DIMACS Technical Report 94-27
May 1994

The Parallel Implementation of N-body Algorithms

by

Pangfeng Liu[1,2,3]
DIMACS center
Rutgers University
Piscataway, New Jersey 08855-1179

[1]Post-doctoral fellow
[2]ONR Grant N00014-93-1-0944, NSF/DARPA grant CCR-89-08285, DARPA contract DABT 63-91-C-0031
[3]DIMACS NSF grant STC–91–19999

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

This dissertation studies issues critical to efficient N-body simulations on parallel computers. The N-body problem poses several challenges for distributed-memory implementation: adaptive distributed data structures, irregular data access patterns, and irregular and adaptive communication patterns. We introduce new techniques to maintain dynamic irregular data structures, to vectorize irregular computational structures, and for efficient communication. We report results from experiments on the Connection Machine CM-5. The results demonstrate the performance advantages of design simplicity; the code provides generality of use on various message-passing architectures. Our methods have been used as the basis of a C++ library that provides abstractions for tree computations to ease the development of different N-body codes.

This dissertation also presents the *atomic message model* to capture the important factors of efficient communication in message-passing systems. The atomic model was motivated by the problem of transferring large messages in a system with limited communication resources and bandwidth at each node. Although the atomic model imposes strict constraints, we show that simple randomized protocols nonetheless provide high communication throughput.

Chapter 1

Introduction

The promises of parallel computers

Massive parallelism is an attractive choice for solving large problems. Improvements in uniprocessor performance notwithstanding, more dramatic gains are achieved by bringing the power of multiple processors together. The largest commercially available machines today have peak rates of over one hundred billion floating point operations per second; this is projected to increase by over an order of magnitude within the next few years. This massive computing power will enable researchers to solve problems that are now computationally intractable.

Many scientific and engineering computations are notorious for their extensive computing requirements. As an example, consider the problem of weather prediction. In 1990 Wilhelmson [25] simulated the evolution of thunderstorms over a region of 5400 square kilometers. On the Cray-2 supercomputer the simulation ran twice as fast as the thunderstorm would evolve in the real world. For a region as large as the continental United States the simulation must run 3000 times faster to achieve the same wall-clock time. For more accurate simulations requiring finer resolution, the computing requirements are even greater. In many applications, the resolution and scale of simulation models that are required for detailed investigation are simply beyond the capabilities of the largest available computers.

There are technological and physical limits to uniprocessor performance that cannot be overcome. For example, clock times cannot be smaller than the response time of electronic circuits, which in turn are limited by physical laws. Parallel computers overcome the inherent performance limitation of uniprocessors by using many processors simultaneously. The recent generation of parallel computers, the Thinking Machines Corporation's Connection Machine CM-5, the Intel Delta Touchstone and Paragon machines, the IBM SP-1, and the Cray Research T3D computers all use microprocessors as their basic processing units; the aggregate performance far exceeds that of a single microprocessor. Parallel computers can multiply on the successes of the microprocessor industry.

Parallel computers have been effectively used to solve many scientific problems. For the most part, computations that have benefited from parallel computers have been well-structured, with regular and static communication patterns, and predictable data access patterns and computational requirements. Examples include several basic linear algebra problems, and iterative methods on static and regular array-structured domains.

The challenges of parallel computers

Not all scientific problems are well-structured. In many applications, the domains are irregular and adaptive; moreover, the changes can be unpredictable. As a result, the communication patterns, data-access patterns, and the computational requirements of individual domains are time-varying. While the underlying systems are, in principle, parallelizable, it is not clear whether the advantages of parallel implementations overcome the costs of parallelization. This dissertation concentrates on issues in implementing such irregular and adaptive computations efficiently in distributed memory parallel computers.

There are several challenges to efficient parallelization of irregular and adaptive computations. First, the entire computation is partitioned among individual processors within a parallel computer. The total execution time will be the time of the processor with the heaviest workload. Any imbalance in distributing the computation tasks will degrade the performance gains from parallel computing. Unlike uniform and regular computations, it may not be straightforward to distribute irregular computations evenly and efficiently among multiple processors. The distribution becomes more difficult when the computation structure changes dynamically.

The second factor is communication overhead. After the computation is partitioned among processors, the necessary data must be transferred to the right processors at the right time. The communication network must keep up with the requests for data, and transfer data to correct destinations. The communication patterns may be irregular and dynamically varying; unlike simple well-structured array computations, compile-time optimizations are not always possible.

The processors must also synchronize their pace so that time-dependent work will be done in correct order. For example, barrier synchronization divides the computation into time-dependent phases; all processors enter a phase only after every processor has completed the previous phase. Deviations in the time spent by different processors in the same phase imply waiting overheads that can add substantially to the overhead due to parallelization.

Finally, the constant and rapid evolution of parallel machines makes efficient implementation a moving target. Different parallel computers have different architectural characteristics, and require different algorithmic design choices to explore their potential advantages. For example, distributed memory and shared memory machines use different mechanisms to coordinate processors, and the requirements for efficient parallel programs on these architectures can be considerably different.

Simulations in Science and Engineering

Supercomputers have changed the the way scientists study natural phenomena. Scientists can now build computational models of complex physical systems; predictions of the models are generated by a computer. A simulation can mimic the development of natural systems: the collisions of galaxies, the evolution of thunderstorms in the atmosphere, or the combustion of fluids in a rocket engine.

Many physical systems are modeled as a collection of bodies which interact with each other, and possibly react to an external field as well. Computer simulations of many-body systems are used to explore the dynamics of the underlying physical systems. Indeed, computer simulations are the only feasible method for studying the dynamics of simple many-body systems. For example, it is well-known that no closed-form solutions are known for the case of three bodies interacting under a gravitational field. In fact, a rich history of many-body calculations in astrophysics has developed over several centuries. Today, supercomputers are routinely used for large-scale astrophysical simulations which require massive amounts of calculation.

Overview

This dissertation studies the various issues involved in efficient N-body simulations on massively parallel computers. Large-scale N-body simulations impose several challenges for distributed-memory implementation, including the need for adaptive distributed data structures, irregular data access patterns, and irregular and adaptive communication patterns. Coping with these issues requires new ways to design data structures and communication protocols for distributed-memory architectures.

While tremendous research effort has concentrated on parallelizing scientific computations with uniform structures, efficient parallelizations of dynamic and irregular computations are not yet well understood. Many high-level languages, including Fortran-D [19], CM-Fortran [48], Vienna-Fortran [17] support parallel operations on uniform parallel arrays. However, none of them supports parallel operations on irregular data structures. Some runtime systems do support run-time pointer (or index) interpretation for remote data access [15, 18, 30, 36], but these approaches take advantage of static data access and communication patterns, and do not provide efficient solutions for dynamically changing data distribution and communication patterns.

This thesis is inspired by the work reported in Salmon's thesis [42] as well as the papers of Warren and Salmon [53, 54]. Salmon implemented a version of the Barnes-Hut algorithm for gravitational N-body simulation and reported results on the NCUBE machine [42] as well as the Intel Touchstone Delta machine [53, 54]. We implement the same version of the Barnes-Hut algorithm, but introduce several new techniques to maintain dynamic data structures and for efficient communication. We report results from experiments on the Connection Machine CM-5. The results demonstrate the advantages of our techniques: design simplicity, better efficiency, and generality of use on various message-passing architectures.

Our contributions cover different aspects of parallel computing. First we present general techniques for efficient implementations of dynamic irregular data structures. We present a method for implicitly representing global tree structures in distributed memory. The trees adjust incrementally as the bodies move in space; the adjustment takes care of smoothing any workload imbalances that might occur as bodies move.

Our methods require very little overhead due to parallelism. The overhead in distributed data structure management, load balancing, and communication is minor compared to the requirements of numerical calculations. The low overhead is remarkable especially since we use the CM-5 vector units to accelerate the numerical calculations.

Our methods have been used as the basis of a library that provides abstractions for tree computations on distributed-memory machines [11]. The goal of the library is to allow different N-body codes to be written at a high-level, independent of the details of data structure management and communication.

The second part of this thesis presents a communication model, the *atomic message model* [33], to capture the important aspects of message-passing systems. The atomic message model is motivated by the problem of transferring large messages in a system with limited resources to store messages in transit, as well as limited bandwidth available at each node to send and receive messages. Resource-efficient communication is essential for N-body simulations, as well as other scientific computations. The atomic message model penalizes algorithms that do not conserve system resources for communication. The atomic model gives realistic performance measurement for message-passing protocols, and provides a guideline for implementing resource-efficient communications. Within the atomic model we prove that simple randomized protocols provide high communication throughput; the randomized protocols suggested by the theoretical results are experimentally observed to be superior to various standard protocols, and are extensively used in the N-body implementation.

Thesis Outline

This thesis is organized as follows. Chapter 2 discusses the N-body problem and reviews previous work. Chapter 3 gives an overview of our implementation techniques. Chapter 4 focuses on our efforts to vectorize the force calculations in N-body simulation. Chapter 5 presents and discusses experimental results. Chapter 6 presents the communication problem encountered in the N-body simulation, as well as the atomic message model. We also analyze the performance of randomized scattering under the atomic message model. Chapter 7 provides a theoretical study of the atomic message model; we present algorithms for backtrack and branch-and-bound search which achieve linear-speedup. Finally, Chapter 8 concludes with a summary of the main contributions and directions for further research.

