Experiences with Parallel N-body Simulation

Pangfeng Liu
DIMACS
Rutgers University
Piscataway, NJ 08855.

Sandeep N. Bhatt
Bell Communications Research
Morristown, NJ 07960.

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 30% of the peak machine rate. Preliminary timings compare favorably with other approaches.

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 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 [4] and Barnes and Hut [6], 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 and provided that certain control parameters are appropriately chosen.

Greengard and Rokhlin [8] developed the fast multipole method with $O(N)$ arithmetic complexity which is accurate to any fixed precision. Sundaram [16] 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 3 dimensional multipole method, the algorithm of Barnes and Hut continues to enjoy application in astrophysical simulations.

Salmon [13] implemented the Barnes-Hut algorithm, with multipole approximations, on message passing architectures including the NCUBE and Intel iPSC. Warren and Salmon [17, 18] report impressive performance from extensive runs on the 512 node Intel Touchstone Delta. Singh etal. [14, 15] also implemented the Barnes-Hut algorithm for the experimental DASH prototype. This paper contrasts our approach and conclusions with both these efforts.

Parallel implementations of various versions of the fast multipole method are described in [9, 10, 12, 14, 19].

The remainder of this abstract 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 Figure 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. Figure 4 is an example of a recursive partition in two dimensions. Alternative tree decompositions have been suggested [3]; the Barnes-Hut algorithm applies to these as well.

```
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
```

Figure 1: The Barnes-Hut algorithm

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 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 [13] 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" [13]. 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 ten thousand particles, more than 90% of the time is devoted to arithmetic operations involved in computing accelerations. Less than 1% 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 of Salmon's thesis [13] which motivated us initially, as well as the more recent reports of Warren and Salmon [17, 18], and of Singh etal. [14, 15]. We also point out the differences of our techniques from these approaches.

Salmon [13] and Warren and Salmon [17] 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. Figure 5 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 non-trivial book-keeping and synchronization. The book-keeping 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.

More recently, Warren and Salmon [18] reported a modified algorithm which uses a different criterion for applying center-of-mass approximations. The new implementation 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. These advantages are balanced by other factors: (1) the computation 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 is complicated and less transparent, (2) the advantages of sender-directed communication are lost, and (3) the data structures are not maintained incrementally. Section 4 gives more details on timing results are comparisons.

The DASH shared-memory architecture group at Stanford [14, 15] has investigated the implications of shared-memory programming for the Barnes-Hut algorithm as well as the 2-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 [18].

The arguments in [14] about the advantages of shared-memory over message-passing implementations are based largely on comparisons to the initial implementations of Salmon [13] and Warren and Salmon [17]. Since our message-passing implementation is considerably simpler and more efficient, the import of the arguments of [15, 14] 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 because of parallelism minimal.

[Pangfeng: The same.]

Figure 2 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 1.2, 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.

[Pangfeng: The same.]

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 processors.

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. [Pangfeng:

```
0. build local BH trees
   for every time step do:
1.  construct the BH-tree representation
1.1  adjust node levels

[Pangfeng:  Need a new phase here.

The computation of the alpha values of particles require
ordering of particles in each filament.  Then we can com
partial moments and combine them together in the next ph

]

1.2  compute partial node values on local
     trees
1.3  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
   enddo
```

Figure 2: Outline of code structure

This is also important for computing alpha because we need to know where the neighbor of a particle is if it is not in the local 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 [14], 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. The ORB tree structure is statically partitioned among processors as follows: each leaf represents a processor and each internal node is stored at the same processor as its left child. 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. A node is overloaded if its weight exceeds the average weight of nodes at its level by a small, fixed percentage, say 5%. 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 non-overlapping regions can be balanced in parallel.

[Pangfeng: The same]

At each step of the load-balancing step it is necessary to move bodies from the overloaded child to the non-overloaded 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. Because of space limitations we do not describe the use of the BH-tree in load-balancing.

[Pangfeng: The same]

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 [17], 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 [7] for further details concerning a library of abstractions for N-body algorithms.

