Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods

David Blackston [1] *Torsten Suel* [2]

Abstract

We describe the design of several portable and efficient parallel implementations of adaptive N-body methods, including the adaptive Fast Multipole Method, the adaptive version of Anderson's Method, and the Barnes-Hut algorithm. Our codes are based on a communication and work partitioning scheme that allows an efficient implementation of adaptive multipole methods even on high-latency systems. Our test runs demonstrate high performance and speedup on several parallel architectures, including traditional MPPs, shared-memory machines, and networks of workstations connected by Ethernet.

1 Introduction

The N-body problem is the problem of simulating the movement of a set of bodies (or particles) under the influence of gravitational, electrostatic, or other type of force. Algorithms for N-body simulations have a number of important applications in fields such as astrophysics, molecular dynamics, fluid dynamics, and even computer graphics [12]. A large number of algorithms for N-body simulations have been proposed; a basic approach taken by most of these algorithms is to simulate the system by advancing the bodies in discrete time steps. In each time step, the algorithm computes (or approximates) the force exerted on each body due to all other bodies; this determines the acceleration and speed of that body during the next time step. Computing the forces among a set of N bodies can be done in a straightforward way by computing all N^2 pairwise interactions. However, a number of more efficient algorithms have been proposed that can approximate the forces among N bodies in close to linear time.

An important class of such algorithms are the tree-based methods, which use a tree data structure to hierarchically group the bodies into clusters, that is, groups of bodies that are fairly close to each other (see, e.g., [1, 2, 3, 7, 10, 14, 30]). These methods can calculate the forces between N bodies in time proportional to $O(N \log N)$ or even $O(N)$, under certain assumptions about the input distributions.

Most of the $O(N)$ methods [1, 7, 10, 14, 30] have non-adaptive versions, which partition the space into boxes of a fixed size, and adaptive versions, which partition the space according to the particular input distribution, and which are more efficient for nonuniform distributions. On the other hand, the non-adaptive methods have a much simpler structure and can be more easily

[1]Computer Science Division, University of California, Berkeley. Email: `davidb@cs.berkeley.edu`.
[2]Information Sciences Research Center, Bell Laboratories. Email: `suel@research.bell-labs.com`.

tuned for high performance. As a result, while there have been many parallel implementations of non-adaptive $O(N)$ methods (e.g., see [4, 6, 17, 22, 27, 29]) and of the $O(N \log N)$ adaptive Barnes-Hut algorithm (e.g., see [20, 4, 24, 28, 26]), there are only very few parallel implementations of the $O(N)$ adaptive methods.

In this paper, we focus on the efficient parallel implementation of $O(N)$ adaptive tree codes, and in particular on the adaptive Fast Multipole Method [7] and the closely related adaptive version of Anderson's Method [1]. In the following, we refer to these algorithms, plus the also closely related method of Hrycak and Rokhlin [14], simply as *adaptive multipole methods*. Our goal is to design techniques that allow highly efficient implementations that are portable over a wide range of parallel architectures. We believe that highly optimized parallel implementations of adaptive multipole methods will eventually outperform the more commonly used methods as computation power and problem sizes continue to increase.

We have completed implementations of the adaptive versions of FMM and Anderson's Method. Some of the techniques that we derived are generalizations of ideas used in a highly portable implementation of the Barnes-Hut algorithm, which we include in our presentation for comparison, and for a simple illustration of some of the ideas. We believe that the main contributions of our work are as follows:

(1) We describe techniques that allow an efficient implementation of adaptive multipole methods on machines with very high latency and fairly low bandwidth.

(2) We provide portable and highly efficient parallel implementations of these methods.

(3) We compare the performance of several adaptive N-body methods on several different platforms.

Before continuing, let us briefly describe the approach taken in our work, and its goals and limitation. As the reader will notice, we focus on the basic algorithmic aspects of the N-body problem, a perspective taken by much of the N-body work from the computer science community. Our goal is to show how the basic structures of adaptive multipole methods can be efficiently implemented on parallel machines. We have tested our codes on several types of input distributions commonly used in the literature, but we have not (yet) applied our codes towards real problems arising, say, in astrophysics or molecular dynamics. Such applications will place additional application-dependent demands and constraints on the basic N-body algorithms, which need to be resolved before making any stronger claims of practicality. We hope to test our codes on a real application in a later phase of this project.

The remainder of this paper is organized as follows. The next section gives a brief description of the N-body algorithms used in our implementations. Section 3 discusses some related work. Section 4 contains a description of our implementations. Section 5 describes our scheme for data replication and work partitioning, and Section 6 contains some experimental results. Finally, Section 7 offers some concluding remarks.

2 Basic N-Body Algorithms

In this section we describe the basic structure of the N-body algorithms used in our implementations. We first explain the Barnes-Hut algorithm, which has a fairly simple structure, and then outline the more complicated structure of the adaptive FMM and Anderson's Method.

All algorithms considered in this paper use an oct-tree data structure to partition three-dimensional space and to group bodies that are close to each other. The tree structure for a given input distribution is obtained by recursively partitioning a box into eight smaller boxes of equal size until each box contains at most d particles (where d is experimentally determined for maximum performance).