Chapter 2

N-body Methods

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 using the N-body method the physical system is modeled by a collection of bodies with interactions governed by physical laws generally described by partial differential equations. The equations are solved numerically to obtain the state of the bodies as a function of time. The simulation mimics the state transition of a natural phenomena within a small time interval; this process is iterated to calculate the state of the system at any desired time.

For simplicity we focus on 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 force induced by each of the other $N-1$ bodies. While this method is conceptually simple, vectorizes well, and is the algorithm of choice for many applications, its $\Theta(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of bodies.

Beginning with Appel [4] and Barnes and Hut [7], there has been a flurry of interest in faster N-body 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 [23] presented an $O(N)$ *Fast Multipole Method* which is provably correct to any fixed accuracy. The underlying numerical representations were subsequently refined and simplified by Zhao [55] and Anderson [2]. Recently, Sundaram [46] extended the fast multipole method to allow different bodies to be updated at different rates; this reduces the arithmetic complexity over a large time period.

While the fast multipole method uses only $O(N)$ arithmetic operations, the number of operations to build and maintain the underlying data structures was not considered in the papers cited above. Callahan and Kosaraju [13] developed better data structures to cluster the bodies hierarchically; they bound the time for both data structure and numerical operations by $O(N)$.

On a related note, Reif and Tate [41] show that integrating N bodies with $2^{-N^{O(1)}}$ accuracy through $N^{O(1)}$ time steps is PSPACE-complete.

Despite the differences in asymptotic running times, the overheads in the fully adaptive version of the fast multipole method are substantial and the algorithm by Barnes and Hut continues to be widely used in astrophysical simulations. Several parallel implementations of the Barnes-Hut's algorithm have been reported recently. Salmon [42] implemented the Barnes-Hut algorithm (with quadrupole approximations) on message-passing architectures including the NCUBE and Intel iPSC. Warren and Salmon [53, 54] report impressive performance from extensive runs on the 512 node Intel Touchstone Delta. Singh etal. [44, 45] implemented the Barnes-Hut algorithm for the experimental DASH prototype. This thesis contrasts our approach and conclusions with both these efforts.

Parallel implementations of the fast multipole method have been developed recently as well. Board and Leathrum [26] have implemented the 3D adaptive Fast Multipole Method on shared memory machines including the KSR [26], Zhao and Johnsson [56, 57] implemented their version of non-adaptive 3D multipole method on the Connection Machine CM-2, and Singh etal. [44] have implemented the 2D adaptive fast multipole method on the DASH prototype. Finally, Nyland, Prins and Reif [37] describe a data-parallel implementation of the 3D adaptive Fast Multipole Method using the Proteus prototyping system.

2.1 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 (Figure 2.1). Salmon's algorithm uses the second order quadrupole for approximation. The fast multipole method extends this basic idea to compute interactions between clusters of bodies.

The accuracy of center-of-mass approximation depends on the ratio between the radius of the cluster and the distance from the cluster to where the potential is evaluated. The effect of a cluster can only be approximated by its center of mass if the distance to the cluster is greater than $\frac{r}{\theta}$ where r is the radius of the cluster. The parameter θ controls the error of the approximation.

The Barnes-Hut algorithm organizes the set of bodies into a hierarchy of clusters. To minimize the number of operations, each body computes interactions with the largest clusters for which the approximation can be applied. As shown in Figure 2.2, the algorithm first computes an oct-tree partition of the three-dimensional box (region of space) enclosing the