[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.

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 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.

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 internal nodes which are split by ORB lines. We omit the details of the level-adjustment procedure in this extended abstract. A similar process was developed independently in [14]; 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 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.

[Pangfeng: The following paragraph has to be replaced.]

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 useful method.

[Pangfeng: The description of how to compute alpha goes here.]

Give the definition of alpha function.

Emphasize that the computation requires communication because the two neighboring particles may not be in the same local tree.

Emphasize that we need a linear ordering of all local particles to 1. compute alpha function 2. find out which particles should be sent out.

Describe the splay search tree implementation. We use a search tree to store the indices of particles for one filament. It is easy to insert/delete indices into/from the tree, and to obtain a linear ordering of all indices. We use a splay tree implementation because it is self-adjusting.

Emphasize that we can use the same sender-orineted protocol for the communicaiton. The reason is that when we go through the local particles in the linear order, if either of a particle's neighbor is missing, this paticle will be sent out (the data dependency is symmetrical). Therefore it is the responsibility of the owner of a data to send its informaiton.

We also keep track of the maximum distance between two neighbors so that we can compute the area to send a particle out. (This is much easier under ORB).

[Pangfeng: We may need to descibe what we meant by multipole moments because we replace “mass” (a scalar) by the alpha (a vector)]

Once level-adjustment is and the computation of alpha complete, each processor computes the 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 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, has several drawbacks: (1) it involves two-way communication, (2) the messages are fine-grain so that either the communication overhead is prohibitive or the programming complexity goes up, and (3) processors can spend substantial time requesting data for BH-nodes that do not exist.

[Pangfeng: The same.]

It is significantly easier and faster to first construct the locally essential trees. 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 with the ORB-map. 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.

[Pangfeng: The same.]

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.

[Pangfeng: The same.]

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 earlier used by Barnes [5].

[Pangfeng: We may want to change the following description of caching. We do not use caching here because of memory overhead, especially when we are using small theta in this CFD code.]

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.

[Pangfeng: Do we want this theorem here?]

Suppose that N is the size of the BH-tree. The number of interactions computed by the Barnes-Hut algorithm is $\Omega(N \log N)$. We show that the number of cache modifications for the sequential code is significantly smaller when the groups are ordered according to a recursive tree traversal.

Theorem 1 *With recursive tree traversal, the number of cache modifications is $O(N)$, when either (a) $\theta = 1$, or (b) the bodies are uniformly distributed.*

The general case, when both the distribution and θ are arbitrary is open.

3.4 Reducing communication times

[Pangfeng: We can also put the computaiton of alpha as an example of all-to-some.]

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 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.

[Pangfeng: We may want to emphasize that the control network of CM-5 makes this approach very efficient.]

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 [1] 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 [11] 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

[The same.]

Our platform is the Connection Machine CM-5 with SPARC vector units. All experiments reported here were run on a 256-node CM-5. Each processing node has 32M bytes of memory and can perform floating point operations at peak rate of 128 Mflop/s. We use the Connection Machine CMMD library (version 3.0). The vector units are programmed in CDPEAC which provides an interface between C and the DPEAC assembly language for vector units. The rest of the program is written in C.

[We need Victor to shred some lights on the inputs.]

Our experiments included three input distributions: uniform and Plummer distributions [2] 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. Figure 6 shows the time spent per phase for the Plummer sphere. 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 5 percent of the total time is spent for these activities. The corresponding figures for the uniform distribution and colliding Plummer spheres are similar, the main difference being that the total execution times were, respectively, 59 seconds and 73 seconds, rather than 88 seconds for the single sphere.

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 4 percent of the total time.

The third category is time to compute accelerations. This category includes the time for vector units to compute interactions among particles, and the time to modify the essential data cache. The vector units compute interactions at the rate of 44 Mflop/s. Even at this rate the time spent by the vector units dominates; only 4 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 38 Mflop/s per processor, or 9.8Gflop/sec for the 256-node configuration. The hand-written CDPEAC assembly routine achieves 44 Mega flops in the interaction computation. The rest of the overhead is less than 13%. For the uniform distribution the corresponding figure is less than 9%.