2.1 Barnes-Hut Algorithm

The basic idea behind the Barnes-Hut [3] algorithm (and many other methods) is to approximate the force exerted on a body by a sufficiently far away cluster of bodies by computing an interaction between the body and the center of mass (or some other approximation) of the cluster. Bodies are grouped into clusters by the tree data structure, and a separation condition (which typically depends on the distance between the body and the cluster and on the size of the cluster) determines whether a cluster is sufficiently far away. The basic steps of one iteration of the (sequential) Barnes-Hut algorithm are as follows:

(1) Build the oct-tree data structure. This can be done by a simple tree insertion algorithm.

(2) Compute the center of mass (or some higher-order approximation) of the bodies in each box of the oct-tree. This is done in a upward traversal of the tree, starting with the centers of mass of the leaf boxes.

(3) For each body, perform a depth-first traversal of the oct-tree in order to compute the force exerted on the body. The traversal starts at the root.

   (3a) At each visited node, if the cluster defined by all bodies inside this node is sufficiently far away, compute an interaction with its center of mass (or some higher-order approximation of the cluster). Otherwise visit all its child nodes.

(4) Compute the acceleration and speed of each body in the current time step, and advance it accordingly.

A common separation condition is that a cluster is sufficiently far away from a body if the ratio of the side length of the box containing the cluster to the distance between body and cluster is smaller than some δ. The precision of the force calculation can be improved by choosing a small δ or, usually more efficiently, by using a higher order approximation of the cluster such as quadpoles [13] or hexadecapoles [19] instead of the center of mass monopole.

2.2 Adaptive FMM and Anderson

In the Barnes-Hut algorithm, interactions between bodies and sufficiently far away clusters are used to reduce the number of interactions from $O(N^2)$ to $O(N \log N)$ in the uniform case. The FMM and Anderson's Method go one step further, by also allowing interactions between two clusters. Thus, the force exerted on a body by a distant cluster is usually determined not by a direct interaction between body and cluster, but by an interaction between the distant cluster and a cluster containing the body. The force exerted on the cluster is then propagated to all the bodies contained in it; this is done in an additional downward pass through the oct-tree after the interaction phase in Step (3) of the algorithm. Apart from this extra step, the high-level structure of the algorithms is the same as in the case of Barnes-Hut, while the total number of interactions is reduced to $O(N)$, under some assumptions about the distribution of the bodies.

However, as a consequence, the internal structure of the interaction phase becomes somewhat more complicated. The adaptive FMM and Anderson's Method have four different types of interactions, which are executed depending on certain conditions about the relative size and location of the two interacting nodes in the tree. For example, in order for two clusters to interact (such an interaction is called a V interaction), they have to be of the same size and *well separated*, that is, have a sufficient distance from each other. Other rules govern the possible interactions between clusters and bodies (W and X interactions) and between bodies (U interactions). The accuracy of the method depends on the distance required for nodes to be well separated, but more importantly on the number of terms used in the approximation of the clusters.

The only difference between the FMM and Anderson's Method is in the way they approximate the force field of a cluster of bodies. While the FMM uses Taylor and Laurent expansions in 2D and expansions based on spherical harmonics in 3D, Anderson's Method is based on Poisson's formula. This makes Anderson's Method slightly easier to implement, while it appears to be still unclear which method gives the better accuracy/performance trade-off.

3 Other Parallel Implementations

In this section we describe some previous parallel implementations of tree-based N-body algorithms, and discuss their relation to our work. Due to the large amount of previous work, we can only discuss the most closely related work.

3.1 Barnes-Hut Algorithm

Over the last decade, there have been a large number of parallel implementations of the Barnes-Hut algorithm; a few examples are [20, 4, 24, 28, 26]. In [28], Warren and Salmon describe a fast message-passing implementation of the Barnes-Hut algorithm. They propose the use of locally essential trees to obtain a purely sender-driven protocol for replicating nodes that are accessed by several processors during the force computation phase. This approach was later refined by Liu and Bhatt [20] in their optimized Barnes-Hut implementation on the CM-5.

In a later implementation [26, 27], Warren and Salmon use a partitioning scheme based on space-filling curves and a distributed tree structure in order to achieve more flexibility in terms of application domains and separation criteria (MACs). A similar scheme based on space-filling curves is used in the shared-memory implementation by Singh [24], who also performs a comparative study of several partitioning and load-balancing schemes.

3.2 FMM and Anderson's Method

There have also been a number of parallel implementations of the non-adaptive versions of the FMM and Anderson's Method, some examples are [4, 6, 17, 22, 27, 29]. Due to the regular structure of these methods, parallel implementations can usually employ fairly simple schemes for partitioning, load-balancing, and replication. In contrast to the non-adaptive case, there have been only very few parallel implementations of the adaptive versions of FMM and Anderson. We are aware of the implementations in [9, 16, 18, 21, 24].

The earliest implementation we know of is the two-dimensional FMM code of Greenbaum [9] for the shared-memory NYU Ultracomputer. Another two-dimensional implementation for shared memory was done by Singh [24], who performed a detailed performance and scalability study for shared-memory machines. However, both approaches do not seem to be appropriate for distributed memory.

Parallel implementations of the three-dimensional FMM have been designed by Nyland, Prins, and Reif [21] and Krishnan and Kale [18]. The code in [21] is written in the high-level Proteus programming language. However, no performance numbers are given, and in fact no compiler for PROTEUS appears to be available yet. The code in [18] is written in the CHARM++ object-oriented programming language, and implements a slightly simplified variant of the adaptive FMM where the higher levels of the tree structure coincide with the ORB partitioning. However, the running times reported in [18] appear to be by about one order of magnitude too high.