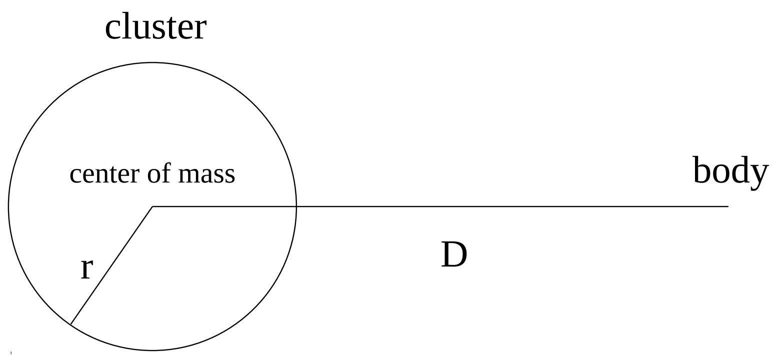


Figure 2.1: Center of mass approximation

```
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 2.2: The Barnes-Hut algorithm

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 2.3 is an example of a recursive partition in two dimensions; the corresponding quad tree, which we call the BH-tree, is shown in Figure 2.4. Alternative tree decompositions have been suggested [3, 13]; the Barnes-Hut algorithm applies to these as well.

The sequential Barnes-Hut algorithm constructs the BH-tree by inserting bodies into the cluster hierarchy one at a time. The ith body is added into the BH-tree consisting of the first $i-1$ bodies. A newly inserted body descends down the BH-tree until it reaches a box of which it is the sole occupant. If a body reaches an existing leaf, the leaf is divided until each of the two bodies is in its own box.

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 sufficiently far away, 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. When the traversal reaches a leaf, a direct two body interaction is computed.

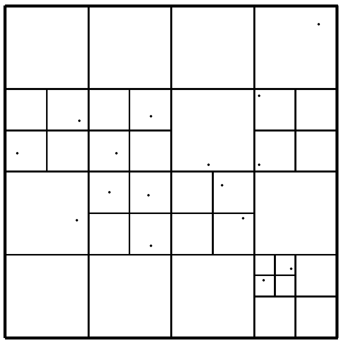


Figure 2.3: BH-tree decomposition

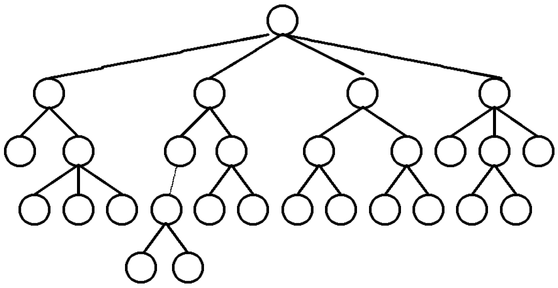


Figure 2.4: BH-tree

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.

Applying center-of-mass approximation in a top down tree traversal reduces the amount of computation in Barnes-Hut algorithm. First, for a body far away from a cluster, the effect of the cluster can be approximated by its center of mass. As a result a simple two body interaction suffices to update the position of the body without calculating individual effects exerted by bodies within the cluster. Secondly, top-down traversal ensures that each body interacts only with the *largest* clusters for which the approximation is valid.

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.

One remark concerning distance measurements is in order. There are several ways to measure the distance between a body and a box. Salmon [42] 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" [42]. In our experiments we use $\theta = 1$; this corresponds to

$\theta = 0.5$ for the original Barnes-Hut algorithm [42].

The overhead in building the tree, and traversing it while computing centers-of-mass and accelerations is negligible in sequential implementations. With ten thousand bodies, 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.

2.2 Parallel Implementations

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.

2.2.1 Issues in Parallel Implementation

Data Structures

There are many difficulties in implementing a dynamic and irregular data structure in distributed memory. First, the large number of BH tree nodes must be equitably distributed among processors. Even distribution of data is essential to solving large scale problems under given resource constraints.

Secondly, data partitioning must preserve data locality. Data should be assigned to processors so that during force computation most of the accesses are primarily in local memory. Otherwise the essential data would have to be fetched into local memory through expensive communication. An inappropriate data mapping which does not preserve locality increases communication overhead and decreases overall performance.

Data locality and even distribution can be contradictory. One can achieve excellent load balancing by mapping data randomly, but lose data locality completely. Or one can assign data to as few processors as possible to improve locality. However the processor utilization will be poor. The partitioning must assign data to processors carefully to keep both properties at the same time.

Thirdly, as a dynamic data structure continuously evolves due to ongoing computation, a static data mapping may not distribute data evenly after a period of time. The mapping must be dynamically updated so that it can adapt to the evolving data structures. For example as bodies move and the distribution of bodies in space changes, the mapping of bodies to processors must be adjusted to ensure balanced distribution.

Finally, the distributed data structure must be consistent with a sequential implementation. As computation proceeds, data elements are inserted and deleted. In a distributed implementation each processor will have only a subset of the entire structure, and the insertion/deletion will have to operate on these local data structures. The collective effect of these local insertions/deletions should be coherent to the result of a sequential implementation.

To sum up, implementing the Barnes-Hut tree in distributed memory requires careful data structure design; the BH-tree structure is adaptive to dynamic and irregular distribution of bodies. The irregular distribution makes data mapping difficult to compute, and the movement of bodies mandates the data mapping to be dynamic.

Computation

The workload of computing new positions must be evenly distributed. The number of floating point operations required to compute the acceleration varies from one body to another. Therefore it is not sufficient to map the same number of bodies to each processor. Instead the same amount of calculation should be assigned to each processor for fast parallel execution.

The Barnes-Hut algorithm uses tree traversal extensively for locating essential nodes, however, tree traversal is expensive and hard to vectorize. We can traverse the children of a node only after we know that the node is too close to apply center-of-mass approximation. This makes tree traversal inherently sequential.

Communication

The communication pattern for adaptive Barnes-Hut algorithm is irregular and dynamic. For example, a processor must collect all the essential data to update the positions of bodies. Since the BH-tree is distributed, each processor must request some essential data from other processors. The distribution of bodies can be highly irregular, so the set of processors where the remote data is located cannot be described by a uniform pattern. Moreover, the irregular communication pattern can change from iteration to iteration because of continuous movement of bodies.

The irregular and dynamic communication pattern is hard to optimize. The communication pattern depends on the input data and is unpredictable at compile time. Some run-time support systems can optimize irregular but static communication pattern by observing the pattern at the first iteration, and then computing an optimized communication schedule which is used in later iterations. The cost of computing the optimized schedule is amortized by the large number of iterations and can be compensated by the increase of communication throughput. However, N-body problems have dynamic communication pattern and the cost of finding a good communication schedule cannot be amortized into later iterations. In this case using run-time systems to find an optimized schedule is not cost-effective.

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. For example, the hierarchical oct-tree is used to represent clusters of bodies in the fast multipole method as well. Each cluster is characterized by a *multipole expansion* computed in an upward phase up the the tree. This is followed by a downward phase to combine multipole expansions and to propagate them to the leaves. At the end of the downward phase each leaf has enough information to compute the force induced by bodies in the *far field* – the area outside of this leaf and its neighbors. Finally a body computes its acceleration by combining the effect from the far field and its neighbors. Although the algorithms differ in many details, the issues of building, traversing, and maintaining tree data structures are similar. In fact, the methods presented in this thesis are being used to develop efficient parallel implementations of the fast multipole method as well.

2.2.2 Related Work

In this section we describe the important aspects of Salmon's thesis [42] which motivated us initially, as well as the more recent reports of Warren and Salmon [53, 54], and of Singh etal. [44, 45]. We also point out the differences of our techniques from these approaches.

Salmon [42] was the first to implement a parallel N-body simulation with adaptive hierarchical tree structure. The implementation is based on a modified version of the Barnes-Hut algorithm. Instead of using only the center of mass to represent a cluster, the quadratic terms in the Taylor's expansion are used for force computation.

Salmon and Warren [53] implemented the modified Barnes-Hut algorithm on a 512 node Intel Touchstone Delta machine. Two very large scale simulations of the Cold Dark Matter model with 17.15 million bodies were reported. This is the largest astrophysics N-body simulation ever done [53].

Salmon [42] and Warren and Salmon [53] weight each body by the number of interactions in the previous time step. The volume enclosing the bodies is then recursively decomposed by orthogonal planes into regions of equal total weight. Figure 2.5 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.

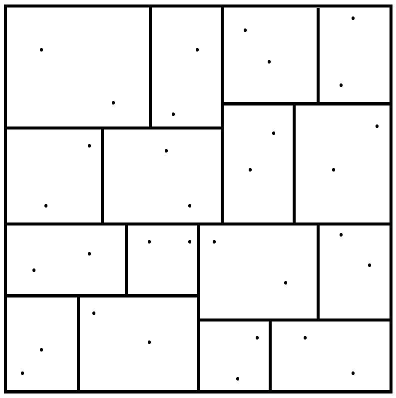


Figure 2.5: A two dimensional orthogonal recursive bisection

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.

The locally essential trees are built by a communication protocol based on the hypercube topology. After each processor builds its local tree, tree nodes are exchanged to build the locally essential trees. The flow of information follows the dimension order of the hypercube. Each processor computes the set of local tree nodes that might be essential to *any* processor in the other half of the hypercube, and sends the data to the corresponding processor differing in dimension 0. After iterating through all dimensions every processor has all of its essential data in its local tree and local essential trees are constructed.

The global tree is never built, either explicitly or implicitly. 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 "store-and-forward" communication requires two tree traversals for each iteration: the first traversal locates tree nodes that may be essential to the domain on the other side of hypercube, and the second traversal removes those that will not be essential on this side of hypercube. Data forwarding takes 40% of the total communication time in Salmon's implementation [44].

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.

Our implementation separates the tree building and essential data transferring processes.

The BH-tree building routines can be abstracted out and used by other adaptive N-body algorithms. The Barnes-Hut tree is built from individual pieces of data structure in each processor. These pieces are then adjusted and combined to form the global data structure.

Our implementation uses a sender-driven protocol that is independent of network topology. Each tree node first determines the volume of space for which it is essential, then computes the set of processors that might have bodies in that region. The sender initiates the communication and the information is directly sent to the destinations. It is essential that shapes of processor domains be regular so that we can quickly compute the set of processors that might have bodies the tree node is essential to.

The data structures are incrementally updated. The BH-tree is adjusted after the new positions of bodies are computed. Those that move out of their original BH boxes are sent to the new positions in the BH-tree. Those move to another processor domain will be inserted into the local trees of corresponding processors. If the movements of bodies cause imbalance of workload, the necessary ORB bisectors are adjusted to maintain the even distribution of workload.

More recently, Warren and Salmon [54] reported a modified algorithm which uses a different criterion for applying center-of-mass approximations. The error bound of the new opening criterion is carefully analyzed for center-of-mass approximation. A similar Cold Dark Matter model with 8.8 million bodies was simulated on a 512 node Intel Touchstone Delta machine. The implementation sustains very high speed at 5.8 Gflop/s.

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 which is the sequence of octants the body will fall into in the finest resolution of BH-tree. The actual position of the body in the BH-tree is a prefix of its key. The 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.

The advantages of this new approach are balanced by other factors. First of all, the advantages of sender-directed communication are lost. The bodies are partitioned according to their positions in the BH-tree, therefore the processor domains have complicated shapes, rather than regular boxes as in ORB. It will be difficult for a BH node to compute the set of processors where its data is essential.

Secondly, the force computation stage is slowed down by communication. Because of the complications of sender-driven protocol, the essential data must be transferred on demand. Each body traverses the BH-tree, and requests essential data that are not in local memory. The fine-grain demand driven communication is not efficient because of two-way message passing and the startup cost of large number of messages.

Warren and Salmon solve the problems by using multiple threads to pipeline tree traversals and to update accelerations. If a body does not obtain an essential node in its local tree, it initiates the communication to get the data, and continues tree traversal on some other part of BH-tree. Moreover, control can be transferred to tree traversals for other bodies while the current one is waiting for its essential data. This multi-threaded method effectively hides latency by switching control among thirty tree traversals. The communication throughput is increased by packing requests/data to/from the same processor into longer messages. However, the multi-thread approach complicates program control structures. Explicit multi-thread control makes the program more complicated and less transparent.

