORB decomposition and build locally essential trees so that the final compute-intensive stage is not slowed down by communication. However, there are significant differences in implementation: 1) we build a distributed representation of a global tree in a separate phase, 2) the locally essential trees are built using a sender-driven protocol that is significantly simpler, more efficient, and network independent, 3) we update the ORB decomposition and global BH-tree incrementally only as necessary rather than recompute them at every iteration, and 4) the computation to update positions and velocities is vectorized to minimize time.

Since we carefully vectorized the final sequential stage, it was imperative that the overhead due to parallelization be as small as possible. Experimental results and comparisons are given in Section 4.

```
• Build local BH trees.
• For every time step do:
   1. Construct the BH-tree representation
      (a) Adjust node levels
      (b) Compute partial node values on local trees
      (c) Combine partial node values at owning processors
   2. Owners send essential data
   3. Calculate accelerations
   4. Update velocities and positions of bodies
   5. Update local BH-trees incrementally
   6. If the workload is not balanced update the ORB incrementally
```

Fig. 4. Outline of code structure.

More recently, Warren and Salmon [27], [35] reported a new algorithm that uses a different criterion for applying center-of-mass approximations. The new criterion considers not only the geometry location of a cell, but also the the detailed information of the particle distribution, including the actual radius of the cluster, the distance to the evaluation point of interest, and the multipole moments. As a result, the set of essential data for a particle cannot be determine a priori by the geometry information alone. Therefore, the new implementation [35], [36] does not build locally essential trees; instead, they construct an explicit representation of the BH-tree. Each body is assigned a key based on its position, and bodies are distributed among processors by sorting the corresponding keys. Besides obviating the need for the ORB decomposition, this also simplifies the construction of the BH-tree. However, without the computation, the stage is slowed down by communication; the latency is hidden by multiple threads to pipeline tree traversals and to update accelerations, but the program control structure becomes complicated and less transparent.

The DASH shared-memory architecture group at Stanford [28], [30] has investigated the implications of shared-memory programming for the Barnes-Hut algorithm as well as the two-dimensional adaptive fast multipole method. Each processor first builds a local tree; these are merged into a global tree stored in shared memory. Work is evenly distributed among processors by partitioning the bodies using a technique similar to [35].

The arguments in [28] about the advantages of shared-memory over message-passing implementations are based largely on comparisons to the initial implementations of Salmon [26] and Warren and Salmon [34]. Since our message-passing implementation is considerably simpler and more efficient, the import of the arguments of [30], [28] is less clear. For example, contrary to their claims, ORB can be implemented efficiently. Indeed, it is expensive to compute ORB from scratch at every time step, but it is simple to incrementally adjust the partition quickly. The same is true for the BH-tree. While shared-memory systems might ease certain programming tasks, the advantages for developing production-quality N-body codes are not entirely clear.

3 Implementation Overview

We separate control into a sequence of alternating computation and communication phases. This helps maintain simple control structure; efficiency is obtained by processing data in bulk. For example, up to a certain point, it is better to combine multiple messages to the same destination and send one long message. Similarly, it is better to compute the essential data for several bodies rather than for one at a time. Another idea that proved useful is sender-directed communication, send data wherever it might be needed rather than requesting it whenever it is needed. Indeed, without the use of the CM-5 vector units, we found that these two ideas kept the overhead minimal because of the parallelism.

Fig. 4 gives a high-level description of the code structure. Note that the local trees are built only at the start of the first time step. Steps 1b, 3, and 4 require no communication; Step 3 is the most time-consuming step.

3.1 Data Partitioning

We use orthogonal recursive bisection (ORB) to distribute bodies among processors. The space bounding all the bodies is is partitioned into as many boxes as there are processors, and all bodies within a box are assigned to one processor. At each recursive step, the separating hyperplane is oriented to lie along the smallest dimension; the intuition is that reducing the surface-to-volume ratio is likely to reduce the volume of data communicated in later stages. Each separator divides the workload within the region equally. When the number of processors is not a power of two, it is a trivial matter to adjust the division at each step accordingly.

The ORB decomposition can be represented by a binary tree, the ORB tree, a copy of which is stored in every processor. The ORB tree is used as a map which locates points in space to processors. Storing a copy at each processor is quite reasonable when the number of processors is small relative to the number of particles.

We chose ORB decomposition for several reasons. It provides a simple way to decompose space among processors, and a way to quickly map points in space to processors. This latter property is essential for sender-directed communication of essential data, for relocating

bodies which cross processor boundaries, and for our method of building the global BH-tree. Furthermore, ORB preserves data locality reasonably well[1] and permits simple load-balancing. Thus, while it is expensive to recompute the ORB at each time step [28], the cost of incremental load-balancing is negligible, as we will see in the next section.

The ORB decomposition is incrementally updated in parallel as follows: At the end of a time step, each processor computes the total number of interactions used to update the state of its particles. A tree reduction yields the number of operations for the subset of processors corresponding to each internal node in the ORB tree. A node is overloaded if its weight exceeds the average weight of nodes at its level by a small, fixed percentage, say 5 percent. It is relatively simple to mark those internal nodes which are not overloaded but one of whose children is overloaded; call such a node an *initiator*. Only the processors within the corresponding subtree participate in balancing the load for the region of space associated with the initiator. The subtrees for different initiators are disjoint so that nonoverlapping regions can be balanced in parallel.

At each step of the load-balancing step, it is necessary to move bodies from the overloaded child to the nonoverloaded child. This involves computing a new separating hyperplane; we use a binary search combined with a tree traversal on the local BH-tree to determine the total weight within a parallelpiped.[2]

We found that updating the ORB incrementally is cost-effective in comparison with either rebuilding it each time or with waiting for a large imbalance to occur before rebuilding.

3.2 Building the BH-Tree

Unlike the first implementation of Warren and Salmon [34], we chose to construct a representation of a distributed global BH-tree. An important consideration for us was to investigate abstractions that allow the applications programmer to declare a global data structure—a tree, for example—without having to worry about the details of distributed-memory implementation. For this reason, we separated the construction of the tree from the details of later stages of the algorithm. The interested reader is referred to [8] for further details concerning a library of abstractions for N-body algorithms.

3.2.1 Representation

We represent the global BH-tree as follows: Since the oct-tree partitions are oblivious of the input distribution, each internal node represents a fixed region of space. We say that an internal node is owned by the processor whose domain contains a canonical point, say the center of the corresponding region. The data for an internal node, the multipole representation, for example, is maintained by the owning processor. Since each processor contains a copy of the ORB-tree, it is a simple calculation to figure out which processor owns an internal node.

The only complication is that the region corresponding to a BH-node can be spanned by the domains of multiple processors. In this case, each of the spanning processors computes its contribution to the node; the owner accepts all incoming data and combines the individual contributions. This can be done efficiently when the combination is a simple linear function, as is the case with all tree codes.

3.2.2 Construction

Each processor first builds a local BH-tree for the bodies which are within its domain. At the end of this stage, the local trees will not, in general, be structurally consistent. The next step is to make the local trees be structurally consistent with the global BH-tree. This requires adjusting the levels of all leaves which are split by ORB lines. A similar process was developed independently in [28]; an additional complication in our case is that we build the BH-tree until each leaf contains a number, L, of bodies. Choosing L to be much larger than 1 speeds up the computation phase, but makes level-adjustment somewhat tricky.

The level-adjusting process can be described as a "*request-and-answer*" process. If a processor p has a leaf u that overlaps with multiple processor domains, p will send a request for u to all the processors overlapping with u. Upon receiving a request, a processor will send back an answer that describes the distribution of bodies within u in its domain. By receiving answers from other processors, p can know the distribution of bodies within u but outside its domain in order to adjust the level of u.

The level adjustment procedure also makes it easy to update the BH tree incrementally. We can insert and delete bodies directly on the local trees because we do not explicitly maintain the global tree. After the insertion/deletion within the local trees, level adjustment restores coherence to the implicitly represented distributed tree structure.

Once level-adjustment is complete, each processor computes the centers-of-mass and multipole moments on its local tree. This phase requires no communication. Next, each processor sends its contribution to an internal node to the owner of the node. Once the transmitted data have been combined by the receiving processors, the construction of the global BH-tree is complete. This method of reducing a tree computation into a one local step to compute partial values, followed by a communication step to combine partial values at shared nodes is generally a useful method.

3.3 Locally Essential Trees

Once the global BH-tree has been constructed, it is possible to start calculating accelerations. The naive strategy of traversing the tree, and transmitting data-on-demand, requires two-way communication (for request and reply), and may cause large communication overheads if not programmed carefully. For example, Warren et al. used a user-level multithreading programming technique to pipeline tree traversals, aggregate individual messages, and hind communication latency [35].

1. Clustering techniques which exploit the geometrical properties of the distribution will preserve locality better, but might lose some of the other attractive properties of ORB.

2. Each internal BH node will keep track of the total workload from the local particles in its domain, therefore we can quickly determine the workload within a parallelpiped by a tree traversal.