Very recently, and while our work was in progress, Hu, Johnsson, and Teng [15, 16] have reported an implementation of the three-dimensional adaptive version of Anderson's Method in High-Performance Fortran (HPF). Their implementation appears to be the only adaptive three-dimensional code that achieves good performance on distributed-memory machines. The work in [15, 16] is also important because it provides techniques for the efficient implementation of adaptive algorithms in data-parallel languages such as HPF, which were mainly designed with more regular applications in mind. (An alternative way to overcome these problems would be to allow nested data parallelism, as provided, e.g., by the NESL language [5].)

However, a data-parallel implementation does not appear suitable for low-cost parallel systems, such as Ethernet-connected PCs or workstations, which typically have much higher latency and much lower bandwidth than their high-cost MPP counterparts. Our goal is to achieve high efficiency even on such low-cost systems. At the same time, we want to achieve very high performance on more powerful MPP systems; an interesting question is to what degree our "raw C" implementation can improve on HPF with its associated overheads.

4 Description of Our Codes

In the following, we describe the structure and implementation of our parallel codes. The basic structure of the codes is similar to that of the sequential algorithms outlined in Section 2, Our implementations share a significant amount of code between them. Hence, unless explicitly stated otherwise, the following descriptions apply to all algorithms that we have implemented.

4.1 Programming Environment

The programs were written in C using an SPMD programming style. Communication is performed through a small library of message-passing functions, called *Green BSP* and described in [8], which implements a set of bulk-synchronous communication primitives as prescribed by Valiant's Bulk-Synchronous (BSP) model of parallel computation [25]. The library offers simple functions for sending, synchronization, and receiving that partition the program execution into a sequence of computation phases called *supersteps* such that messages sent in one phase are received by the receiver only at the beginning of the next phase. This allows for a very efficient implementation of very-coarse grained parallel programs that are especially suitable for high-latency parallel machines. Thus, the library provides a very natural match for the communication schemes arising in our implementations. Versions of the library have been implemented on a number of different platforms.

On the other hand, our implementation techniques are not dependent on a particular library, but can be used on any reasonable message-passing library (e.g., MPI [11]), assuming that appropriate buffering is used to avoid software overheads due to the large number of fairly small messages sent by the program. In fact, one of the versions of the library used in our experiments is basically a thin layer on top of MPI that enforces certain buffering policies.

One slight disadvantage of the very large supersteps in our algorithms is that they require significant amounts of buffer space at both sender and receiver. (This problem is due to the nature of buffered message-passing in BSP, and the fact that the library version we used allocates a fixed amount of buffer space at the beginning of the program run.) To avoid this problem, we decided to split the largest supersteps into several smaller supersteps, resulting in a slight increase in code complexity.

4.2 Tree Construction

One problem faced by any implementer of tree-based parallel N-body methods is how to represent and efficiently build the tree structure used for the force calculation. A convenient way of doing this on shared-memory machines is to have all processors jointly build the tree, and use locks for consistency. However, such a scheme would be inefficient on high-latency distributed-memory platforms, and hence we decided on a scheme in which no global tree structure is built, but each processors constructs a separate tree structure that contains all its particles, and that is "consistent" with the (nonexisting) global tree.

The construction of these trees is very simple. Each processor starts by building a tree on its own particles, starting out with the entire extension of space in the root node, but essentially pretending that there are no other particles apart from its own. This results in a tree structure that is not fully consistent with the global one at nodes that are not completely within the boundaries of the local partition. However, as it turns out, these inconsistencies can be easily removed during the transmission of the locally essential trees described further below.

Thus, to built the trees in our parallel code, we just use the same tree insertion routine as in the sequential case, perform an upwards pass to compute the expansions, and then wait for the locally essential trees scheme to correct any inconsistencies. (Note that some of the expansions need to be updated after the locally essential trees have been received and inserted into the local tree.) Trees are completely rebuilt in each iteration.

4.3 Data Partitioning and Load Balancing

We have implemented two different schemes for partitioning and load balancing. Following Liu and Bhatt [20], load balancing is only performed when the load imbalance exceeds a certain threshold (typically about 2%). The weight each body contributes to the load of its processor is determined by the amount of work "caused" by the body in the previous superstep. In the case of Barnes-Hut, this is just the number of interactions and quadpole evaluations, while for Anderson and FMM, all costs incurred by internal nodes are propagated to the bodies in the downward pass of the algorithm.

Our first implementation was based on the Orthogonal Recursive Bisection (ORB) scheme [28], where space is recursively partitioned along the longest dimension, such that each of the two resulting rectangular regions receives a number of bodies equivalent to half of the remaining work. We initially implemented a fully distributed version of ORB partitioning in which the space is recursively halved by a parallel median finding algorithm. To increase the simplicity and flexibility of our code, we later switched to a mainly sequential implementation, where each processor sends a random sample of its body positions and weights to a single processor. This processor then locally computes a partition based on the sample, and broadcasts the result. Afterwards, the quality of the load balancing is checked and, if necessary, the process is repeated. We found this implementation to be very efficient even for fairly small input sizes.

The ORB partitioning resulted in excellent speed-up for the Barnes-Hut algorithm. For the FMM and Anderson's Method, however, we experienced significant load imbalances that severely limited the parallel speed-up (to about 6 on 16 processors for $64K$ bodies). The reason is that since work is really associated with nodes rather then bodies, any repartitioning of the load only has an effect if it changes the partitioning of the nodes. For example, in a simulation with $64K$ bodies and up to 100 bodies per leaf, we end up with about 2000 leaf nodes, or about 13 leaves along each of the three dimensions. This small set of possible partitionings along a single dimension results in a load imbalance that is compounded over the number of processors and dimensions.