Finally, the data structures are not maintained incrementally. The program must sort all the keys to distribute bodies and build BH-trees for every iteration. Chapter 5 gives more details on timing results and comparisons.

The DASH shared-memory architecture group at Stanford [44, 45] has investigated the implications of shared-memory programming for the Barnes-Hut algorithm. 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 [54].

The shared memory provides a single address space for all processors. Memory coherence is enforced by hardware and memory cache is used to improve data access efficiency. In his thesis Singh [44] concluded that a shared memory implementation can exploit temporal locality by caching the essential data for different bodies. He claims that a shared memory implementation can provide both programming simplicity and better performance than an explicit message passing implementation.

The arguments in [44] about the advantages of shared-memory over message-passing implementations are based largely on comparisons to the initial implementations of Salmon [42] and Warren and Salmon [53]. Since our message-passing implementation is considerably simpler and more efficient, the import of the arguments of [44, 45] 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 unclear.

An additional example is the Barnes-Hut tree building. In his thesis [44] Singh reported two algorithms for building global BH-tree. The first algorithm has processors insert their bodies into a shared tree structure concurrently. Unfortunately, processors must interlock one another to modify the shared tree nodes during the insertion, thus the performance is not satisfying. The second method works much like a distributed memory algorithm; it builds a local tree in each processor, and combines them into a single global tree using a technique similar to the one in our distributed memory implementation. The second distributed memory style algorithm runs twice as fast as the first pure shared memory method. Contrary to the claim that shared memory implementation provides better performance [44], the distributed memory style of programming used in the second algorithm provides better efficiency than the first pure shared-memory implementation, even on a shared-memory architecture.

Finally, the fine grain and demand driven communication in shared memory machines may not be efficient in a large system. The cache coherence mechanism on current shared memory machines is driven by cache misses and each communication only transfers a small amount of data. The communication is in small units and no aggregated data transferring is possible. On the other hand the message-passing communication can pack data going to the same destination into longer messages for better efficiency.

To sum up, N-body tree codes can be efficiently implemented in distributed memory, and the details of direct message passing can be hidden from users by using various parallel programming tools. Our experimental results suggest that direct message-passing does not imply large run-time overhead, or impede the development of efficient parallel N-body programs.

Chapter 3

The Parallel Implementation

The design of the parallel code is guided by the principle of bulk computation and communication. The overheads in communicating and operating on data can be amortized by processing data in bulk quantities. 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.

To process data in bulk, we separate control into a sequence of alternating computation and communication phases. The communication phase fetches all the remote data essential to the following computation phase. Thus a computation phase starts with all the necessary data available in local memory, and is not slowed down by communication. Alternating computation and communication also results in simple control structure with cleanly defined functionality for each phase.

This chapter provides an overview of our parallel implementation based on the above observations. We begin with the techniques of maintaining dynamic irregular tree structures in distributed memory. The tree building process is divided into a computation phase which computes local information, and a communication phase which combines local information into a global Barnes-Hut tree. A subsequent communication phase collects all the data necessary for the following force computation phase. Finally, the process of updating positions is also divided into communication and computation phases.

Figure 3.1 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. Step 1 builds an implicit representation of the global Barnes-Hut tree from the local trees. Step 2 combines remote essential data with local trees into local essential trees, so that step 3 can access all the essential data to compute the acceleration of bodies. Steps 1.2, 3, and 4 require no communication; step 3 is the most time-consuming step. Steps 5 and 6 incrementally update the data structures to conform to the new distribution of bodies.

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

3.1 Data Partitioning

The force calculation phase dominates simulation time; it is essential that the work during this phase be evenly distributed among processors. The workload of a body is the number of floating point operations required to update its position. The workload of a body depends on its surrounding and varies from one to another. As a result equal amount of workload, not number of bodies, should be assigned to each processor.

We use orthogonal recursive bisection (ORB) to distribute bodies among processors. The space bounding all the bodies is partitioned into as many boxes as there are processors, and all bodies within a box are assigned to one processor. The bodies assigned to a processor are called *local bodies* to that processor. At each recursive step, the separating plane 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. Figure 3.2 shows an two dimensional ORB decomposition of 16 processors.

The ORB decomposition can be represented by a binary tree, the ORB tree, a copy of which is stored in every processor. Each internal node of the ORB tree represents a bisector plane and the domain it bisects, and each leaf is a processor domain. 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 bodies.

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-

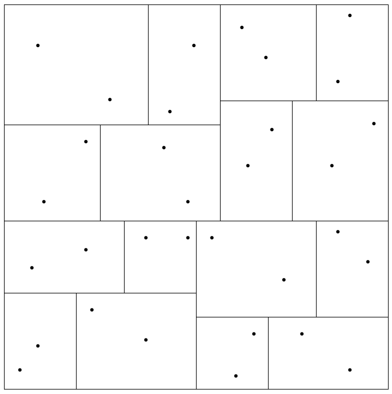


Figure 3.2: A two dimensional orthogonal recursive bisection

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 [44], the cost of incremental load-balancing is negligible as we will see in Chapter 5.

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. For high simulation accuracy, the bodies should move gradually. The load distribution among processors will only fluctuate a small percentage through iterations. As a result even a static ORB can balance workload for several iterations. When the ORB can no longer balance the workload, we can still adjust it with minimum changes so that it becomes balanced again.

The ORB decomposition is incrementally updated in parallel as follows. The ORB tree structure is statically partitioned among processors. At the end of a time step each processor computes the total number of interactions used to update the state of its local bodies. 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%. A top-down search on ORB tree marks 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. Also the top-down search reduces the number of initiators, as well as the number of regions that require remapping.

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 bisector plane so that the right amount of workload can be shifted to the under-loaded child. If we can quickly determine the weight within the parallelepiped between the old and any given plane, we can find the correct bisecting plane by a binary search. The workload within a parallelpiped is, in turn, computed by traversing the local BH-tree.

[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.

3.2 Building the BH-tree

Unlike the first implementation of Warren and Salmon [53], 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 [11] for further details concerning a library of abstractions for N-body algorithms.

3.2.1 Representation

We represent the global BH-tree as follows. Since the oct-tree partitioning always divides a box at the center, 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.

3.2.2 Construction

The adaptive Barnes-Hut tree construction begins with building a local tree in every processor using local bodies only. The local tree building process is as same as in sequential adaptive Barnes-Hut method. However, the local trees are built with respect to the entire domain, not to its own processor domain. A local tree describes the distribution of bodies within a processor domain, and represents the local view of a processor.

The local trees will not, in general, be structurally consistent. For example, Figure 3.3 shows a set of processor domains that span a BH-node; the BH-node contains four bodies, one per processor domain. Each local tree will contain the BH-node as a leaf; but this is inconsistent with the global tree.

The next step is to make the local trees be structurally consistent with the global BH-tree. This requires adjusting the levels of all leaf nodes which are split by ORB bisector planes. A similar process was developed independently in [44]; an additional complication in our case is that we build the BH-tree until each leaf contains up to L bodies. Choosing L to be much larger than 1 speeds up the computation phase, but makes level-adjustment somewhat tricky. We made this modification mainly to adapt Salmon's multipole expansion [42] for better simulation accuracy. The same approach is also used in Greengard-Rokhlin's Fast

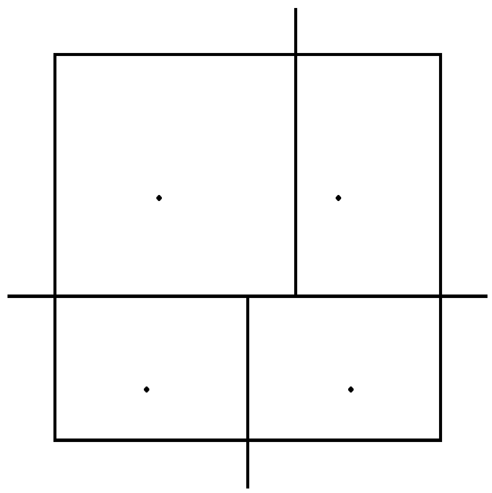


Figure 3.3: An example that a leaf in the local trees is actually an internal node in the global tree.

Multipole Method [23].

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

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

3.2.2.1 Level Adjustment

The owner of each internal tree node receives all the contributions to that node only after all local trees are consistent. For example, a body α in level ℓ of a local tree may actually be in a deeper level $\ell' > \ell$ in the global tree. If the level of α is not adjusted down to ℓ', the owners of tree nodes on the path from ℓ' to ℓ will not receive contribution from α, and will not have correct global information.

For efficiency we locate and adjust only those bodies that may not be in the correct levels. If a processor domain covers an entire leaf, the bodies within the leaf require no adjustment[2]. Otherwise this leaf is a *broken leaf* since its domain spans multiple processors. It is easily seen that only those bodies within broken leaves require level adjustment. For each broken

[2]Recall that up to L bodies can be in the same leaf.

leaf u, the *covering processors* $\mathcal{CP}(u)$ are those whose domains overlap with u. The level of a body α is determined solely by its covering processors.

Each body in a broken leaf has a *level-info*; this is the deepest known leaf into which this body must go, and up to $L-1$ other bodies within the leaf. Initially, the level-info of a body α within a broken leaf u contains u and the local bodies of u besides α.

We find the correct level for a body by refining its level-info. Each body α receives a level-info from each member of $\mathcal{CP}(u)$. The level-info received from p contains the leaf that p would insert α into, as well as p's local bodies that are also in that leaf. Each covering processor computes the level-info based on its own distribution of bodies within u. A body refines its initial level-info using those received from other covering processors; it always chooses the deeper leaf as the new level-info. Each body can then go down to the correct level indicated by its final level-info.

Request-and-Answer Model The level-info refining process can be described as a "*request-and-answer*" process. If a processor p locates a broken leaf u in its local tree, p will send a *request* for u to all the other members of $\mathcal{CP}(u)$. We will call p the *requester* for u, and all the other members of $\mathcal{CP}(u)$ the *responders*. Upon receiving a request, a responder will send back an *answer* to describe the distribution of bodies within u in its domain. By receiving information from responders, a requester can know the distribution of bodies outside its domain.

For ease of explanation we focus on a body α in a broken leaf u within processor p. First p sends a request for u to every responder. The request contains u and the local bodies of p within u. When a responder q receives this request, it searches its local tree and computes a *modifier* for each body in the request. A modifier for α from a responder q contains the deepest leaf that q would insert α into, and other local bodies from q that are within the same leaf.