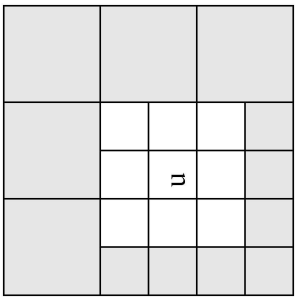

Fig. 5. The influence ring of a node u when $\theta = 1$.

We propose that we first construct the locally essential trees before computing the velocity. The owner of a BH-node computes the destination processors for which the node might be essential; this involves the intersection of the annular region of influence of the node, called *influence ring*, with the ORB-map. Fig. 5 shows the influence ring of a node u when $\theta = 1$. Those particles that are not within the influence ring are either too close to u to apply center-of-mass approximation, or far away enough to use u's parent's information, therefore u will be essential to only particles within its influence ring. As a result, each processor first collects all the information deemed essential to other nodes, and then sends long messages directly to the appropriate destinations. Once all processors have received and inserted the data received into the local tree, all the locally essential trees have been built.

The locally essential tree approach has its advantages and limitations. First, the processors send the essential data to where they are needed directly, without two-way request and reply overheads. In addition, after the processors collect the essential data, they can proceed their computation without any communication. However, as pointed out in [27], [35], sometimes it is difficult to determine a priori where a cluster will be needed by its geometry information alone. In other words, the "Multiple Acceptability Criterion" (MAC)—the test to determine whether a cluster is far away to apply approximation—requires more information than just the location of the cell. For example, if we measure the distance from a body to a cell as the *the minimum distance from the body to any point in the cell*, (MD MAC in [27]), then a processor can easily determine the influence area *without* knowing particle distribution inside the cell. However, this naive MAC will cause serious accuracy problem (e.g., "Detonating Galaxy" in [27]). On the other hand, the remedy of using a strict θ in MAC will cause unnecessary high computation costs. As a result, it will be more cost-effective to include the particle distribution into the MAC. For example, Barnes and Hut's method defines the distance from a body to a cell as the L_2 distance from the body to *the center of mass* of the cell (Barnes-Hut MAC in [27]). Salmon and Warren [27], [35] also introduced MAC that guarantees the error bound for each approximation, which requires the interaction distance, the cell size, and the multipole moments for better tradeoff between computation costs and accuracy.

It is possible for the locally essential tree approach to adapt to the complicated MAC that requires more than geometry information of the cell alone. For example, in our tree code the center of mass and the multiple moments are computed recursively with the representative approach in Section 3.2. As a result, the locally essential tree approach can be extended so that a cell, having all the necessary information from its parent and itself, can determine the region that will need this cell as essential data. However, there are several issues one must resolve. First, communication may still be necessary for a cell to determine to which region it should send the data, since the representative of its parent may not be in the same processor. Second, we will not be able to determine the influence region if we want to impose error tolerances which vary from particle to particle [27], [35]. Finally, with MAC other than MD MAC, it will be difficult to guarantee the caching performance in Theorem 1 that will be described in details next.

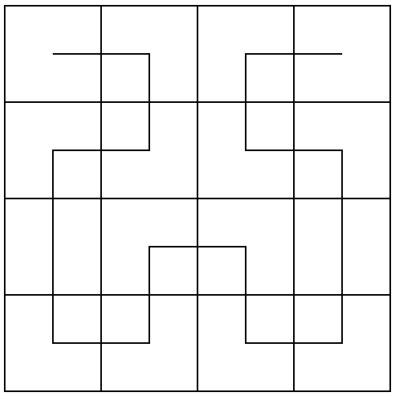

Fig. 6. Peano-Hilbert sequence.

3.3.1 Calculating Accelerations

The final phase to compute accelerations does not require any communication. In order to use the CM-5 vector units effectively, we calculate the accelerations of groups of bodies. Instead of measuring distances from bodies to BH-boxes, we instead measure distances between bounding boxes for groups of bodies and BH-boxes. This guarantees that the resulting calculations are at least as accurate as desired.

Grouping bodies does increase the number of calculations, but it also makes them more regular. More significant is the reduction in the time spent traversing the tree. This idea of grouping bodies was used earlier by Barnes [4].

A further reduction in tree traversal is possible by caching essential nodes. The key observation is that the set of essential nodes for two distinct groups that are close together in space are likely to have many elements in common. Therefore, we maintain a software cache for the essential nodes.

A judicious choice of caching strategy is necessary to ensure that cache maintenance overheads do not undermine the gains elsewhere. It is also important to order the different groups such that the total number of cache modifications is minimized. Our strategy is to pick a space-filling curve; the groups are chosen in their order along the space-filling curve. Fig. 6 shows an example using the Peano-Hilbert curve.

In what follows, we show that, under certain conditions, the number of cache modifications is asymptotically smaller

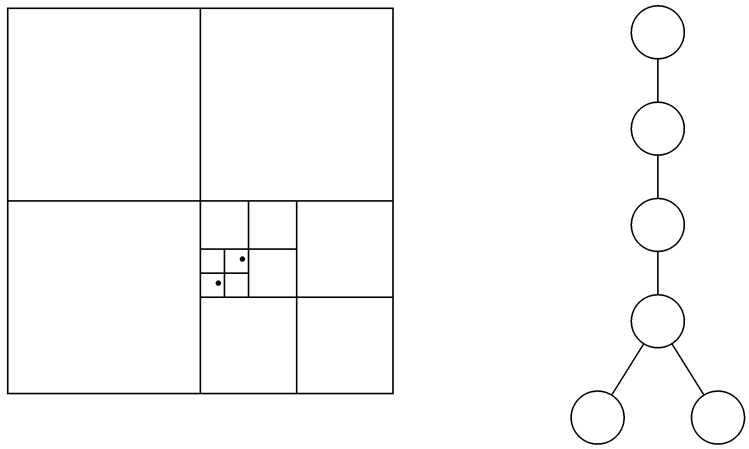

Fig. 7. A chain of two nearby bodies.

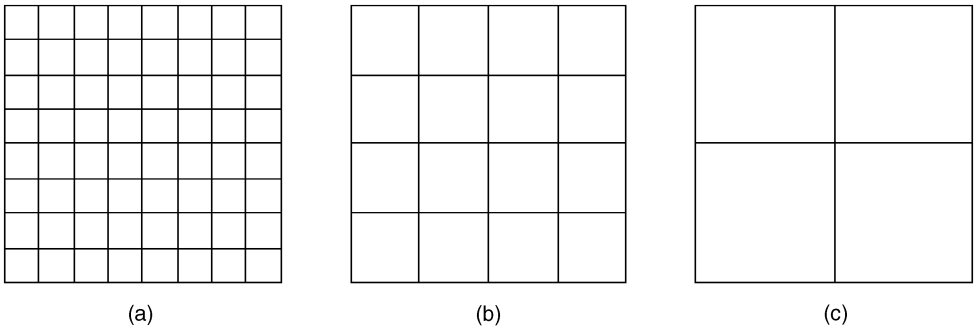

Fig. 8. Boundary lines in different levels. (a) 0-boundary. (b) 1-boundary. (c) 2-boundary.

than the number of two-body calculations when the velocity of particles is computed in their order along the space-filling curve. In particular, suppose there are N bodies in the system. The number of interactions computed by the Barnes-Hut algorithm is $\Omega(N \log N)$. We show that the number of cache modifications is bounded by $O(N)$ under recursive tree traversal, if either $\theta = 1$ or if the distribution of bodies is uniform. The general case, when $\theta \neq 1$ and the distribution of bodies is nonuniform, remains open. It is worthwhile to note that under the L_∞ metric we use, $\theta = 1$ is a reasonable choice for large systems.

Theorem 1. *When $\theta = 1$, the number of cache modifications under recursive tree traversal is $O(N)$, independent of the distribution of bodies.*

Proof. For ease of exposition, we give the proof for two dimensions; the extension to three dimensions is straightforward. The theorem follows from two observations. First, the number of times a tree node u enters the cache (denoted by t_u) is the number of times the traversal enters the influence ring of u. Therefore, the total number of cache modifications is the sum of t_u, over all tree nodes u.

Second, when θ is 1, the influence ring of any tree node u coincides with a BH-node boundary; the influence ring can be partitioned into twelve squares, as in Fig. 5, and each square corresponds to a possible BH-tree node. If the corresponding tree node of a square does not exist, then a leaf must cover this square, as well as some other adjacent squares. In any case, 12 tree nodes suffice to cover any influence ring.

Node u enters the cache each time the traversal enters the influence ring of u from outside, but this can happen at most 12 times. It follows that the number of cache updates is bounded by a constant factor times the size of the BH-tree.

Finally, the tree size is proportional to the number of bodies. The only complication is that two bodies that are very close together can form a long chain in the tree (see Fig. 7). However, this chain can be lumped into a single node because the tree nodes along the chain represent the same cluster. As a result, the size of BH-tree is $\Theta(N)$ and the theorem follows. □

Suppose the bodies are uniformly distributed in a unit square; we show that the number of cache modifications is bounded by $O(N)$ for any θ. We first define some notations. The unit square is uniformly refined until no more than a constant number of bodies is in any bottom level box (Fig. 8). Since the bodies are uniformly distributed, the number of bottom level boxes is $\Theta(N)$. A rectangle is $m \times n$ if its length and width consist of m and n bottom level boxes, respectively. A partition line is on *i-boundary* if it is between two $2^i \times 2^i$ BH-tree nodes.