This effect was already described by Singh [24], who observed only a fairly moderate degree of load imbalance in his two-dimensional runs. Our observation is that this problem is significantly worse in three dimensions. While the problem (very) slowly decreases with larger input sizes, and additional improvements might be possible by modifying the ORB scheme, we were not able to come up with a simple and convincing scheme for three dimensions. Hence, we conclude that ORB partitioning is probably not a good choice for adaptive multipole methods.

To solve these problems, we implemented the Cost Zones partitioning scheme described in [24]. Our implementation is again based on sending a random sample to a single processor, who then builds an oct-tree based on the bodies in the sample. Informally speaking, the overall shape of the resulting "sample tree" is very similar to that of the global oct-tree on all bodies. The processor then applies Cost Zones partitioning to this tree. By choosing the maximum number of bodies per leaf somewhat smaller than in the global tree, we can reduce the required sample size. The implementation runs very efficiently, and obtains good load balance. The size of the locally essential trees is only slightly larger than in the case of ORB, and could be somewhat reduced by using a continuous space-filling curve instead of the simpler non-continuous curve we have implemented.

4.4 Communication and Force Computation

As mentioned before, we use an extension of the locally essential trees approach in order to replicate tree nodes that are needed by more than one processor. This scheme is described in detail in the next section. Thus, at the beginning of the force computation phase, every processor already contains all of the data that it has to access during this phase, and the computation is completely local and can be done by the same code as in the sequential case.

Our implementation of the interactions in the FMM is based on the slight reformulation of the original method of Greengard and Rokhlin provided by Singer [23]. For Anderson's Method we relied on the description in [1], with precomputed matrices used in the V interactions and the upward and downward passes. Finally, our implementation of quadpole moments in Barnes-Hut follows the description in [13].

5 Replication Scheme

In this section, we give a high-level and general description of the replication scheme that we use in our codes. In the following, we assume that the bodies have already been partitioned among the processors using ORB, Cost Zones, or some other data partitioning scheme.

In order to perform all the interactions, we require two additional ingredients. First, we need a work partitioning that assigns every interaction to one of the processors. Such a scheme will be partly induced by the data partitioning, but also needs to uniquely assign interactions that require data from two or more processors. Second, we have to design an efficient communication and replication scheme that ensures that every processor has all of the information necessary to perform its interactions. The simplicity and efficiency of such a scheme depends of course on the data and work partitioning scheme.

What we are describing in the following is basically a generalized version of the "locally essential trees" scheme of Warren and Salmon, which was originally introduced for a Barnes-Hut algorithm with ORB partitioning, to adaptive multipole methods and other classes of data partitionings such as Cost Zones or the RRB scheme of Hu, Johnsson, and Teng [15].

5.1 Barnes-Hut and ORB

We first explain the scheme for the case of a Barnes-Hut implementation with ORB partitioning. We point out that this scheme is very similar to those in [28] and [20]. Note that in the Barnes-Hut algorithm, all interactions are between a particle and a node of the oct-tree. This makes the work partitioning very simple: a processor is responsible for all interactions involving particles located in its ORB region.

Thus, it remains to describe the replication scheme. During the Barnes-Hut algorithm, a particle will interact with a node if the particle is far enough away from the center of mass of the node, relative to the side length of the node.

Let P_1 and P_2 be two processors. Let P_1 have a locally constructed oct-tree T, and let P_2 have ORB region R. We can then perform a depth-first traversal of T in order to determine the locally essential tree with respect to P_2, that is, all the information in T that is needed by P_2 for its interactions.

More precisely, when we visit a node s in T we do the following. If s is a leaf node, we simply send all of the bodies within s to P_2. Otherwise, if s is contained entirely within P_1's ORB region, then we can calculate how far the center of mass is from the boundary of R. If this distance is large enough relative to the side length of s, then we only need to send the center of mass (or higher order approximation) of s to P_2, and return to the next higher level. If the center of mass is too close, then we recursively visit the children of s. If s is not contained entirely within P_1's ORB region, then P_1 does not know the true location of the center of mass of s in the global tree. We therefore assume the worst case, that the center of mass is as close to R as possible.

It can be shown that this procedure will transmit all information that is needed by P_2 to perform its interactions. Moreover, after the data has been appropriately inserted into the oct-tree at the receiving end, and after the centers of mass have been updated accordingly, each locally held tree will be consistent with the corresponding region of the (nonexistent) global tree.

5.2 Problems in the Generalization

Before explaining the generalization of the replication scheme to FMM and Anderson's Method, we describe some of the new problems that arise, and that make it non-trivial to extend and implement the replication scheme. The main challenges are as follows:

(1) In the Barnes-Hut method, the well-separatedness criterium depends only on the size of the source box, and its distance from the particle. In FMM and Anderson's Method, we have cluster-cluster interactions whose criterium can depend on the size of either the source or the object box (e.g., X and W interactions). This makes it more difficult to design a sender-driven protocol, since the sender does not know the structure of the local tree inside the receiver.

(2) FMM and Anderson's Method have four different types of interactions, each with its own set of conditions about when the interaction is applicable.

(3) In FMM and Anderson's Method, work is also associated with internal nodes, which can overlap the domains of several processors. Thus we need a work partitioning scheme that uniquely assigns these interactions to processors, and that is compatible with a simple sender-driven approach.