A responder computes modifiers by assuming all the bodies in the request will be inserted into its local tree; each body descends the responder's local tree as if it were actually inserted. When a body reaches a leaf, it will report the leaf, along with the responder's local bodies within the leaf, as the modifier. The "pseudo" insertion of bodies may modify a responder's local tree; the leaf u may be further divided because there could be more than L bodies in u including those from the request.

Since all bodies in the request are already in a leaf u, the modifier is always as deep as u. Responders also use this information to update their local trees. For instance, if a responder receives a request which is a descendant of a leaf in its local tree, the responder will push the leaf down at least to the level of the incoming request.

Once the responders compute all the modifiers, those for the same request are packed into an *answer* and sent back to the requester. A body in a broken leaf will receive a modifier from each responder. The modifiers refine the initial level-info of the body so that the final level-info contains the correct level.

The level-info of a body α is refined as follows. First, if the leaf in the modifier is as same as the one in α's current level-info, the bodies in the modifier are added into the level-info.

This may cause more than $L-1$ bodies in the level-info so the leaf will be further divided. The child node containing α becomes the new leaf in the level-info, and those bodies not in it are discarded. The refinement continues until the level-info contains at most $L-1$ bodies. Secondly, If the leaf in the modifier is deeper, the level-info adopts that leaf and discards the bodies not within it. The update then follows the first case.

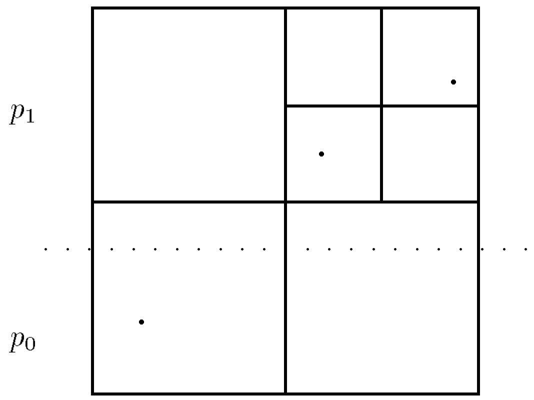


Figure 3.4: The requester p_0 needs the answer from p_1 to push its body down to the correct level.

One might argue that since incoming requests update local trees of responders, the bodies will eventually be pushed down to the correct level, making the answering process unnecessary. Unfortunately, the requester/responder relation is not symmetric. For example, in Figure 3.4 processor p_0 considers the whole tree node as a broken leaf and sends request to processor p_1. However, p_1 does not have any broken leaf that has bodies inside, and will not send out any request. As a result the critical information that more than one body are in this tree node will not be sent to p_0 unless p_1 answers the request from p_0. The bodies in p_0 will not be pushed down automatically just by processing incoming requests.

The correctness of final level-info does not depend on the order in which modifiers are received. Consider a body α in level ℓ of the global tree. First, the final level-info of α will not be deeper than ℓ, independent of the order the modifiers are applied. If the level ℓ leaf is divided, it must contain more than $L-1$ bodies that α knows of. This contradicts the assumption that α is in level ℓ of the global tree. On the other hand, if the final level-info of α is in level $\ell' < \ell$, it means α can only find at most $L-1$ bodies in the level ℓ' tree node. From the definition α is in a broken leaf u no deeper than ℓ' before the adjustment. Thus the level ℓ' node is completely covered by $\mathcal{CP}(u)$, and any bodies within the level ℓ' tree node should have been discovered. This also contradicts the assumption that there are more than L bodies in the level ℓ' tree node.

3.2.2.2 Combining Partial Information

Each processor computes its contribution to the global tree in a computation phase. For ease of explanation we will use *node-info* to denote the center of mass and multipole moments. The node-info in an internal BH-node is calculated by combining the node-info of its children. This computation is similar to the center-of-mass calculation in sequential Barnes-Hut algorithm.

The final step of global tree construction is to combine partial contributions to the same node into global node-info. Similar to the level adjustment, a local tree node has correct node-info if it is completely covered by one processor domain. Otherwise the local node-info from the covering processors of a broken BH-node must be combined into global node-info.

The covering processors of a broken tree node must agree on which processor the local node-info should be sent to. We formalize this concept as the *owner* of a tree node. The owner of a tree node is the processor that contains its geometric center. This mapping function can be computed easily by different processors with consistent results using the ORB tree. All the local node-info for the same BH-node will be sent to the owners and combined into global node-info.

The incoming partial node-info is combined into local trees as follows. If the incoming node is already in the local tree, its partial node-info is added into the tree node. However, a processor p can be the owner of a node u which is not in p's local tree; p may contain the center of u but does not have any body in u. In this case p creates u in its local tree to store the global node-info.

A tree node u is a *representative* if it is in the local tree of the owner of u; representatives contain the global node-info values. The distributed global Barnes-Hut tree has representatives as internal nodes and the bodies as leaves. Notice that we do not explicitly store the global Barnes-Hut tree as a separate structure. The representatives are a subset of local tree nodes. We can access any global tree node by determining its owner, then fetch the representative node in the owner's local tree.

3.2.3 Incremental Tree Update

After the bodies move to their new positions, the Barnes-Hut tree must be updated so that it becomes consistent with the new distribution of bodies. The common approach in adaptive N-body methods [7, 42, 44, 53, 54] is to rebuild the entire Barnes-Hut tree. In contrast our implementation dynamically adjusts the BH-tree to conform to the new distribution of bodies. To capture the dynamics of highly accurate N-body simulations, the time interval should be small enough so that the bodies change their positions gradually. Thus the shape of the BH-tree evolves slowly, so it would be more cost-effective to incrementally update the tree rather than rebuilding it.

There are two kinds of adjustment on the Barnes-Hut tree. First a body may move into another processor domain and has to be moved from one local tree to another. Secondly, a body may remain in the same processor but move into a new tree node. If there is no other body within the leaf the body left, then the leaf must be deleted, along with some of its ancestors. If the body joins a leaf with already L bodies inside, it must be divided until no more than L bodies are in any new leaf.

By allowing up to L bodies in one leaf we reduce the number of bodies moving into new BH-nodes, and increase the efficiency of incremental update on the BH-tree. When L increases, the average size of leaf nodes increases, making it less likely that a body moves into a new leaf in a single time step. The reduced chance of a body moving into a new tree node improves the efficiency of incremental updates on the BH-tree. Chapter 5 compares the efficiency of incremental tree updates versus rebuilding the tree.

3.3 Locally Essential Trees

Once the global BH-tree has been constructed it is possible to start calculating accelerations. A top-down BH-tree traversal (similar to the one in sequential Barnes-Hut algorithm) collects tree nodes essential to calculating acceleration. However, a processor may require essential tree nodes that are not in the local tree. Or the local tree has the nodes, but they are not representatives and do not have global node-info. In either case the processor sends a request to the owner asking for the essential data. The tree traversal continues after the owner returns the essential node-info.

The naive strategy of traversing the tree, and transmitting data-on-demand, has several drawbacks. First each processor sends and receives twice as many messages as the number of remote data because of two-way communication. The startup cost for sending large number of messages translates into huge communication overhead. One way to hide communication latency is to execute multiple tree traversals concurrently [54]. However, the multi-threaded method requires a complicated control structure. The two-way fine-grain communication either makes the communication overhead prohibitive or increases the programming complexity.

Secondly, a processor may request a BH-node that does not exist. When a tree node is missing from a local tree, it could be that there are bodies inside the node but they are not within this processor's domain, or there is nothing in it at all. We cannot distinguish two cases unless we send a request to the owner of the node. This uncertainty complicates program control structures and increases communication overheads.

The implementations of Singh [44] and Salmon-Warren [54] use this "transfer-data-on-demand" approach to collect essential data. Singh's implementation is based on shared memory architecture. All references of remote data are accomplished by implicit communication. The memory cache on each processor stores remote data and complicated hardware maintains cache coherence among processors. The fine-grain demand-driven communication requires complex hardware to channel data among processors efficiently. Warren and Salmon's implementation [54] performs up to thirty tree traversals concurrently so that when one is blocked by communication, the others can still continue. The tree traversal, communication for remote data, and force calculation are all combined together to hide latency. The complicated multi-threaded control structure switches among multiple tree traversals and requires extensive bookkeeping.

It is significantly easier and faster to first construct the locally essential trees. Each processor first locates all the information deemed essential to other nodes, and then sends long messages directly to the appropriate destinations. Once all processors have received and inserted the data received into the local tree, all the locally essential trees have been built.

The owner of a tree node sends information only to a region called *influence ring* – the possible positions of bodies that the tree node is essential to. Consider a tree node u and its parent v. Let B_u be the region within which the approximation cannot be applied on u. B_v is defined similarly for v. Those bodies outside B_v should apply approximation on v instead. The bodies within B_u could not apply approximation on u either. Thus u is essential to only those bodies within the annular region $B_v - B_u$.

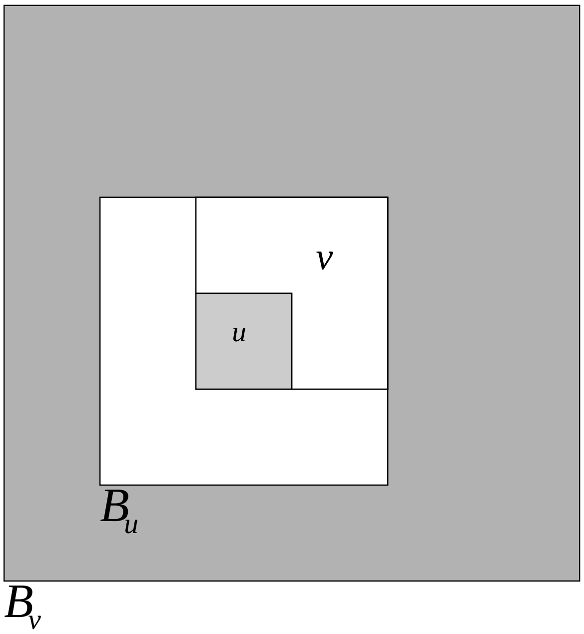


Figure 3.5: Influence ring of u ($\theta = 1$)

The owner of a tree node sends information only to processors whose domains overlap with the influence ring. The destination set can be easily computed from the influence ring and the ORB tree. Notice that the destination set includes processors to which the tree node *may* be essential. Some of the processors may not have bodies in the influence ring and do not need u as an essential data. Nevertheless every processor receives all the essential data it needs.

The one-way communication for essential data is implemented as a "sender-oriented" protocol. The *sender* denotes the owner that sends out essential data, and the *receivers* are processors to which data is essential. Instead of letting receivers initiate the fine-grain demand-driven communication [44, 54], the sender figures out its receivers and sends the information directly. The sender-oriented protocol avoids the inherent sequential access pattern in receiver-oriented communication. In a sender-oriented protocol each owner decides where to send its information independently.

3.4 Calculating Accelerations

Once the local essential trees are in place, we can update the positions of bodies by traversing local essential trees as in sequential Barnes-Hut algorithm. The tree traversal may encounter tree nodes which do not have global information; they are not representatives, nor essential tree nodes received from owners. The tree traversal can safely skip these nodes because they cannot possibly be the essential data to any local bodies. As a result the parallel implementation can actually speed up the tree traversal by ignoring some tree nodes.

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

Grouping bodies does increase the number of calculations, but it also makes them more regular. More significant is the reduction in the time spent traversing the tree. This idea of grouping bodies was earlier used by Barnes [6].

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

A judicious choice of caching strategy is necessary to ensure that cache maintenance overheads do not undermine the gains elsewhere. It is also important to order the different groups such that the total number of cache modifications is minimized. Our strategy is to pick a space-filling curve; the groups are chosen in their order along the space-filling curve. Chapter 4 provides more details of the force calculation stage.

3.5 Reducing Communication Times

Each communication phase can 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.

A better method is to use a series of global reductions on 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 [51] and the number of channels is limited.

Instead, we used a protocol which alternates sends with receives (Figure 3.6). 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 Chapter 6 we develop the *atomic message model* to investigate this phenomenon. Consistent with the theory, we found that sending messages in random order worked best.