These figures compare favorably with those reported by Warren and Salmon [17, 18] (see Figure 3). 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.

	WS92	WS93	Current
machine	512-Delta	512-Delta	256-CM-5
# bodies($\times 10^6$)	8.8	8.8	10
distribution	uniform (?)	uniform	uniform
time per step	77 sec.	114 sec.	59 sec.
force calc.	85%	47%	91%
other overhead	15%	53%	9%

Figure 3: Comparisons with implementations of Warren and Salmon [17,18]. The second last row is the percentage of time devoted exclusively to computing interactions (The entry for WS92 includes time for tree traversal).

Our incremental tree structure is more efficient than the conceptually simpler method of [18]. The tree building phase in their implementation takes more than 12% of the total time. Singh etal. present a method similar to ours which takes about 5% 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 5% of the total time to update the tree.

4.1 Discussion of results

BH tree Adjustment Figure 8 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.

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 the extra overhead in releasing and allocating all the BH nodes.

In [14] 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 Figure 9 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. Secondly, 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 small increase in G may not affect the total time for vector units to compute interactions.

Figure 10 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 bounding box will decrease cache hit rate because the distance from one group to the next increases. As a result more expand/shrink operations become necessary and increases the cost.

Figure 11 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.

Figure 7 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 groups of bodies to the next group.

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. The results of further improvements which could not be included in this version will be reported at the conference.

Acknowledgment

We 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. Thanks to Joshua Barnes and George Lake for encouragement.

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

References

[1] Cmmd reference manual. Technical report, Thinking Machine Corporation, 1993.

[2] S.J. Aarseth, M. Henon, and R. Wielen. *Astronomy and Astrophysics*, 37, 1974.

[3] R. Anderson. personal communication. 1993.

[4] A.W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6, 1985.

[5] J. Barnes. A modified tree code: Don't laugh; it runs. *Journal of Computational Physics*, 87:161–170, 1990.

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

[7] S. Bhatt, M. Chen, C-Y Lin, and P. Liu. Abstractions for parallel N-body simulations. In *Proceedings of SHPCC 92*.

[8] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Physics*, 73:325, 1987.

[9] L. Johnsson and Y. Hu. personal communication. 1993.

[10] J. F. Leathrum Jr. and J. Board Jr. The parallel fast multipole algorithm in three dimensions.

[11] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message passing. In *5th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1993.

[12] L. Nyland, J. Prins, and J. Reif. A data-parallel implementation of the adaptive fast multipole algorithm. In *DAGS/PC Symposium*, 1993.

[13] J. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, Caltech, 1990.

[14] J. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors.* PhD thesis, Stanford University, 1993.

[15] 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 University, 1992.

[16] S. Sundaram. *Fast Algorithms for N-body Simulations.* PhD thesis, Cornell University, 1993.

[17] M. Warren and J. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing*, 1992.

[18] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm, 1993.

[19] F. Zhao and S.L. Johnsson. The parallel multipole method on the connection machine. Technical Report DCS/TR-749, Yale University, 1989.