5.3 The General Scheme

In the general scheme, we assume that each processor has complete knowledge of the data partitioning. It is convenient to assume that the partitioning is represented in some simple geometric form, e.g., boxes in ORB, or unions of cubes (each corresponding to an oct-tree node) in Cost Zones.

First, we describe the method for assigning interactions to processors. As mentioned before, there are four types of interactions that can occur during the force calculation phase of the FMM algorithm. Each interaction involves a source node s and a target node t, and can be denoted by the pair (s, t). All of these interactions involve taking field information from s and converting it into field information for t.

For each node of an oct-tree, we define its location with respect to the local partitioning domain. A node is *inside* if it is completely inside the domain, *outside* if it is completely outside, and *shared* otherwise. Using this notation, we define the work partitioning by the following rules: (1) If the target node of an interaction is inside a partition, then the interaction is performed by the processor owning that partition. (2) If the target node of an interaction is shared, then the interaction is performed by the processor whose partition contains the unique point inside the target node that is closest to the geometric center of the source node. Note that this means that if a target node is shared, then the work associated with that node will usually be shared by several processors.

Considering the different types of interactions in FMM and Anderson's Method, we observe that in any interaction (s, t), the target node t must intersect a cube centered at the center of source node s and with side length proportional to the side length of s. The constant of proportionality depends on the separation parameter of the algorithm. Let us call this cube the *interaction cube*.

As before, to determine which parts of a local tree T in processor P_1 have to be sent to another processor P_2, we perform a depth-first traversal of T. When we encounter a node s, we do the following. If s is a leaf, then we simply send all of the bodies within s to P_2. If s is not a leaf, then we look at all of the interaction cubes for children of s. If any of these cubes intersect R, then we recursively continue the depth-first search. If none of the cubes intersect the region R, then we know that none of the descendents of s are going to be the source in an interaction that is assigned to P_2, and we simply send the expansion information for s to P_2 and return to the next higher level.

It can again be shown that this procedure will transmit all information that is needed by P_2 to perform its interactions. Moreover, after the data has been appropriately inserted into the oct-tree at the receiving end, and after the expansions have been updated accordingly, each locally held tree will be consistent with the corresponding region of the (nonexistent) global tree.

Note that the above description refers to the data partition only indirectly via geometric terms such as "inside a partition", "outside a partition", and "distance from a partition", which can be implemented based on the geometric representation of the particular partitioning scheme. This allows a nice separation between the data partitioning and the replication scheme in our code.

The actual scheme that we use in our codes is slightly more complicated than the description we have given here. In particular, we avoid the additional upward pass for updating the expansions at the receiver side by transmitting the expansion of a node even if we also transmit the information associated with all its children (or all its bodies if it is a leaf), thus trading off bandwidth versus computation. We also use some other conditions for pruning the size of the locally essential trees. These optimizations are motivated primarily not by bandwidth concerns, but by computation costs and space concerns at the receiving side.

6 Experimental Results

In this section, we present and discuss the results of an extensive series of test runs of our codes. We consider total work, parallel speed-up and scale-up, precision, and memory and communication requirements. Given the vast space of parameter settings, we have to restrict ourselves to a few interesting cases.

Our set of experiments is structured as follows. In the first subsection, we explore the basic characteristics of FMM and Anderson's Method through a sequence of sequential runs on a single node of the Cray T3E. In Subsection 6.2, we present parallel speed-up and scale-up results on three machines representing different classes of parallel architectures, for a small set of problems sizes and fixed precision settings. Finally, in Subsection 6.3 we discuss the memory and communication requirements of the parallel codes.

Note that all running times in this section are for a single iteration of the algorithm. We do not include the times spent on advancing the bodies (which is less than 1% of the total time), and the time spent on load balancing. It is somewhat difficult to estimate the cost of load balancing as it is only periodically performed. Both the frequency and the cost of the load balancing step depend heavily on the particular input distribution and the chosen time step. However, even in the worst case, (which is given by the initial partitioning step before the first iteration, and which is unlikely to ever occur thereafter), the time for a load balancing step is small compared to the time for the interactions.

Throughout this section, we choose the maximum number of bodies per leaf node as 100 for FMM and Anderson's Method, and 20 for Barnes-Hut. Note that the precise optimal value for this parameter depends in a non-trivial way on many factors such as input size and distribution. However, the values we have chosen come quite close to optimal in most cases we have encountered.

6.1 Sequential Performance Characteristics

We start out by presenting some basic sequential performance characteristics of the algorithms we have implemented. These results were obtained by running our codes on a single node of the Cray T3E. We emphasize that these sequential runs correspond to fairly generic sequential implementations of the algorithms, and that they do not incur any of the overheads associated with a parallel implementation.

One characteristic of the sequential algorithms that is not surprising is that the vast majority of the computation time is spent in the interaction computation phase of the algorithm. In all of the sample runs referred to in Tables 6.1 and 6.2, between 90 and 95 percent of the computation time was spent in this phase. The remaining 5 to 10 percent were primarily spent in the upward and downward passes, and only very little time was spent in building the tree.

Tables 6.1 and 6.2 show the percentages of time spent performing the various interactions during the interaction phase for uniform and Plummer distributions. Note that the percentages for the W and X interactions in the uniform case fluctuate in a complicated way with the average number of bodies per leaf node and with the structure of the tree.