```
all_to_some_communication
     generate all messages;
     compute the number of incoming messages;
     while there is message to send/receive
          if there is incoming message
               receive incoming message;
          if there is message to send and resource to send it
               send the message;
     endloop
```

Figure 3.6: The all-to-some communication protocol

By deferring message sending, information going to the same destination can be packed into long messages to improve communication efficiency. The fine grain, demand driven approach may not be suitable for transferring large amount of information. For example, in Warren and Salmon's implementation [54] each processor traverses the BH-tree and demands remote essential data. In order to avoid the problems of this fine grain, demand driven communication, thirty tree traversals are executed concurrently so that the requests/data to/from the same processor can be aggregated.

3.6 Summary

Our implementation introduces two novel ideas: building the global Barnes-Hut tree by level adjustment of local trees, and representing the global tree implicitly by having an easily-computed owner for each global tree node. Level adjustment provides a clean and efficient mechanism to combine individual tree structures together. The concept of owners gives a simple representation of distributed data structures by implicitly embedding global data within local data structures.

We also introduced the sender-oriented protocol for collecting essential data. The owners of tree nodes determine where to send the information independently, and initiate data transfer as one-way communication. This protocol does not use any particular communication topology. The essential data is sent directly to its destination.

All the data structures are incrementally updated. The local BH-trees are incrementally updated to be consistent with the new distribution of bodies. Similarly, the ORB decomposition is incrementally updated to conform to changes in workload distribution.

Chapter 4

Efficient Force Computation

Calculating interactions among bodies is the most time consuming phase of the Barnes-Hut algorithm. Warren and Salmon [53, 54] report that more than 85% of time in their implementation is devoted to force calculation. A simple body-to-body interaction requires thirty floating point operations. The more complicated quadrupole approximation takes more than seventy operations [42, 53].

There are two aspects of force calculation: tree traversal and two-body calculations. Tree traversal identifies essential nodes for calculating the acceleration of a body. Two-body calculations are used between the body and each of its essential nodes. Although the latter requires more floating point operations than tree traversals, the calculation is relatively simple and easy to vectorize.

Tree traversal is inherently sequential and hard to vectorize. The opening tests must be applied to BH-nodes sequentially down the tree; the opening test will not be applied on children unless the parent fails the approximation criterion. Because of this dependency it is difficult to vectorize the opening tests efficiently even on a level-by-level basis. In addition, the shape of the BH-tree can be highly irregular and unpredictable; therefore we cannot vectorize the opening tests by assuming the shape of the BH-tree beforehand.

Tree traversal is also expensive. Warren and Salmon reported that tree traversal uses more than 34% of the time in their latest implementation [54]. A tree traversal has to apply an opening test on every BH-node it encounters, and each test consists of distance calculation and possibly square root computation. The large number of floating point operations in the opening tests make tree traversal even more expensive.

In his thesis [44] Singh surveyed many different approaches for vectorizing tree traversal and force calculation, and concluded that none of them provides sufficient speed up for conducting large-scale N-body calculations on vector supercomputers. Hernquist [24] suggested an algorithm that vectorizes the interaction calculations between the current body and the essential data in the same level, and computes the overall acceleration by computing the interactions one level at a time. Makino [34] suggested an algorithm that vectorizes tree traversals for different bodies. Finally Barnes [6] suggested an algorithm which reduces the number of tree traversals by computing accelerations on a group of bodies at a time. Each member of the same group use the same essential data for force calculation. These techniques speed up the the computation by only a factor of five on vector supercomputers [44]. We use Barnes' technique of grouping bodies. Each of the largest tree nodes that have at most G bodies is considered a group[1]. The implementation details of grouping are in Section 4.5.

We avoid expensive tree traversals by caching essential data. By caching the essential data for one group in a software cache, the next group can access many of its essential nodes in cache, thereby avoiding tree traversal. The essential data cache is initialized for the first group using a tree traversal, then incrementally updated to contain the correct essential data for the group whose acceleration is being evaluated.

There are two necessary conditions for caching to be efficient. First, the sequence in which groups of bodies are considered must be such that consecutive groups share most of their essential data. If the cache hit rate is high, i.e. the percentage of essential data that can be reused is high, then caching pays off and the cost of maintaining the cache will be much less than doing tree traversal for each group.

The second requirement of efficient caching is a fast cache modification algorithm. The cache update routine must decide which portions of cache are invalid for the current group, and quickly replace them with correct essential data. Cache modification must be efficient since the cache is modified for each group.

To summarize, we must answer the following questions in designing an efficient caching algorithm for essential data.

- In what sequence should the groups be traversed so that consecutive groups of bodies share most of their essential data?

- How do we quickly mark and replace cache nodes which are not essential for the current group of bodies?

We provide answers in the following two sections.

4.1 Ordering of Bodies

The group sequencing problem can be stated as follows. Given the BH-tree, find a traversal order which minimizes the number of modifications necessary to update the essential data cache. For ease of exposition we will assume in this section that each leaf contains a single body and that each group contains one body; as we shall see, the results extend easily to larger group sizes.

An important consideration in ordering the bodies is that the order must be induced by a traversal of the tree. The reason is that the bodies appear as leaves of the tree; an arbitrary ordering of bodies would require additional complexity and overhead. Fortunately, a recursive tree traversal, wherein all the bodies in a subtree are visited before a sibling of

[1] Assuming the group size G is larger than the maximum number of bodies in a leaf.

the root of the subtree is visited, preserves locality. All bodies in one octant are visited before entering a different octant.

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

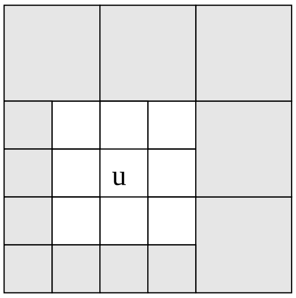


Figure 4.1: The influence ring of a node u can be partitioned into twelve squares.

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

Proof.

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

Second, when θ is 1, the influence ring of any tree node u coincides with a BH-node boundary; the influence ring can be partitioned into twelve squares as in Figure 4.1 and each square corresponds to a possible BH-tree node. If the corresponding tree node of a square does not exist, then a leaf must cover this square, as well as some other adjacent squares. In any case, twelve tree nodes suffice to cover any influence ring.

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

Finally, the tree size is proportional to the number of bodies. The only complication is that two bodies that are very close together can form a long chain in the tree (see Figure 4.2). However, this chain can be lumped into a single node because the tree nodes along the chain

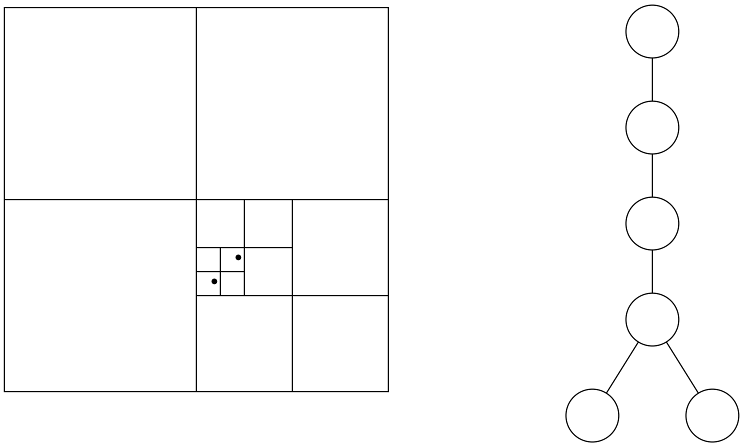
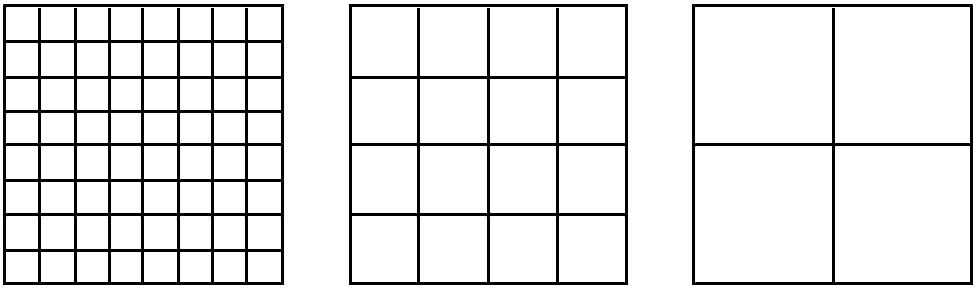


Figure 4.2: A chain of two near-by bodies

represent the same cluster. As a result the size of BH-tree is $\Theta(N)$ and the theorem follows. ■



0-boundary 1-boundary 2-boundary

Figure 4.3: Boundary lines in different levels.

Suppose the bodies are uniformly distributed in a unit square, we show that the number of cache modifications is bounded by $O(N)$ for any θ. We first define some notations. The unit square is uniformly refined until no more than a constant number of bodies is in any bottom level box (Figure 4.3). Since the bodies are uniformly distributed the number of bottom level boxes is $\Theta(N)$. A rectangle is $m \times n$ if its length and width consist of m and n bottom level boxes respectively. A partition line is on *i-boundary* if it is between two $2^i \times 2^i$ BH-tree nodes.

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

Proof. We decompose the rectangle into layers of increasing width, with thinner layers on the outside. If a boundary line of the rectangle is not on 1-boundary, we cut a strip of

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

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

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

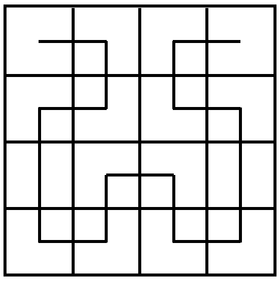


Figure 4.4: Peano-Hilbert sequence

The case of non-uniform distribution and $\theta \neq 1$ remains open. In this case we can bound the total distance covered under a specific tree traversal. While this does not directly bound the number of cache modifications, the intuition is that keeping the average distance between consecutive bodies should keep the number of cache modifications small. In particular, we use a recursive tree traversal corresponding to the Peano-Hilbert curve in Figure 4.4. The worst-case length of a space-filling curve for N bodies in the unit square is $\Theta(\sqrt{N})$ [38]. In fact it has been established that the length of the Peano-Hilbert curve is always within a $O(\log N)$ factor of the optimal TSP tour in d-dimension [8, 10, 38].

Theorem 3 *[38] Suppose that N bodies are distributed within the unit square (alternatively, the unit cube). Then the total distance covered by the Peano-Hilbert traversal is $O(\sqrt{N})$ ($O(N^{\frac{2}{3}})$).*

4.2 Cache Modification

A good traversal sequence alone does not guarantee efficient caching of essential data. The cache modification routine must efficiently update the contents of the cache as well. For large scale N-body simulations the number of cache modifications is enormous because all but the first group require cache update.

Each cache update has two phases. First we decide which nodes in cache are essential for the new body (marking phase); next we remove inessential nodes and bring in essential nodes that are absent from the cache (replacement phase).

The marking phase divides cache data into three categories: "too large", "too small" and "just right". The first category denotes those internal nodes that are too large to be approximated. The second category are those that are so small that their ancestors can be approximated. The rest are the data that can be reused. The category of a node can be easily computed by applying opening tests to the node and its parent. Before the acceleration calculation starts, those data that are not "just right" should be replaced.

The replacement phase uses two operations to update cache: *expand* and *shrink*. A expand operation divides a "too large" node u into a set of "just right" nodes which form a frontier in the subtree rooted at u. A shrink operation combines a group of "too small" nodes into their common ancestor as the correct essential node for the current body. Figure 4.5 describes the caching algorithm in details.

Because of the dynamic insertions and deletions of data, the most suitable data structure for implementing essential data cache is a linked list. To implement efficient shrink operations, all the descendants of a particular tree node must be located and removed from cache very quickly. If the cache data structure maintains a left-to-right order among cached tree nodes, the descendants of any tree node will appear consecutively, and can be quickly replaced by the appropriate ancestor. Using a linked list to store the essential data, an expand operation can replace a tree node with a set of nodes while a shrink does the contrary, both without disturbing the left-to-right order among essential nodes.

4.3 CM-5 Vector Units and Force Calculation

Although a linked list implementation of essential data cache makes efficient update operations easy to implement, it does not vectorize well. The linked list does not occupy contiguous memory and cannot take advantage of fast vector operations. On the contrary, an array implementation can exploit the potential performance gain by allowing vector instructions operating directly on contiguous memory locations.

This section describes the SPARC vector accelerators of the Connection Machine CM-5, and how it affects the choice of data structure and cache update algorithm.