Lemma 1. *Every $m \times n$ rectangle can be partitioned into $O(m + n)$ BH-tree nodes.*

Proof. We decompose the rectangle into layers of increasing width, with thinner layers on the outside. If a boundary line of the rectangle is not on 1-boundary, we cut a strip of 1×1 BH-nodes from the boundary so that the new boundary lines are on 1-boundary. After removing at

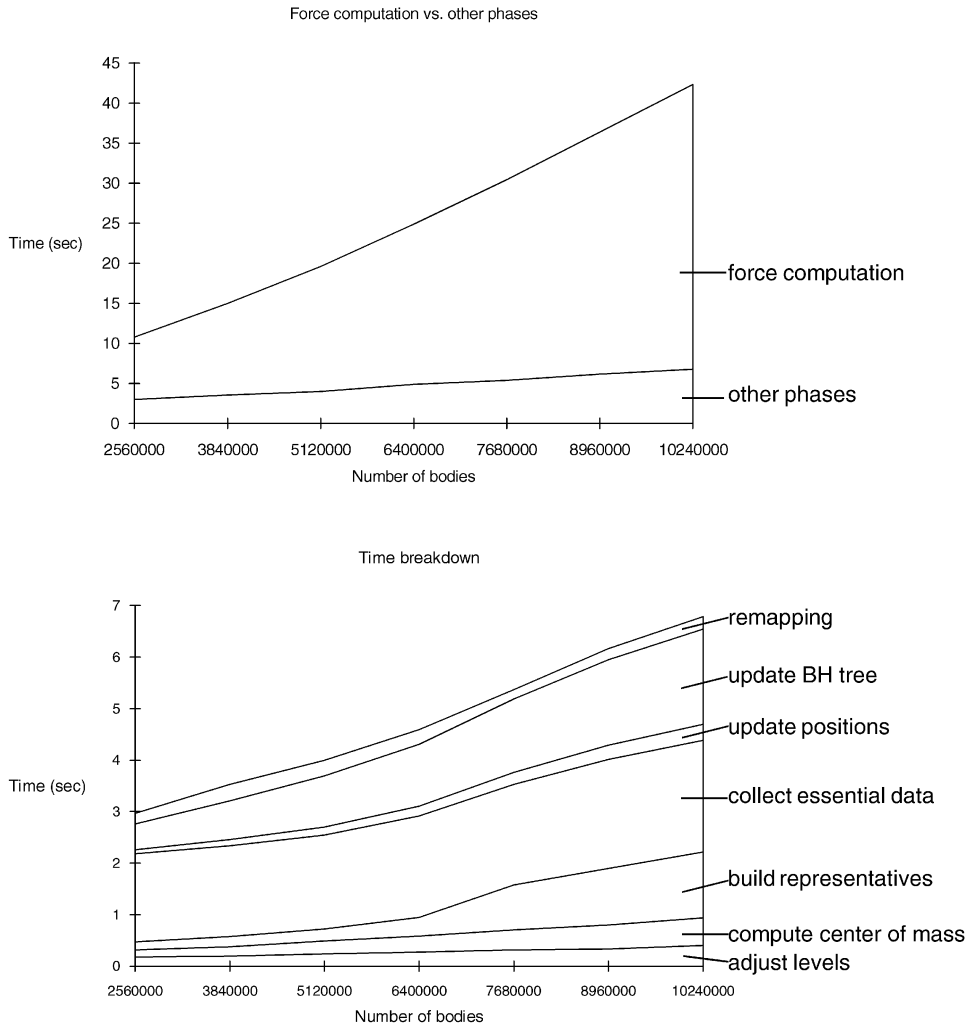

Fig. 9. Time breakdown for Plummer sphere.

most $m+n$ 1×1 boxes, the new area can be partitioned into 2×2 boxes. In general, the number of $2^i \times 2^i$ boxes removed at the ith iteration is $O(\frac{m+n}{2^i})$, and all the new boundary lines will be on i+1-boundary. Summing up, the total number of BH-nodes required to partition an $m \times n$ rectangle is $O(m+n)$. □

Theorem 2. *When the bodies are uniformly distributed, the number of cache modifications under recursive tree traversal is $O(N)$, independent of θ.*

Proof. Similar to the proof of Theorem 1, we bound the number of BH-nodes required to cover an influence ring. We extend the influence ring just enough to cover the bottom-level boxes that were partially covered. The influence ring of a BH-node u in level ℓ has $O(\frac{\sqrt{N}}{2^\ell})$ boxes on the perimeter, and can be partitioned into four rectangles of $O(\frac{\sqrt{N}}{2^\ell}) \times O(\frac{\sqrt{N}}{2^\ell})$. From Lemma 1, the number of tree nodes required to cover the influence ring of u is $O(\frac{\sqrt{N}}{2^\ell})$. Therefore, the number of cache modifications is $\sum_{\ell=1}^{\log_4 N} 4^\ell \frac{\sqrt{N}}{2^\ell} = O(N)$. □

The case of nonuniform distribution and arbitrary θ remains open. In this case, we can bound the total distance covered under a specific tree traversal. While this does not directly bound the number of cache modifications, the intuition is that keeping the average distance between consecutive bodies should keep the number of cache modifications small. In particular, we use a recursive tree traversal corresponding to the Peano-Hilbert curve in Fig. 6. The worst-case length of a space-filling curve for N bodies in the unit square is $\Theta(\sqrt{N})$ [23]. In fact, it has been established that the length of the Peano-Hilbert curve is always within a $O(\log N)$ factor of the optimal TSP tour in d-dimension [6], [7], [23].

Theorem 3. [23]. *Suppose that N bodies are distributed within the unit square (alternatively, the unit cube). Then the total distance covered by the Peano-Hilbert traversal is $O(\sqrt{N})$ ($O(N^{\frac{2}{3}})$).*

3.4 Reducing Communication Times

The communication phases can all be abstracted as the "all-to-some" problem. Each processor contains a set of messages; the number of messages with the same destination can vary arbitrarily. The communication pattern is irregular and unknown in advance. For example, level

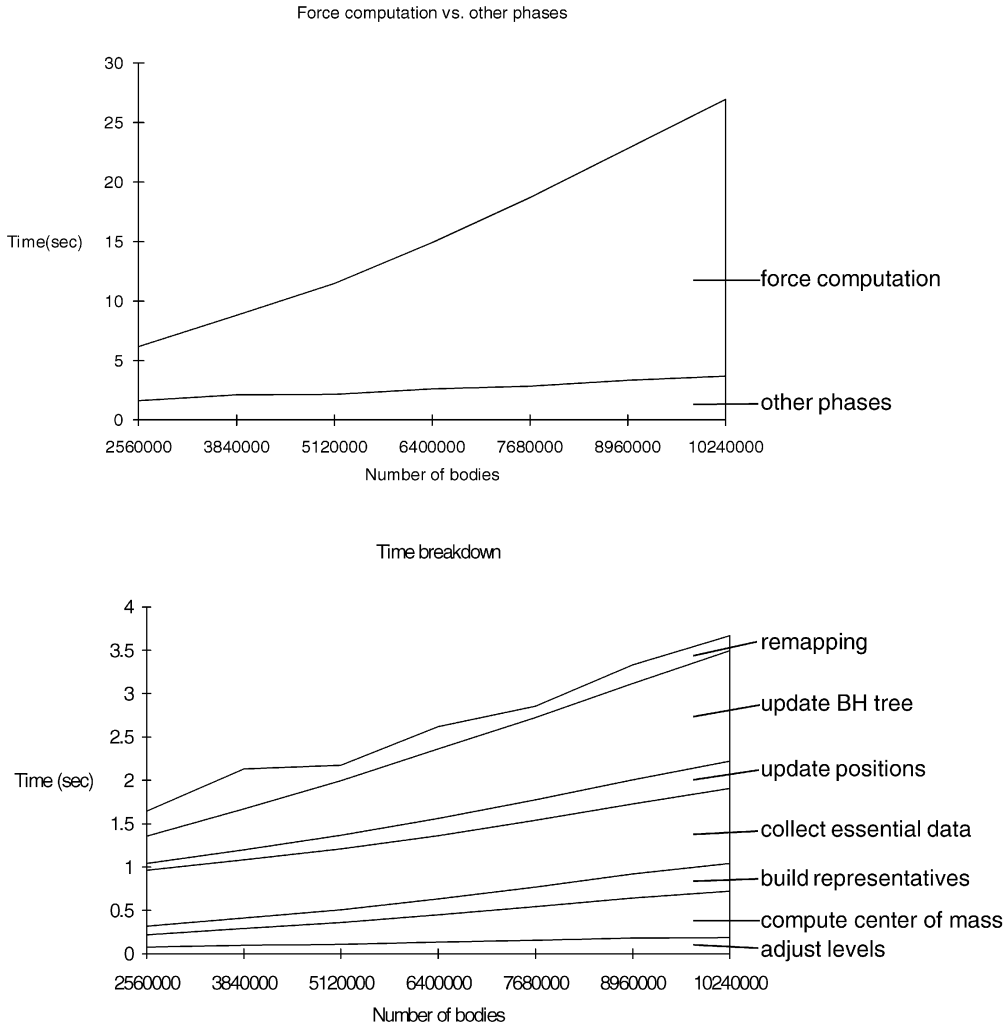

Fig. 10. Time breakdown for uniform distribution.

adjustment is implemented as two separate all-to-some communication phases. The phase for constructing locally essential trees uses one all-to-some communication.