In Table 6.3 we compare the running time of FMM and Anderson's Method for several accuracy settings. The input data is a uniformly distributed set of particles. The measured times are for the computation of the potentials, and do not include the time for the fields. The error (RMS) is calculated as follows. For any particle x_i, let $\Phi_c(x_i)$ be the potential at x_i calculated by the algorithm, and let $\Phi_r(x_i)$ be the potential computed by the direct method. Then the RMS error for the calculation is defined as

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\frac{(\Phi_c(x_i)-\Phi_r(x_i))}{\Phi_r(x_i)})^2}.$$

According to Table 6.3, Anderson's Method outperformed FMM for similar levels of accuracy. However, we point out that these results are still very preliminary, since our current version of the

Particles	U-Int	V-Int	W-Int	X-Int
1000	45.6	49.1	2.5	2.9
5000	21.3	45.8	11.8	19.0
10000	44.9	24.9	11.2	18.9
20000	77.1	17.3	2.3	3.3
50000	29.6	52.3	7.4	10.7

Table 6.1: Percentages of interaction time spent on the different interaction types (uniform distribution)

Particles	U-Int	V-Int	W-Int	X-Int
1000	35.5	9.4	20.1	34.9
5000	30.9	9.9	21.5	37.6
10000	27.8	15.7	20.7	35.8
20000	29.4	17.8	19.4	33.4
50000	28.1	21.9	18.6	31.4

Table 6.2: Percentages of interaction time spent on the different interaction types (Plummer model)

FMM code is less optimized than the Anderson code, and a fully optimized version might actually be slightly faster than Anderson's Method, We plan to perform a detailed comparison in the near future.

Finally, we repeated some of the sequential experiments on our other platforms. While the absolute times vary significantly, we found the relative behavior between different algorithms and settings to be similar on all platforms.

6.2 Parallel Performance Results

In this subsection, we present results on the parallel speed-up and scale-up achieved by our codes on three different machines, which cover several interesting classes of parallel architectures. The machines are:

(1) a Cray T3E, located at the National Energy Research Scientific Computing (NERSC) Center in Berkeley. The machine has 152 DEC Alpha processors with 256 MB of main memory each.

(2) an SGI Challenge shared-memory machine, located at the NEC Research Institute in Princeton. The machine has 16 MIPS R4400 processors and 1 GB of main memory.

(3) a network of PCs, also located at NEC. Each machine has two 200 Mhz Pentium Pro processors and 128 MB of main memory. The machines run the Linux OS and are connected

Accuracy	Algorithm	Particles	Time	RMS Error
Low	FMM (N=4)	1000	.318	2.25e-4
		5000	4.13	2.12e-4
		10000	7.97	2.01e-4
		20000	12.22	2.02e-4
		50000	40.85	.
	And (K=5)	1000	.171	5.15e-4
		5000	1.41	4.78e-4
		10000	3.62	4.71e-4
		20000	10.08	4.57e-4
		50000	17.13	.
Medium	FMM (N=6)	1000	.726	3.19e-5
		5000	9.90	2.44e-5
		10000	15.81	2.06e-5
		20000	19.33	1.90e-5
		50000	103.0	.
	And (K=9)	1000	.299	4.00e-5
		5000	3.08	3.49e-5
		10000	5.87	3.51e-5
		20000	12.51	3.46e-5
		50000	34.53	.
High	FMM (N=8)	1000	1.74	8.49e-6
		5000	22.64	6.77e-6
		10000	31.73	5.76e-6
		20000	36.31	5.41e-6
		50000	252.5	.
	And (K=11)	1000	.545	1.00e-5
		5000	5.93	7.95e-6
		10000	9.48	7.61e-6
		20000	16.15	7.40e-6
		50000	66.29	.

Table 6.3: Relative performance of Anderson's Method and the Fast Multipole Method using 1-separation (potential computation only)

Platform	Processors	16K	64K	256K	1024K
Cray T3E	1	19.7	86.4	.	.
	2	7.8	35.6	.	.
	4	4.0	17.7	69.3	.
	16	1.3	5.0	17.9	67.8
	32	.	2.8	12.9	46.4
SGI Challenge	1	77.2	325.8	1251.8	.
	4	21.0	86.7	330.1	.
	8	11.6	44.9	167.4	.
	16	6.7	24.1	87.0	.
PC Network	1	32.4	138.8	538.9	.
	2	17.2	72.4	281.6	.
	4	10.4	39.8	149.5	.
	8	8.6	26.5	89.6	.

Table 6.4: Performance of Anderson's Method (field calculation) with expansion order 9 and 1-separation on a Plummer model.

by a Synoptics 28115 Fast Ethernet switch. Only one processor per node was used in the experiments.

The results of our runs are summarized in Tables 6.4, 6.5, and 6.6. In the runs, we used Cost Zones partitioning for FMM and Anderson's Method, and ORB partitioning for Barnes-Hut. The main observation from the tables is that we obtain efficient parallelism across algorithms and parallel platforms, with speed-up going towards linear as we increase the input size. The speed-up is particularly good for the Barnes-Hut algorithm, which can be very accurately load-balanced fairly easily. Some small improvements in certain cases of Anderson and FMM might be possible by fine-tuning the cost functions associated with the interaction types for each platform. (Our current cost functions were derived on a Sparc workstation.)

The speed-up on 8 processors of the network of PCs was limited to between 6 and 7 due to inefficiencies in the communication phase. Note that these inefficiencies are not due to the bandwidth or latency limitations of the underlying network, which in principle is powerful enough to support nearly linear speed-up. We conjecture that the problem is primarily due to problems with the network cards of the PCs, and maybe to a lesser extent due to the communication algorithm used in the BSP library implementation.