```
compute_acceleration
   loop through all bodies
         if the current body is the first one
            initialize cache by a tree traversal;
         else
            for each BH-node n in cache
                  determine the category of n;   /* marking phase  */
                  switch (category) /* replacement phase */
                     case TOO LARGE: /* expand */
                        do a tree traversal from n;
                        insert the essential data into cache;

                     case TOO SMALL: /* shrink */
                        find the ancestor that is the essential data;
                        remove descendent of this ancestor from cache;
                        insert the ancestor into cache;

                     case JUST RIGHT:
                        skip n;

         /* compute acceleration */
         for each essential node in cache
            compute the interaction between the body and the node;
```

Figure 4.5: The caching algorithm for force computation.

4.3.1 Vector Units on CM-5

The accelerator hardware of the CM-5 consists of four vector units (VU) on each processing node (PN) [49, 52]. Each vector unit retrieves the same instructions from a SPARC instruction unit through a 64-bit bus. Therefore the vector units execute the same computation synchronously on different data. The data may be located in the 128 single-precision registers of each VU, or through various addressing modes in 8 Mega bytes of private memory in each VU. The VU instructions can be vectorized so more than one arithmetic computation/memory accessing can be accomplished by a single vector instruction.

The vector units provide tremendous computing power. The accelerator executes instructions in pipelined fashion: that is, it attempts to begin the execution of each element of a vectorized arithmetic or memory instruction at each clock cycle. When the pipelines on the four vector units are completely full and all the operands are in vector registers, the vector units operate at the theoretical peak rate of 128 Mflop/s [52].

4.3.2 Using Vector Units for Force Computation

The CM-5 vector units provide a feasible way to speed up force calculation. The interactions between a body and each of its essential data are calculated by the same formula. As a result four vector units can execute the same instructions simultaneously to compute the force induced by different essential data. The parallelism among vector units and the pipelined floating point operation hardware within each vector unit can greatly improve the efficiency of force calculation.

An array implementation of essential data cache provides much faster data access for vector units than a linked list. When the essential data cache is implemented as a linked list, the indirect data access through pointers is expensive and does not have data locality. On the other hand, when the essential data is stored in an array, the four vector units can access different blocks of the cache in parallel. In addition, the vector units allow an instruction itself to be vectorized, so each vector unit can load an array of data with given memory stride into its registers in one vector instruction.

The vector units can also speed up the marking phase. The opening test uses the same formula for all cache data, thus the vector units can mark the cache in the same data parallel fashion as in the interaction computation. The performance gain is more significant when the cache data is stored in fixed memory stride of an array structure.

4.4 Data Structure Issues

The major problem in fully utilizing vector units to calculate acceleration is that the cache modification and interaction computation prefer different data structures for cache implementation. A linked list implementation supports efficient expand/shrink operations but could not exploit the advantage vector units can provide. An array implementation makes it easy to use data parallel vector operations, but difficult to expand/shrink cache data dynamically. A good data structure should ensure that the shrink and expand operation can easily maintain the left-to-right order of cache elements, and the vector units have fast access to essential data for interaction computation and cache marking.

To strike a balance between the efficiency of data access and dynamic cache modifications, we use an array to store essential data, and preserve the left-to-right order by maintaining a linked list within the array. The shrink and expand operations can be implemented efficiently by standard techniques of maintaining a linked list within an array. At the same time the vector unit can efficiently access essential data located at fixed memory stride.

The essential data cache is a linked list within the array; each data has a pointer to the next essential data in the left-to-right order. The free array elements are also linked together as a free list. An expand operation fills information into elements of the free list and links them into appropriate locations in the essential data list. An shrink operation does the contrary. Both operations maintain the left-to-right order so the cache update routine can go through the essential data list and apply expand/shrink operations efficiently.

Unlike cache modification, cache marking and force computation need not follow the left-to-right order while accessing essential data. For example, the accelerations due to each of the essential data can be calculated and added in any order. In addition, the category each cache data belongs to can be computed independently. In other words, cache marking and force computation can use vector operations to process blocks of consecutive array elements concurrently. The dynamic expand/shrink operations may leave free array elements in some blocks passed to vector units for processing. These "holes" have no effects on cache marking, and will not change the results of force computation if they contain unit operand of interaction computation.

The free array elements among valid essential data waste essential resources. The vector units mark or calculate forces on a block of array data at a time, up to the last block containing any essential data. If the percentage of free array elements in these blocks increases, the number of blocks vector units have to process also increases. Consequently vector units waste time in processing and memory in storing these holes that do not contain valid data.

Our caching method reduces the percentage of holes by always allocating the free array element with minimum index when one is needed. The intuition is that by filling the lower index portion first, the cache will be more compact and will not grow beyond a certain limit unless it cannot find any free cell within that limit.

To implement the "minimum index first" heuristic, we use a heap to maintain the indices of free array elements so that the minimum one can be found in constant time. When an expand operation demands free cells, it simply uses and removes the minimum indices from the heap. On the other hand, a shrink operation inserts the indices released from the essential data list into the heap. All the heap operations can be efficiently implemented as a d-heap within an array [47].

There are several drawbacks in the simple approach of maintaining free indices as a heap. First of all, not all of the free indices have to be stored in the heap. If we keep track of the maximum block number that has essential data inside (denoted by b), then only those free cells in block 1 through b have to be in the heap. Those free cells with higher indices will not be selected by the "minimum index first" heuristic.

Secondly, the "minimum index first" requirement is unnecessarily strong. Since the vector units process essential data in blocks, any free index in the block with minimum block number suffices; there is no difference in which cell we allocate as long as it is in the block with minimum block number. The number of blocks will not grow beyond a limit unless no free cell can be found before it.

Finally, neither "minimum index first" or "minimum block first" heuristic guarantees the minimum number of blocks containing essential data. The cache modification may release a lot of array cells while processing the end of the essential data list. These free cells can scatter anywhere in the array and there is no expand operation to reclaim them.