The first issue is detecting termination: When does a processor know that all messages have been sent and received? The naive method of acknowledging receipt of every message, and having a leader count the numbers of messages sent and received within the system, proved inefficient.

A better method is to use a series of global reductions, the control network of the CM-5, to first compute the number of messages destined for each processor. After this, the send/receive protocol begins; when a processor has received the promised number of messages, it is ready to synchronize for the next phase.

We noticed that the communication throughput varied with the sequence in which messages were sent and received. As an extreme example, if all messages are sent before any is received, a large machine will simply crash when the number of virtual channels has been exhausted. In the CMMD message-passing library (version 3.0), each outstanding send requires a virtual channel [32] and the number of channels is limited.

Instead, we used a protocol which alternates sends with receives. The problem is thus reduced to ordering the messages to be sent. For example, sending messages in order of increasing destination address gives low throughput, since virtual channels to the same receiver are blocked. In an earlier paper [18], we developed the atomic message model to investigate this phenomenon. Consistent with the theory, we find that sending messages in random order worked best.

4 Experimental Results

This section describes experimental results we obtained from two sets of implementations. Both implementations are based on the same key ideas described in Section 3.

4.1 Connection Machine CM-5

Our platform is the Connection Machine CM-5E with SPARC vector units. All experiments reported here were run on a 256-node CM-5. Each processing node has 128 Mbytes of memory and can perform floating point operations at a peak rate of 160 Mflop/s. We use the Connection Machine CMMD library (version 3.0). The vector units are programmed in CDPEAC which provides

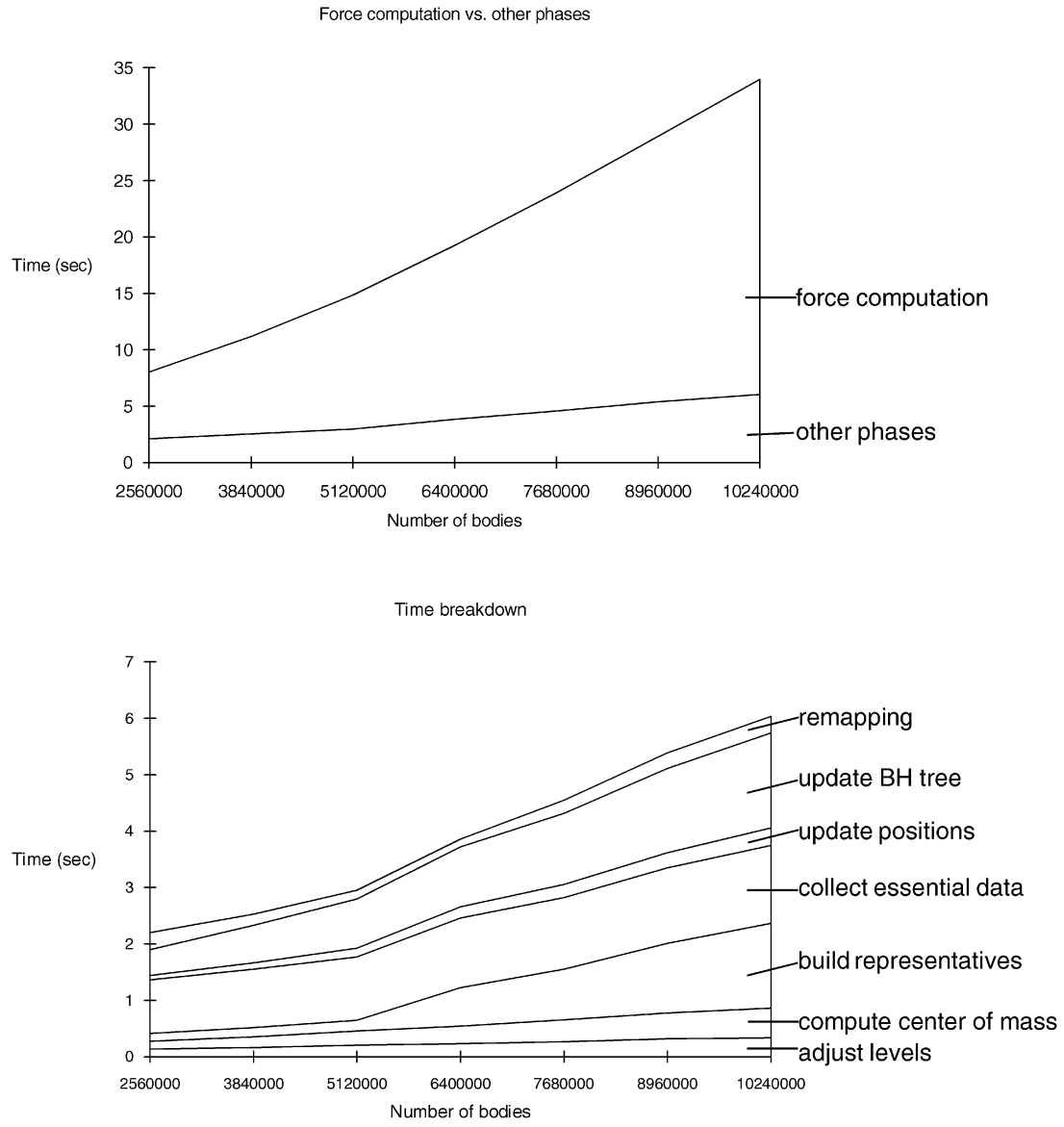

Fig. 11. Time breakdown for two Plummer spheres.

an interface between C and the DPEAC assembly language for vector units. The rest of the program is written in C.

Our experiments included three input distributions: uniform distribution, Plummer distributions [1] with mass M = 1 within a sphere, and two colliding Plummer spheres. The Plummer sphere has very large density in the center. All examples contained about 10 million particles. Fig. 9 shows the time spent per phase for the Plummer sphere. Fig. 10 and Fig. 11 show corresponding numbers for uniform distribution and two colliding Plummer spheres. The time can be classified into four categories. The first is the time to manage the distributed Barnes-Hut tree. This includes level adjustment, BH-tree update, and combining the local trees into the global representation. Less than 10 percent of the total time is spent for these activities for uniform and Plummer distribution. The corresponding figures for the two Plummer sphere distribution is 13 percent. The total execution times were 26 seconds, 42 seconds, and 33 seconds, respectively.

The second category is the time for constructing locally essential trees. The implementation packs information into long messages to improve communication throughput. This phase uses less than 5 percent of the total time.

The third category is time to compute accelerations. This category includes the time for vector units to compute

	WS92	WS93	Current
machine	512-Delta	512-Delta	256-CM-5E
# bodies($\times 10^6$)	8.8	8.8	10
distribution	uniform	uniform	uniform
time per step	77 sec.	114 sec.	26 sec.
force calc.	85%	47%	76%
other overhead	15%	53%	24%

Fig. 12. Comparisons with implementations of Warren and Salmon [34] [35]. The second last row is the percentage of time devoted exclusively to computing interactions (the entry for WS92 includes time for tree traversal).