The times spent in the different phases of the algorithm are nearly the same as in the sequential case, which is to be expected given that the parallel code consists of the same basic components, plus some additional communication steps that take only a small amount of time.

Finally, we have also performed parallel runs with other settings of the integration order, and with 2-separation criterium and supernodes. The speed-up results that we observed are very similar to the numbers we have reported here. This, together with the efficient speed-up, implies that the

Platform	Processors	16K	64K	256K	1024K
Cray T3E	1	24.3	99.7	.	.
	2	12.4	50.6	.	.
	4	6.4	27.0	95.6	.
	16	1.9	7.5	28.5	103.5
	32	.	4.3	14.8	51.7
SGI Challenge	1	97.1	412.2	.	.
	4	26.3	111.0	.	.
	8	14.0	57.4	.	.
	16	9.0	31.0	.	.
PC Network	1	42.7	183.3	675.1	.
	2	22.7	94.3	349.8	.
	4	11.9	51.0	182.7	.
	8	10.3	31.9	107.1	.

Table 6.5: Performance of FMM (potential calculation) with expansion order 4 and 1-separation on a Plummer model.

Platform	Processors	16K	64K	256K	1024K
Cray T3E	1	9.400	43.26	.	.
	2	4.587	21.10	.	.
	4	2.213	10.56	46.94	.
	16	0.572	2.69	12.36	53.95
	32	.	.	.	13.49
SGI Challenge	1	17.52	80.4	364.0	.
	4	4.36	20.0	90.3	.
	8	2.22	10.1	45.4	.
	16	1.28	5.6	24.7	.
PC Network	1	7.75	34.21	163.8	.
	2	3.98	17.86	82.4	.
	4	2.16	10.06	43.1	.
	8	1.78	7.48	24.5	.

Table 6.6: Performance of Barnes-Hut with separation constant 1.0 and quadpoles on a Plummer model.

relative performance of the different methods and accuracy settings is similar as in the sequential case.

6.3 Memory and Communication Requirements

We now discuss the memory consumption and bandwidth requirements of our parallel schemes. These two issues are actually closely related in our implementations, since the locally essential trees that are transmitted have to be stored by the receiver for the entire interaction phase. As pointed out by Singh [24], the size of these replicated data sets can be a significant fraction of the total memory consumption, particularly for small memory and problem sizes and larger numbers of processors.

We point out that the numbers we present in this subsection are based on our current partitioning and replication schemes, and that some improvements will be possible by switching to a continuous curve in the Cost Zones partitioning, and by additional pruning of the locally essential trees. Nonetheless, the basic trends should stay the same. In our discussion, we distinguish among the following sources of memory consumption:

- the number of bodies assigned to a processor by the data partitioning,
- the size of the tree before the sending of the locally essential trees,
- the number of bodies replicated on a processor after receiving the locally essential trees, and
- the increase in the size of the tree after receiving the locally essential trees.

In our measurements, a body assigned to a processor requires 128 bytes (including position, field, speed, and previous acceleration, as used by a leap-frogging scheme), a node of the tree created before the replication requires 2056 bytes (including 1536 bytes for the two expansions of length 32), a replicated body requires 32 bytes (only position and charge needed, except for a small number of bodies located in shared leaves), and a node of the tree during replication requires 1288 bytes (no local field needed, except for a few shared nodes). Not included in our numbers are some other sources of memory consumption, including precomputed matrices that may be used to speed up the interactions. While these sources are not completely negligible, they depend on a number of specific design decisions that are mostly orthogonal to the issue of replication.

In Tables 6.7 and 6.8, we present two different views of the memory consumption of Anderson's Method with expansion order 9 and 1-separation. All memory consumption is in bytes per body, The data was measured over a single iteration on a Plummer model, where for each of the four sources of memory consumption, we took the maximum value over all processors.

Table 6.7 shows how the memory consumption increases as we fix the number of particles at $256K$ and vary the number of processors from 1 to 32. In the 32 processor case, the memory consumption per body is about three times larger than in the single processor case, mainly due to the large number of nodes in the replicated part of the tree.

A more optimistic scenario is given by Table 6.8, which shows the memory consumption for 16 processors as the input size increases from $16K$ to $4096K$. While memory consumption is very high for $16K$, it drops to a ratio of less than 1.5 for $4096K$ bodies. (This is a conservative

Source	1	2	4	8	16	32
Bodies before replication	128.0	130.7	130.3	129.4	129.8	131.7
Tree before replication	77.8	78.4	82.8	83.8	93.7	115.4
Replicated bodies	0,0	4.0	18.1	28.2	42.9	67.8
Replicated nodes	0.0	13.4	53.8	98.5	166.8	319.4
Total	205,8	226.5	285.0	339.9	433.2	634.3
Ratio to sequential case	1.00	1.10	1.38	1.65	2.10	3.08

Table 6.7: Memory usage in bytes as a function of the number of processors, for $256K$ particles

Source	64K	256K	1024K	2024K	4096K
Bodies before replication	130.4	129.8	129.8	132.1	132.5
Tree before replication	115.7	93.7	85.6	91.5	88.6
Replicated bodies	83.9	42.9	25.5	21.5	21.1
Replicated nodes	334.9	166.8	94.0	71.3	59.7
Total	664.9	433.2	334.9	316.4	301.9

Table 6.8: Memory usage in bytes as a function of the number of particles, for 16 processors

estimate as the precise ratio depends on the memory consumption of the sequential code on the same number of particles, which we could not measure.)