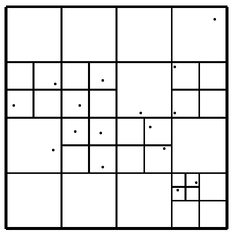

Figure 4: BH tree decomposition

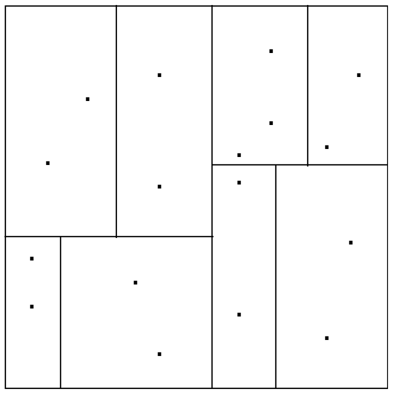

Figure 5: A two dimensional orthogonal recursive bisection

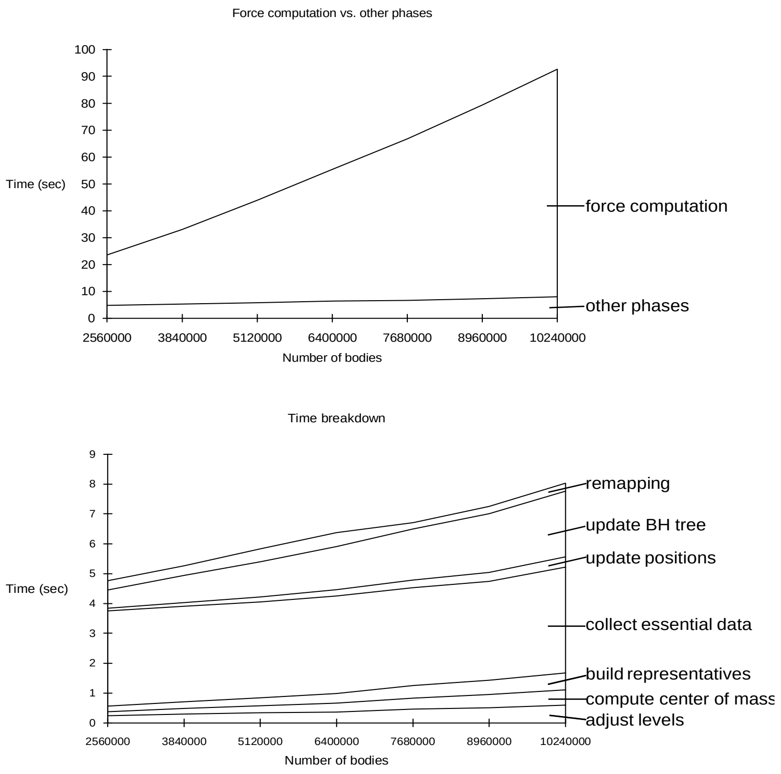

Figure 6: Time breakdown for Plummer sphere

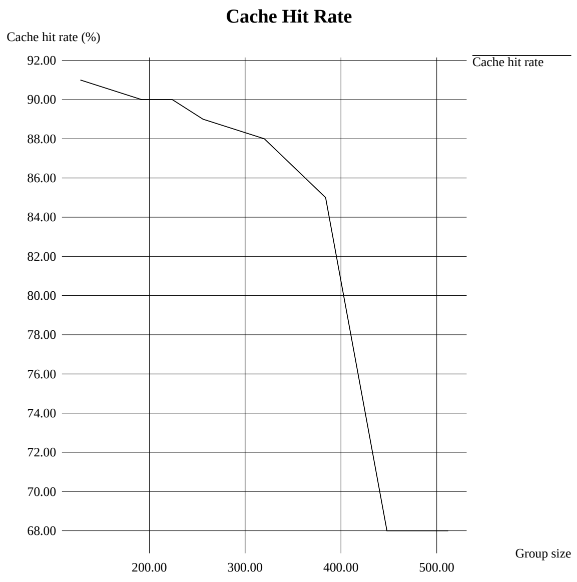

Figure 7: Essential data cache hit rate

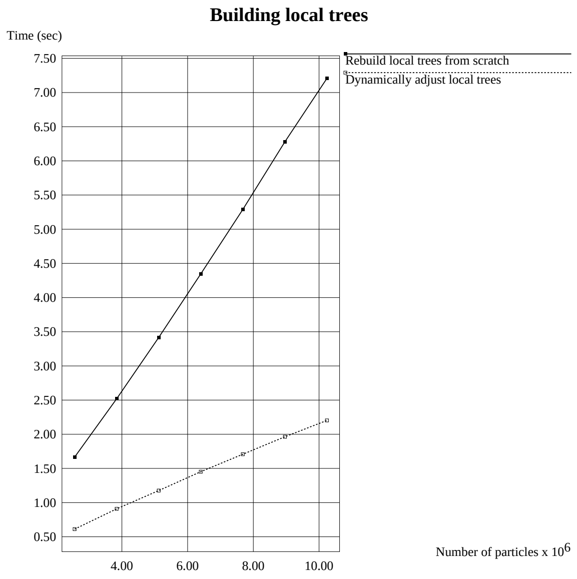

Figure 8: Adjusting vs rebuilding BH trees