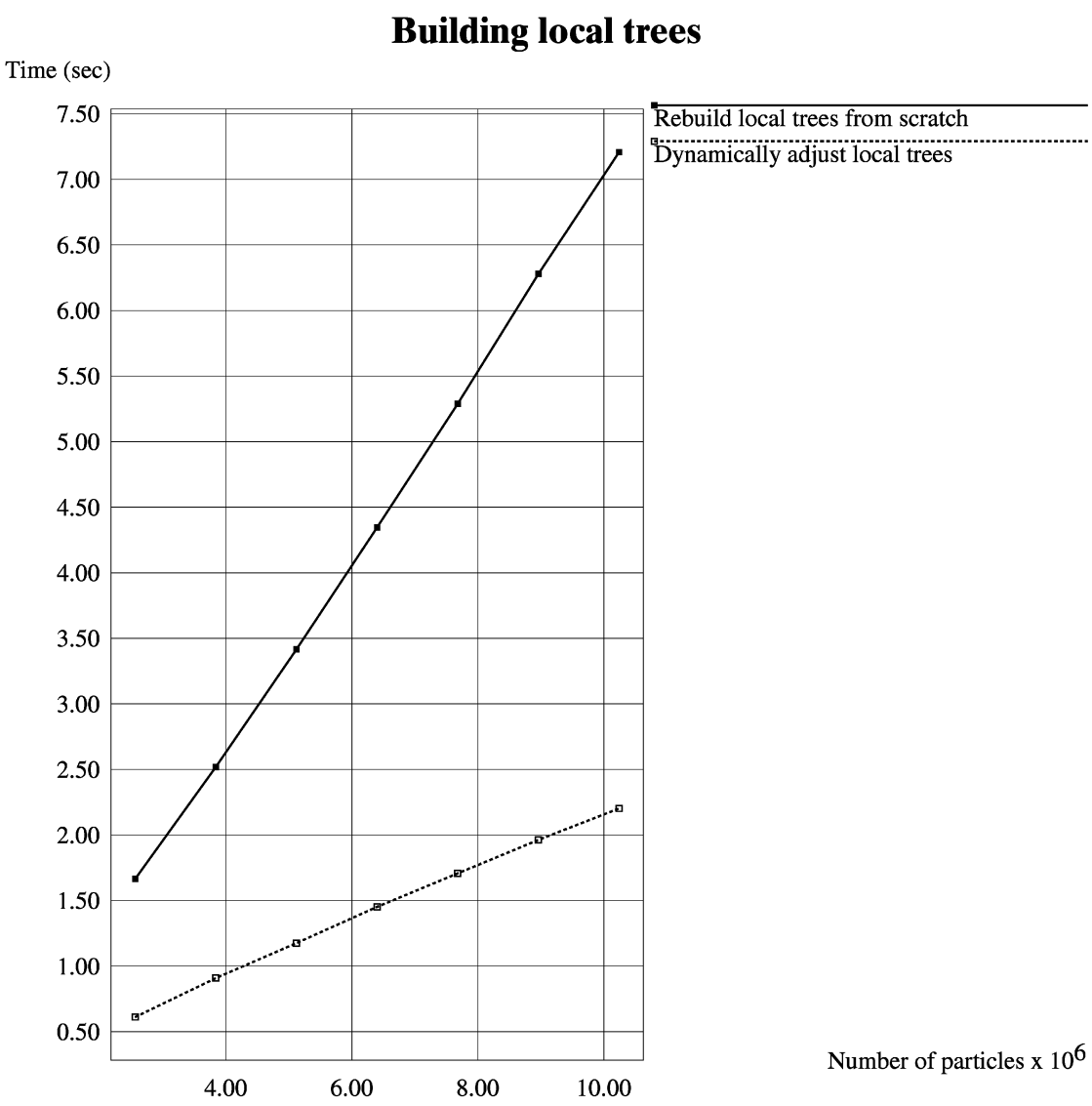

Fig. 13. Adjust vs. rebuild BH tree.

interactions among particles, and the time to modify the essential data cache. The vector units compute interactions at the rate of 96-113 Mflop/s, depending on the distributions. Even at this rate, the time spent by the vector units dominates; only 7-10 percent of the total time goes to cache modification.

The final category is the time for load balancing. Our implementation successfully balances the workload with negligible overhead. The simulation adjusts the workload distribution only when the imbalance exceeds 5 percent. As a result, the amortized cost for remapping is extremely small per simulation step.

The implementations sustains an overall rate of over 77 Mflop/s per processor, or 19.7Gflop/sec for the 256-node configuration. The hand-written CDPEAC assembly routine achieves 113 Mega flops in the interaction computation. The rest of the overhead is less than 24 percent. For the uniform distribution, the corresponding figure is less than 19 percent.

These figures compare favorably with those reported by Warren and Salmon [34], [35] (see Fig. 12). One important remark is in order: while our simulations were run over several minutes of wall-clock time, Warren and Salmon's figures are averages over almost 17 hours.

Our incremental tree structure is more efficient than the conceptually simpler method of [35]. The tree building phase in their implementation takes more than 12 percent of the total time. Singh et al. present a method similar to ours which takes about 5 percent to build the tree. If the final phase in both these approaches is speeded up by grouping bodies as we do, then the fraction of time in building the tree will be significantly higher. In contrast, our code spends less than 10 percent of the total time to update the tree for uniform distribution.

4.2 Discussion of Results

BH tree Adjustment. Fig. 13 compares the time to dynamically adjust the BH tree versus building it from scratch. The time for rebuilding the tree is taken from the first time local trees are built. The actual rebuilding time in later steps is larger because the number of bodies per processor can vary greatly after the first time step.

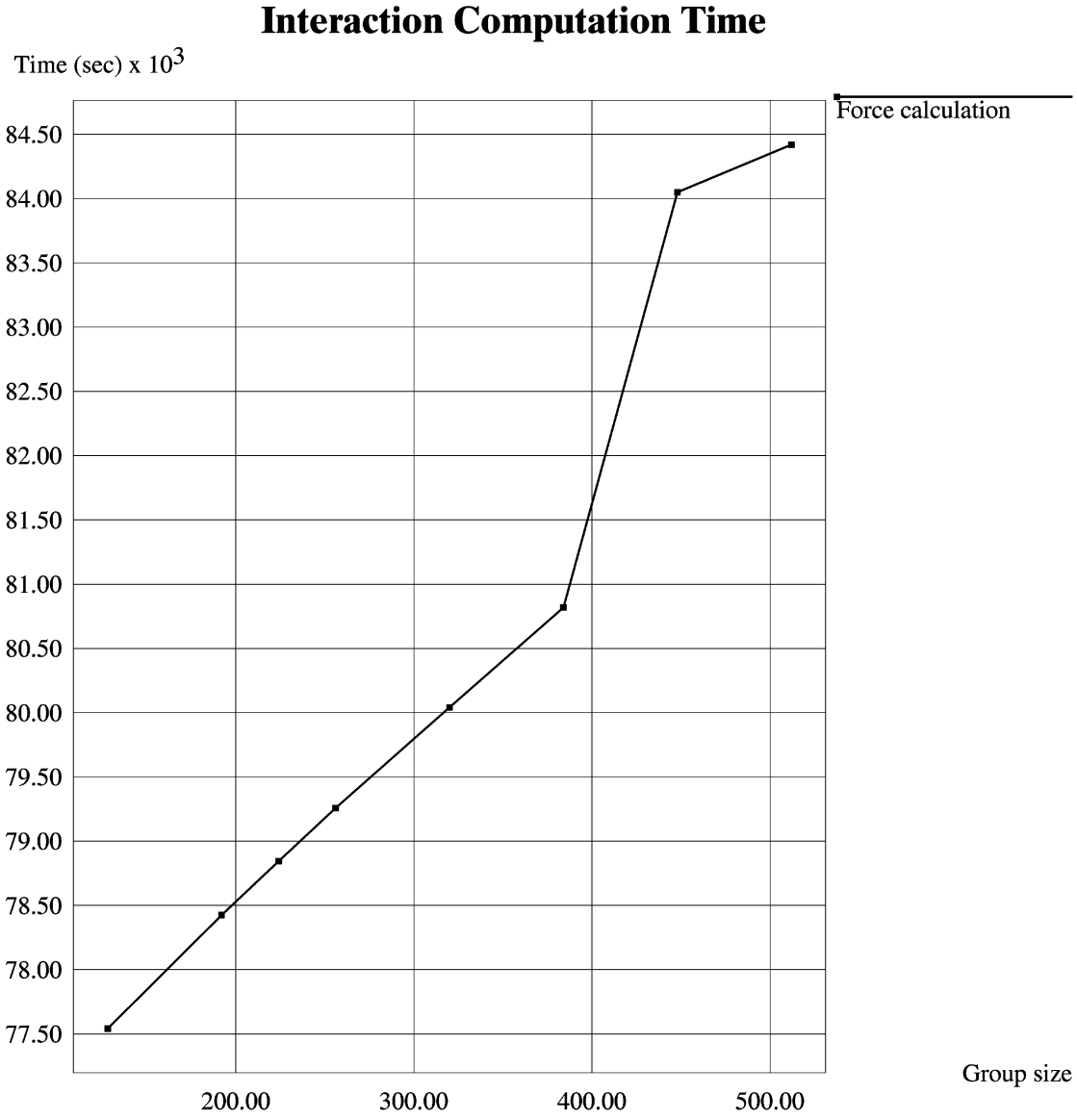

Fig. 14. Time to compute interactions for different group sizes for a Plummer sphere containing 10 million bodies.

The memory allocation routine is the major overhead in tree building process. Whenever a new BH node is inserted into the tree, the implementation must allocate memory for it. The memory management routines (malloc) provided by UNIX operating systems has extra overhead and contributes to the slow tree building process. In the implementation, we use our customized memory allocation routine to acquire memory for BH tree. Although the customized routine reduces the overhead in memory management considerable, the rebuilding is still more expensive than adjustment because of the extra overhead in releasing and allocating all the BH nodes.

In [28], Singh suggests that shared memory architecture has substantial advantages in programming complexity over an explicit message-passing programming paradigm, and the extra programming complexity translates into significant runtime overheads in message-passing implementation. However, in our implementation, we do not see this happen. Our implementation uses direct message-passing communication to manage the BH tree, but the overhead is very small with respect to the overall execution time.