Note that our numbers are based on a 1-separation criterium, and that they will get worse in the case of 2-separation. We have also not included the numbers for Barnes-Hut, which are slightly better then those for Anderson and FMM.

Finally, the amount of communication in the replication phase is roughly equivalent to the size of the replicated bodies and nodes. However, we found the cost of transmitting the data over the network to be quite small, even if we include the (usually slightly larger) computational cost of inserting the received data into the tree structures.

7 Concluding Remarks

In this paper, we have described portable and efficient parallel implementations of several adaptive N-body methods, including the adaptive FMM and the adaptive version of Anderson's Method. Our experimental results demonstrate that our codes achieve high performance and efficient speed-up across several classes of parallel architectures. A downside of our implementations are the significant memory overheads due to our replication scheme. We are currently working on reducing these overheads, and are also looking at alternative communication schemes that do not suffer from this problem.

Our experimental data only provides a very preliminary direct comparison between the adaptive versions of FMM and Anderson's Method. A more detailed study is needed in order to assess the advantages and disadvantages of the different methods. Such a study would also have to include the recent algorithm of Hrycak and Rokhlin [14], which may give significant performance gains over the other methods. An interesting question in the context of such a comparative study is how to best optimize these methods for current CPUs and memory systems. While we have made an effort to achieve efficiency in our codes, additional improvements are certainly possible, e.g., by carefully tuning for the memory hierarchy.

Finally, it would be interesting to see how our techniques and implementations perform on a real application domain.

8 Acknowledgements

We would like to thank Kevin Lang and Satish Rao for many helpful discussions. Early parts of this work were done while the second author was at the NEC Research Institute, and while visiting UC Berkeley. Access to the SGI Challenge and the network of PCs was generously provided by the NEC Research Institute. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy.

References

[1] C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput.*, 13(4):923–947, July 1992.

[2] Andrew W. Appel. An efficient program for many-body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6(1):85–103, January 1985.

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

[4] Guy Blelloch and Girija Narlikar. A practical comparison of N-body algorithms. In *Dimacs Implementation Challenge Workshop*, October 1994.

[5] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[6] John A. Board, Ziyad S. Hakura, William D. Elliott, and William T. Rankin. Scalable variants of multipole-based algorithms for molecular dynamics applications. In *Proceedings of the 7th Conference on Parallel Processing for Scientific Computing*, pages 295–300. SIAM Press, February15–17 1995.

[7] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing*, 9(4):669–686, July 1988.

[8] Mark W. Goudreau, Kevin Lang, Satish Rao, Torsten Suel, and Thanasis Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12, June 1996.

[9] A. Greenbaum. Parallelizing the adaptive fast multipole method on a shared memory MIMD machine. NYU Ultracomputer Note 162, June 1989.

[10] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[12] P. Hanrahan, D. Saltzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, July 1991.

[13] L. Hernquist. Performance characteristics of tree codes. *Astrophysical Journal Supplement Series*, (64):715–734, 1987.

[14] T. Hrycak and V. Rokhlin. An improved fast multipole algorithm for potential fields. Technical Report RR-1089, Department of Computer Science, Yale University, November 1995.

[15] Y. Hu, S. L. Johnsson, and S.-H. Teng. A data-parallel adaptive N-body method. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM Press, 1997.

[16] Y. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the 6th ACM Symp. on Principles and Practice of Parallel Porgramming*. ACM Press, 1997.

[17] Yu Hu and S. Lennart Johnsson. A data-parallel implementation of hierarchical N-body methods. *International Journal of Supercomputer Applications and High Performance Computing*, 10(1):3–40, 1996.

[18] S. Krishnan and L. V. Kalé. A parallel adaptive fast multipole algorithm for N-body problems. In *Proceedings of the 24th International Conference on Parallel Processing*, pages III:46–51, Oconomowoc, WI, August 1995.

[19] George Lake, Thomas Quinn, and Derek C. Richardson. From Sir Isaac to the Sloan survey: Calculating the structure and chaos owing to gravity in the universe. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, New Orleans, Louisiana, 5–7 January 1997.

[20] Pangfeng Liu and Sandeep N. Bhatt. Experiences with parallel N-body simulations. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, pages 122–131, 1994.

[21] Lars S. Nyland, Jan F. Prins, and John H. Reif. A data-parallel implementation of the adaptive fast multipole algorithm. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 111–123, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[22] K. E. Schmidt and M. A. Lee. Implementing the fast multipole method in three dimensions. *Journal of Statistical Physics*, 63:1223–1235, 1991.

[23] Jürgen Singer. *The Parallel Fast Multipole Method in Molecular Dynamics*. PhD thesis, University of Houston, August 1995.

[24] Jaswinder Pal Singh. *Parallel Hierarchical N-Body methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.

[25] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[26] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Supercomputing '93*, pages 12–21, 1993.

[27] Michael S. Warren and John K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87, 1995.

[28] M.S. Warren and J.K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Supercomputing '92*, pages 570–576, 1992.

[29] F. Zhao and S. L. Johnsson. The parallel multipole method on the connection machine. *SIAM Journal on Scientific and Statistical Computing*, 12:1420–1437, 1991.

[30] Feng Zhao. An $O(N)$ algorithm for three-dimensional N-body simulations. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, October 1987.