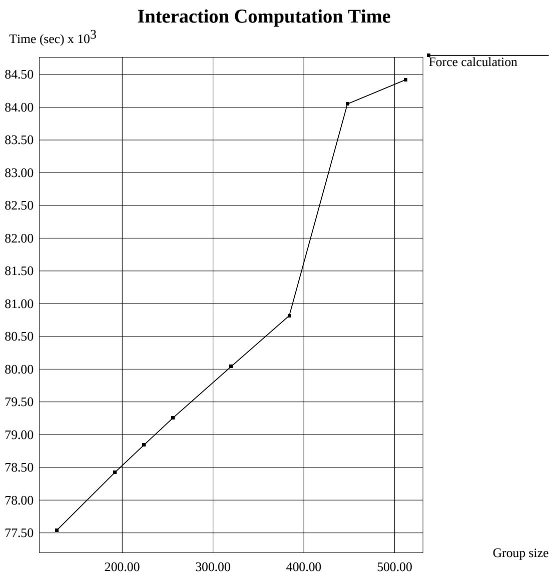

Figure 9: Time to compute interactions for different group sizes for a Plummer sphere containing 10 million bodies.

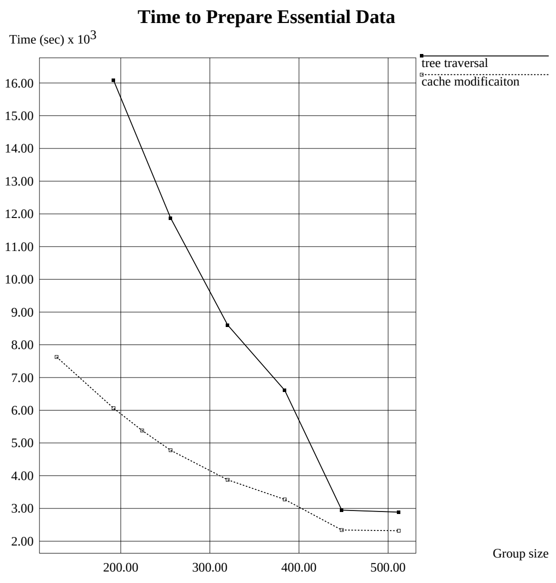

Figure 10: Comparing caching versus tree traversal to collect essential data.

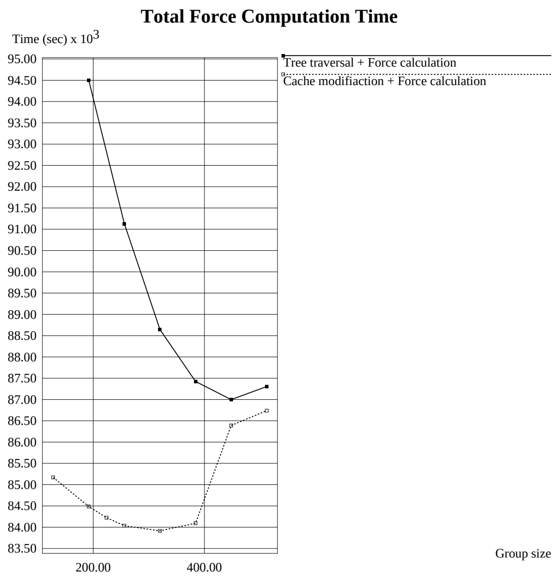

Figure 11: Total time to collect essential nodes and to compute interactions.