Caching vs. Traversal. Fig. 14 shows the effect of different group size on the time for vector units to compute interactions. The computation time increases when the maximum number of bodies in a group (denoted by G) increases. As we compute acceleration for larger groups, the bounding box for the group increases in size. As a result, the number of BH boxes opened unnecessarily also increases, as does the size of essential data cache. Therefore, the time for vector unit to process essential data increases.

The increase in computation time is not significant until G increases to around 400 for the following reason. Consider a uniform particle distribution. In order to double the size of the bounding box, the number of bodies must increase by a factor of eight in three dimensions. Therefore, the increase in G must be significant to increase the cache length. Second, only those BH boxes surrounding the bodies will be affected by the change in G. Finally, the vector units process essential data in blocks of sixteen, so a

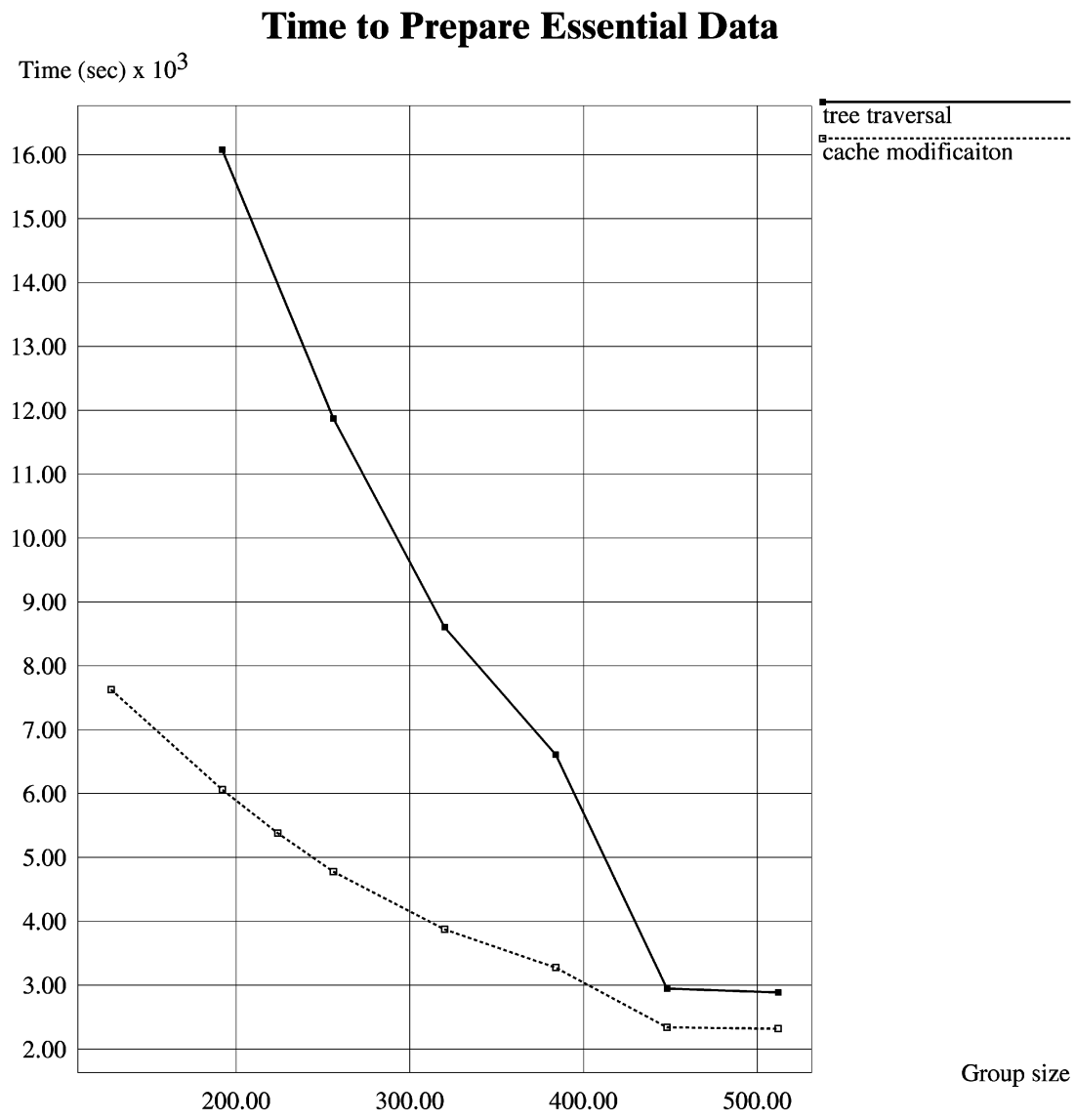

Fig. 15. Comparing caching vs. tree traversal to collect essential data.

small increase in G may not affect the total time for vector units to compute interactions.

Fig. 15 shows the effect of G on the time to prepare essential data for interaction computation. When G increases, the time to collect essential nodes decreases in both tree traversal and caching method. The effect on tree traversal strategy is easy to understand. The number of tree traversals is inversely proportional to G, so the tree traversal time decreases as G increases.

Increasing G has two different effects on the time to modify cache data. First, the number of cache modification decreases as more bodies are processed at a time, so the time should decrease as G increases. On the other hand, each cache modification will become more expensive when G increases. The increased size of the bounding box will decrease the cache hit rate because the distance from one group to the next increases. As a result, more expand/shrink operations become necessary and this increases the cost.

Fig. 16 shows the total time for force computation under different values of G. The combined effect of increasing vector unit times for computing interactions and decreasing time for preparing essential data gives minimum total time when G is about 320 for caching (450 for tree traversal). Although the advantage of caching gradually disappears when the group size increases to very large values, it outperforms tree traversal for all group size up to 512, and gives the overall minimum force computation time.

From the experiments, we can see that the effect of reduced number of cache modification is more significant than the increased cost per cache modification. As a result, the time for cache modification decreases as G increases. The reducing rate is slower than the tree traversal method in which the cost per group does not change.

Fig. 17 also shows that the cache hit rate decreases as more bodies are processed in a group. The increased bounding box size increases the distance from one group of bodies to the next group.

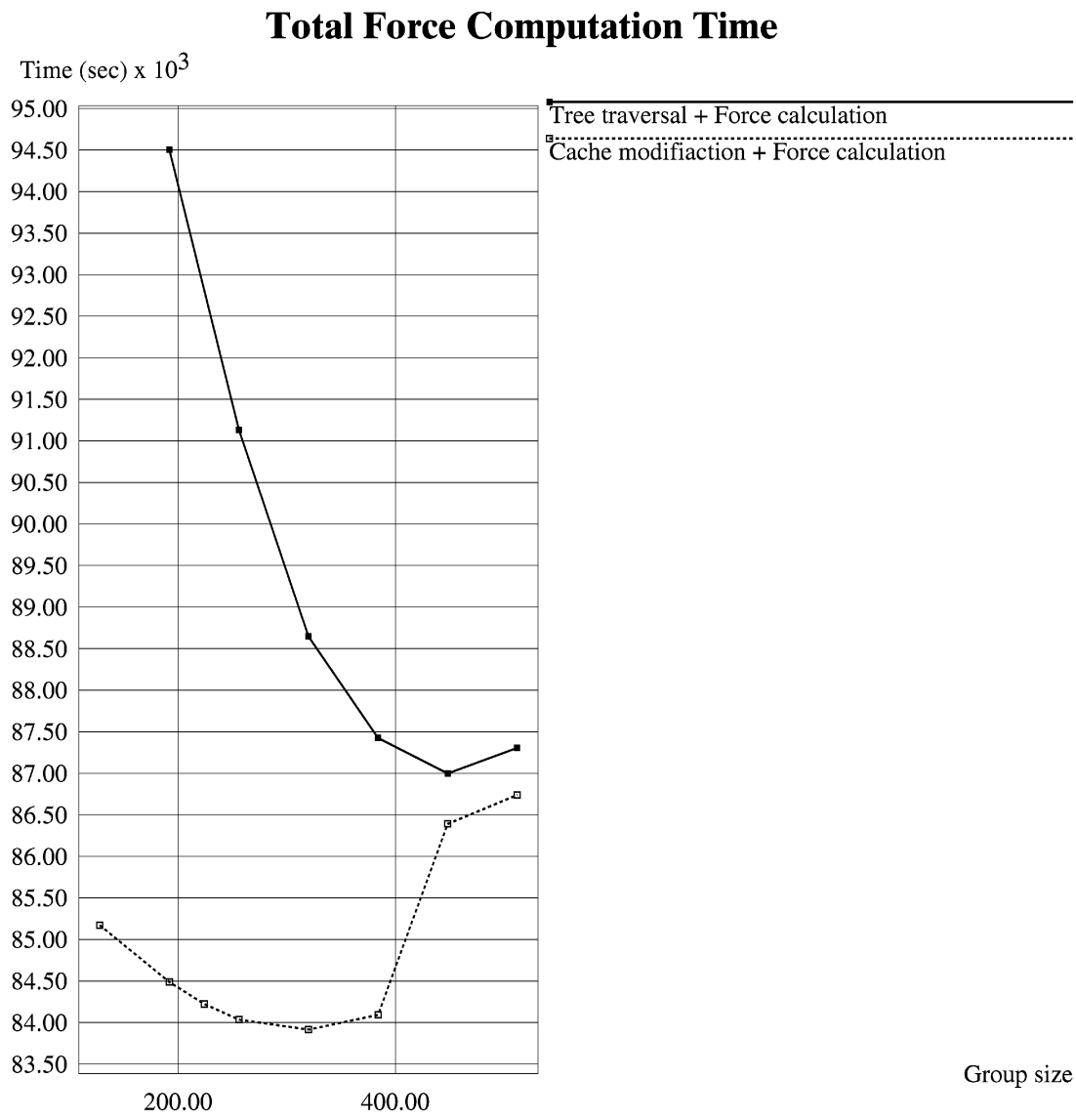

Fig. 16. Total time to collect essential nodes and to compute interactions.

4.3 Network of Workstations

Based on the CM-5 implementation, we also implemented a platform-independent parallel N-body framework [19], [20]. Within the framework, the users will be able to concentrate on the computation kernels that differentiate different tree-structured scientific simulation problems, and let the framework take care of the tedious and error-prone details that are common among these applications.

We demonstrate the flexibility of the parallel tree framework by implementing two applications—a gravitational force field computation and a multifilament fluid dynamic calculation. Both applications were developed within the tree library framework; therefore, all the tree structure details and communications were taken care of by predefined tree operations and the communicator classes.

The experiments were conducted on four UltraSPARC-II workstations located in the Institute of Information Science, Academia Sinica. The workstations are connected by a fast Ethernet network capable of 100 Mbps per node. Each workstation has 128 Mbytes of memory and runs on SUNOS 5.5.1. The communication library in the framework is implemented on top of MPI.

4.3.1 Gravitational Force Field Calculation

Table 1 summarizes the speedup factors of our parallel implementation on a cluster of four workstations. To get fair speedup numbers, we compare the parallel execution time with the timing from a highly optimized sequential C code, implementing the same algorithm, written by Barnes and Hut. The input configuration is a set of uniformly distributed particles in three dimension. The C code uses various techniques including in-memory caching of the vector $\vec{x_i} - \vec{x_j}$ between determining whether to open a cell (traverse down the tree for smaller subcluster) and the actual evaluation of the potential filed and acceleration. Nevertheless, our implementation gives competitive performance, even compared with this highly optimized C code.

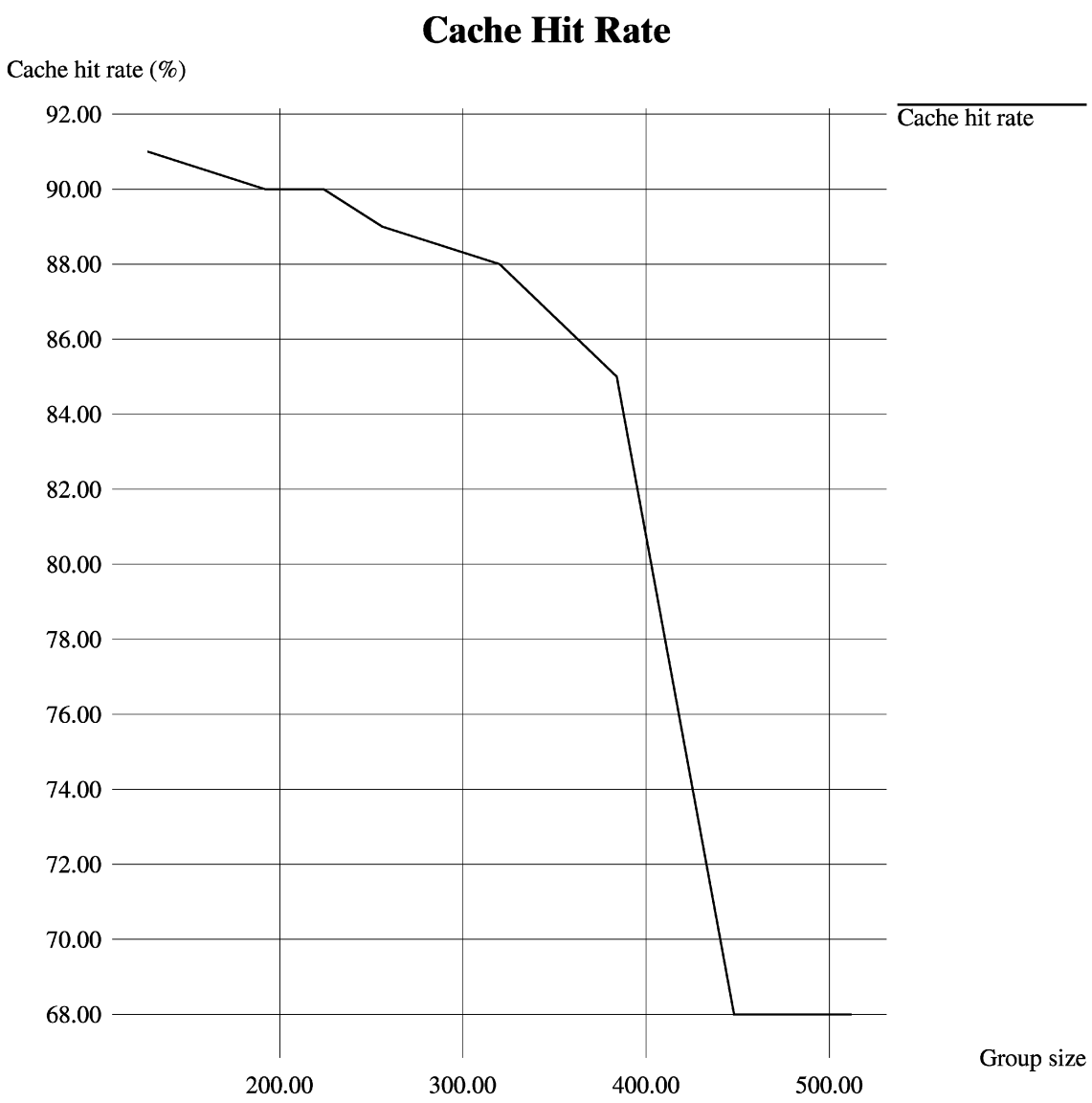

Fig. 17. Cache hit rate.

TABLE 1
Timing Comparison between the Parallel C++ Code Using the Framework and a Sequential C Implementation for Gravitation Field Computation

problem size	8k	16k	24k	32k	40k	48k	56k	64k	128k	256k
sequential time	14.39	19.43	30.47	41.73	54.60	65.62	81.23	93.59	186.12	413.75
parallel time	5.84	6.28	9.80	13.25	17.50	21.03	26.17	29.17	58.78	125.78
speedup	2.46	3.10	3.11	3.15	3.12	3.12	3.12	3.17	3.12	3.23

The time units are seconds.

TABLE 2
Timing Comparison for the Fluid Codes

problem size	8k	16k	24k	32k	40k	48k	56k	64k	128k	256k
sequential time	17.38	41.78	67.53	93.75	122.23	148.60	175.46	204.18	404.34	801.81
parallel time	5.51	12.81	19.77	27.84	34.96	43.00	51.37	59.05	117.20	231.07
speedup	3.15	3.26	3.42	3.38	3.46	3.42	3.42	3.46	3.45	3.47

The time units are seconds. The parallel code was written using the tree framework, and the sequential code was converted from Barnes and Hut's code.

4.3.2 Multiple-Filament Vortex Simulation

We implemented a multifilament vortex method using our framework. This line of work is based on a previous CM-5 implementation [15], [10]. We solve Biot-Savart's interaction between vortex elements using the algorithm by Knio and Ghoniem [17]. The multiple-filament vortex method computes the vorticity on each particle, and requires an extra phase in the tree construction to compute the vorticity. The vorticity of a particle is defined as the displacement of its two neighbors in the filament (see (1)). Once the vorticity on each particle is computed, we can compute the multipole moments on the local trees. Finally, each processor sends its contribution to a node to the owner of the node so that individual contributions are combined into globally correct information, as in the gravitational case.

$$\frac{d\vec{x}_i}{dt} = -\frac{1}{4\pi}\sum_j \frac{(\vec{x}_i - \vec{x}_j)\times\Delta\vec{x}_j}{|\vec{x}_i - \vec{x}_j|}\left(1 - e^{-\frac{|\vec{x}_i - \vec{x}_j|^3}{\delta_j^3}}\right) \qquad (1)$$

$$\Delta\vec{x}_j = \frac{1}{2}(\vec{x}_{j+1} - \vec{x}_{j-1}).$$

Table 2 summarizes the timing comparison between our parallel code and a sequential C code modified from the previously mentioned Barnes and Hut's implementation, which is highly optimized. The fluid dynamics code developed using the tree framework also delivered competitive performance. The speedup factors are higher than those of the gravitation code, because the fluid dynamics code performs more computation on each particle, which amortizes the overhead of parallelization and object orientation.

5 Conclusions

Our experiments demonstrate that adaptive and irregular tree structures for N-body simulations can be implemented efficiently in distributed memory using explicit message-passing communication. Maintaining incremental data structures substantially reduces the overheads due to parallel implementation. We also find that Barnes' technique of grouping bodies to compute accelerations reduces the time dramatically by allowing efficient utilization of the vector units.

Acknowledgments

The authors are grateful to Lennart Johnsson and Thinking Machines Corporation for their support and for access to their machines. The experiments also used the CM-5 at the Naval Research Laboratories and Minnesota Supercomputer Center. We thank John Salmon for early discussions, and Marina Chen, Young-il Choo, Cheng-Yee Lin, Satish Pai, Abhiram Ranade, and Fang Wang for many helpful discussions. Fang Wang was invaluable in programming the vector units. The authors would also like to thank Joshua Barnes and George Lake for encouragement.

This research was supported in part at Rutgers by ONR Grant N00014-93-1-0944. Partial support at Yale was provided by the US National Science Foundation/DARPA grant CCR-89-08285, DARPA contract DABT 63-91-C-0031 monitored by the US Army DOC, and Air Force grant AFOSR-89-0382.

References

[1] S. Aarseth, M. Henon, and R. Wielen, "Numerical Methods for the Study of Star Cluster Dynamics," *Astronomy and Astrophysics,* vol. 37, 1974.

[2] R. Anderson, "Tree Data Structures for *N*-Body Simulation," *Proc. 37th Ann. Symp. Foundations of Computer Science,* 1996.

[3] A. Appel, "An Efficient Program for Many-Body Simulation," *SIAM J. Scientific and Statistical Computing,* vol. 6, 1985.

[4] J. Barnes, "A Modified Tree Code: Don't Laugh; It Runs," *J. Computational Physics,* vol. 87, 1990.

[5] J. Barnes and P. Hut, "A Hierarchical $O(N \log N)$ Force-Calculation Algorithm," *Nature,* vol. 324, 1986.

[6] J. Bartholdi and L. Platzman, "Heuristics Based on Space-Filling Curves for Combinatorial Problems in Euclidean Space," *Management Science,* vol. 34, 1988.

[7] D. Bertsimas and M. Grigni, "Worst-Case Examples for the Space-Filling Curve Heuristic for the Euclidean Traveling Salesman Problem," *Operation Research Letter,* vol. 8, 1989.

[8] S. Bhatt, M. Chen, C. Lin, and P. Liu, "Abstractions for Parallel N-Body Simulation," *Proc. Scalable High Performance Computing Conf. (SHPCC '92),* 1992.

[9] S. Bhatt and P. Liu, "A Framework for Parallel N-Body Simulations," *Proc. Third Int'l Conf. Computational Physics,* 1995.

[10] S. Bhatt, P. Liu, V. Fernadez, and N. Zabusky, "Tree Codes for Vortex Dynamics," *Proc. Int'l Parallel Processing Symp.,* 1995.

[11] G. Blelloch and G. Narlikar, "A Practical Comparison of n-Body Algorithms," *Parallel Algorithms.* Am. Math. Soc., 1997.

[12] J. Board, Z. Hakura, W. Elliot, D. Gray, W. Blanke, and J.F. Leathrum, "Scalable Implementations of Multipole-Accelerated Algorithms for Molecular Dynamics," Technical Report 94-002, Duke Univ., 1994.

[13] J. Board, Z. Hakura, W.S. Elliot, and W. Rankin, "Scalable Variants of Multipole-Accelerated Algorithms for Molecular Dynamics," Technical Report 94-006, Duke Univ., 1994.

[14] P. Callahan and S. Kosaraju, "A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields," *J. ACM,* vol. 42, no. 1, pp. 67–90, Jan. 1995.

[15] V. Fernadez, N. Zabusky, S. Bhatt, P. Liu, and A. Gerasoulis, "Filament Surgery and Temporal Grid Adaptivity Extensions to a Parallel Tree Code for Simulation and Diagnostics in 3D Vortex Dynamics," *Proc. Second Int'l Workshop in Vortex Flow,* 1995.

[16] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *J. Computational Physics,* vol. 73, 1987.

[17] O.M. Knio and A.F. Ghoniem, "Numerical Study of a Three-Dimensional Vortex Method," *J. Computational Physics,* vol. 86, 1980.

[18] P. Liu, W. Aiello, and S. Bhatt, "An Atomic Model for Message Passing," *Fifth Ann. ACM Symp. Parallel Algorithms and Architecture,* 1993.

[19] P. Liu and J. Wu, "A Framework for Parallel Tree-Based Scientific Simulation," *Proc. 26th Int'l Conf. Parallel Processing,* 1997.

[20] P. Liu and J. Wu, "Supporting Efficient Tree Structures for Distributed Scientific Computing," *J. Information Science and Eng.: Special Issue on Compiler Technique for High-Performance Computing,* vol. 14, no. 1, 1998.

[21] P. Mills, L. Nyland, J. Prins, and J. Reif, "Prototyping N-Body Simulation in Proteus," *Proc. Sixth Int'l Parallel Processing Symp.,* 1992.

[22] L. Nyland, J. Prins, and J. Reif, "A Data-Parallel Implementation of the Adaptive Fast Multipole Algorithm," *DAGS/PC Symp.,* 1993.

[23] L.K. Platzman and J.J. Bartholdi, "Spacefilling Curves and the Planar Traveling Salesman Problem," *J. ACM,* 1989.

[24] G. Pringle, "Numerical Study of Three-Dimensional Flow Using Fast Parallel Particle Algorithms," PhD thesis, Napier Univ., 1996.

[25] J. Reif and S. Tate, "The Complexity of N-Body Simulation," *Proc. Int'l Colloquium on Automata Languages and Programming,* 1993.

[26] J. Salmon, "Parallel Hierarchical N-Body Methods," PhD thesis, Caltech, 1990.

[27] J. Salmon and M. Warren, "Skeletons from the Treecode Closet," *J. Computational Physics,* vol. 111, no. 1, pp. 136–155, 1994.

[28] J. Singh, "Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors," PhD thesis, Stanford Univ., 1993.
[29] J. Singh, J. Hennessy, and A. Gupta, "Implications of Hierarchical *N*-Body Methods for Multiprocessor Architectures," *ACM Trans. Computer Systems,* vol. 13, no. 2, pp. 141–202, May 1995.
[30] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load Balancing and Data Locality in Hierarchical N-Body Methods," Technical Report CSL-TR-92-505, Stanford Univ., 1992.
[31] S. Sundaram, "Fast Algorithms for N-Body Simulations," PhD thesis, Cornell Univ., 1993.
[32] Thinking Machine Corp., "CMMD Reference Manual," 1993.
[33] M. Warren, D. Becker, M. Goda, J. Salmon, and T. Stering, "Parallel Supercomputing with Commodity Components," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications,* 1997.
[34] M. Warren and J. Salmon, "Astrophysical N-Body Simulations Using Hierarchical Tree Data Structures," *Proc. Supercomputing,* 1992.
[35] M. Warren and J. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm," *Proc. Supercomputing,* 1993.
[36] M. Warren and J. Salmon, "A Portable Parallel Particle Program," *Computer Physics Comm.,* 1995.
[37] G. Xue, "An $O(n)$ Time Hierarchical Tree Algorithm for Computing Force Field in n-Body Simulations," *Theoretical Computer Science,* vol. 197, nos. 1–2, pp. 157–169, May 1998.
[38] F. Zhao and S. Johnsson, "The Parallel Multipole Method on the Connection Machine," *SIAM J. Scientific and Statistical Computing,* 1991.

Pangfeng Liu received the BS degree in computer science from National Taiwan University in 1985, and the MS and PhD degrees in computer science from Yale University in 1990 and 1994, respectively. He is now an associate professor in the Department of Computer Science and Information Engineering of the National Chung Cheng University, Taiwan. His research interests include parallel and distributed computing, randomized algorithms, and object-oriented methodology. He is a member of the ACM and the IEEE.

Sandeep N. Bhatt received the BS. MS, and PhD degrees from the Massachusetts Institute of Technology. Previously, Dr. Bhatt was on the research staff at Telcordia Technologies (formerly Bellcore), and on the faculty at Yale University. Currently, he is a principal research scientist at Akamai Technologies in Cambridge, Massachusetts. At Akamai, he leads the Systems Performance Group in the design and development of tools to monitor and analyze the end-to-end performance of the Akamai network. Prior to Akamai, his research interests include network performance and fault monitoring, the design of self-configuring systems for network security management, models and algorithms for distributed computing and communications, distributed simulations of network signaling and routing protocols, and *N*-body systems. His theoretical research focuses on the theory of graph embeddings, with applications to VLSI circuit layout and parallel computing.