```
delete_min_index(heap)
   if (heap is empty)
         increase b by one;
         for all indices i in block b
            insert_index(i, heap);
         return delete_min_index(heap);
   else
         assign the minimum index in the heap to i;
         housekeeping to maintain heap order;
         return(i);
```

Figure 4.6: The heap deletion routine

Our caching method avoids the drawbacks of the naive heap implementation. First, the implementation keeps only the necessary indices in the heap. Initially the block number b is set to 0 and the heap is empty. If an expand operations cannot find free indices in the heap, it increases b by one and inserts the indices in block b into the heap. The reduced number of heap elements improves the performance of heap insertions and deletions. Figure 4.6 outlines the heap deletion routine. A standard heap insertion routine can be found in [47].

Secondly, we use block number instead of array index to determine which free cell should be used first. The heap contains only block numbers that have free cells. The indices of free cells within the same block are stored in a linked list accessible from the block number in the heap. The number of keys in the heap and the time for heap insertion/deletion are further reduced because all free cells in one block are treated as one key in the heap.

Finally, a garbage collection routine eliminates holes in the cache array when necessary. Recall that b is the maximum block number that has essential data inside. If there are enough holes in the b blocks to make up at least an entire block, the garbage collection routine shifts them towards the end of the array so that maximum number of blocks of free cells can return to the free list. In other words, the garbage collection guarantees the minimum number of blocks for vector units to process.

Garbage collection works by moving essential data with higher indices to fill holes with lower indices. Let b_r denote the number of blocks just enough to store the entire essential data list. The garbage collection routine scans through block $b_r + 1$ to b for essential data, and moves them to free cells with block number no larger than b_r. The indices of these free cells can be found in the heap. Finally b_r replaces b as the new block number, and those keys greater than b_r are removed from the heap.

Garbage collection minimizes essential data copying. A naive implementation computes ordinal number for each essential data, then move the ith essential data to the ith element of cache array. This method leaves no holes but requires far more data copying than necessary. In contrast our garbage collection only moves data when necessary; those essential data in block 1 to b_r will not be affected. The garbage collection avoids unnecessary data copying by allowing holes in the cache array, and still minimizes the number of blocks vector units will process.

4.5 Computing Acceleration in Groups

By computing acceleration for multipole bodies as a group, Barnes's force computation algorithm [6] reduces the number of tree traversals without loss of accuracy. The bodies in one group share the same essential data collected from a single tree traversal. To maintain accuracy, the opening test divides a cluster if *any* body in a group cannot apply approximation on it. The essential data is refined into a "lowest common frontier" for all bodies in the group; every body in the group can apply multipole approximation on every cluster in the essential data set.

The new opening test determines whether to divide a cluster in a conservative way. The opening test measures the distance from the *boundary* of the bounding box, instead of individual bodies within the group. Therefore if any body in a group needs to divide a cluster, it is divided. This is a conservative approach because some tree nodes may be unnecessarily opened because the actual distance from the bodies to the cluster might be larger.

Our implementation adopts the grouping technique quite easily. The same space-filling curve in Figure 4.4 determines the order by which the groups of bodies calculate their accelerations. Let G denote the group size. If the traversal reaches a tree node with at most G bodies inside, they form a group. The bounding box of the group is simply the domain of the tree node. The space-filling sequence maintains adjacency among consecutive groups. The caching method incorporates the grouping method by simply changing the opening criterion to the conservative test mentioned earlier.

The grouping technique has its drawbacks. First, if the group size increases, so does the average size of the bounding box and the number of clusters opened unnecessarily. The increased bounding box size decreases essential data cache hit rate, and the extra computation decreases force computation efficiency. These problems do not occur when acceleration is computed for one body at a time. In addition, the bodies within the same group must compute the interaction among themselves by direct pairwise method. The cost for solving the interaction within a group is $\Theta(G^2)$ and becomes unacceptable when G is very large. When the group size is extremely large, the grouping method degrades to the $O(N^2)$ direct pairwise algorithm.

As an idealized example, suppose the space-filling sequence partitions the bodies into $\Theta(\frac{N}{G})$ groups, each with $\Theta(G)$ bodies. The total number of interactions computed is $\Omega(N \log N + GN)$; the second term counts the number of interactions within every group. Therefore the average number of essential nodes for a group is $\Omega(\log N + G)$, and the total number of essential data for all groups is $\Omega(\frac{N \log N}{G} + N)$. The number of cache modifications, bounded by $O(N)$, is asymptotically smaller.

4.6 Summary

Caching essential data is a practical alternative to tree traversal for collecting essential data. Unlike tree traversal, caching vectorizes well. As long as we maintain high cache hit rate by exploiting spatial locality among bodies (for example, using the space filling curve sequence), caching is cost-effective. Chapter 5 gives comparisons from experimental results.

Software caching is different from the hardware caching in shared memory machines. Software caching is a mechanism that speeds up data access *within* the local memory of one processor. It breaks the access restriction of pointer-based data structures and provides direct access to data that are frequently used. On the other hand, hardware caching speeds up data access *among* local memory of different processors.

The vector units on CM-5 dramatically affect the algorithm and data structure designs in our implementation. We chose an array implementation over a linked list to improve data access throughput to/from vector units. However, to preserve the left-to-right order the essential data is implemented as a linked list within the array. Nevertheless, the holes in cache array, a complication from implementing a linked list in an array, reduce efficiency. Thus a "minimum block first" heuristic is used to address the problem. To sum up, the data structure designs take advantages of fast vector operations of CM-5 accelerator, and various algorithmic techniques overcome complications that occur because of dynamic cache modifications.

Chapter 5

Experimental Results

This chapter presents experimental results and compares them with related work. Our platform is the Connection Machine CM-5 with SPARC vector units [49]. Each processing node has 32M bytes of memory and can perform floating point operations at peak rate of 128 Mflop/s [52]. We use the CMMD communication library (version 3.0) [51]. The vector units are programmed in CDPEAC which provides an interface between C and the DPEAC assembly language for vector units [50]. The rest of the program is written in C.

The experiments sketched here included three input distributions: the uniform distribution, the Plummer distribution [1] with mass M = 1 within a sphere, and two Plummer distributions at a colliding course. The Plummer model has very large density in the center. All three cases contained about 10 million bodies.

5.1 Breakdown of Running Time

Figure 5.1, 5.2, and 5.3 show the time spent per phase for the Plummer model, two Plummer models at a colliding course, and uniform distribution. The time can be classified into four categories. The first is the time to manage the distributed Barnes-Hut tree. This includes level adjustment, incremental BH-tree update, and combining the local trees into the global representation. In all three cases less than 5% of the total time is spent for these activities.

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% of the total time for both Plummer model cases, and 3% for uniform distribution.

The third category is time to compute accelerations. This category includes the time for vector units to compute interactions among bodies, 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% of the total time goes to cache modification for Plummer model, 6% for two Plummer models at a colliding course, and 1.5% for uniform distribution.

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%. As a result the amortized cost for remapping is extremely small per simulation step.

The implementations sustain 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 Mflop/s in the interaction computation. The rest of the overhead is less than 13% for a Plummer model, 15% for two colliding Plummer models. For the uniform distribution the corresponding figure is less than 9%.

The performance figures from our implementation compare favorably with those reported by Warren and Salmon [53, 54] (see Figure 5.4). With uniform distribution of bodies, our implementation spends 91% of the total time performing interaction computations, and uses less than 9% of the time to manage Barnes-Hut tree, construct locally essential trees, prepare essential data via caching, and balance workload. The small overhead implies significant efficiency especially since the interaction computation is already speeded up by the use of vector units. Finally 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.

5.2 Tree Management

Level Adjustment and Global Tree Construction

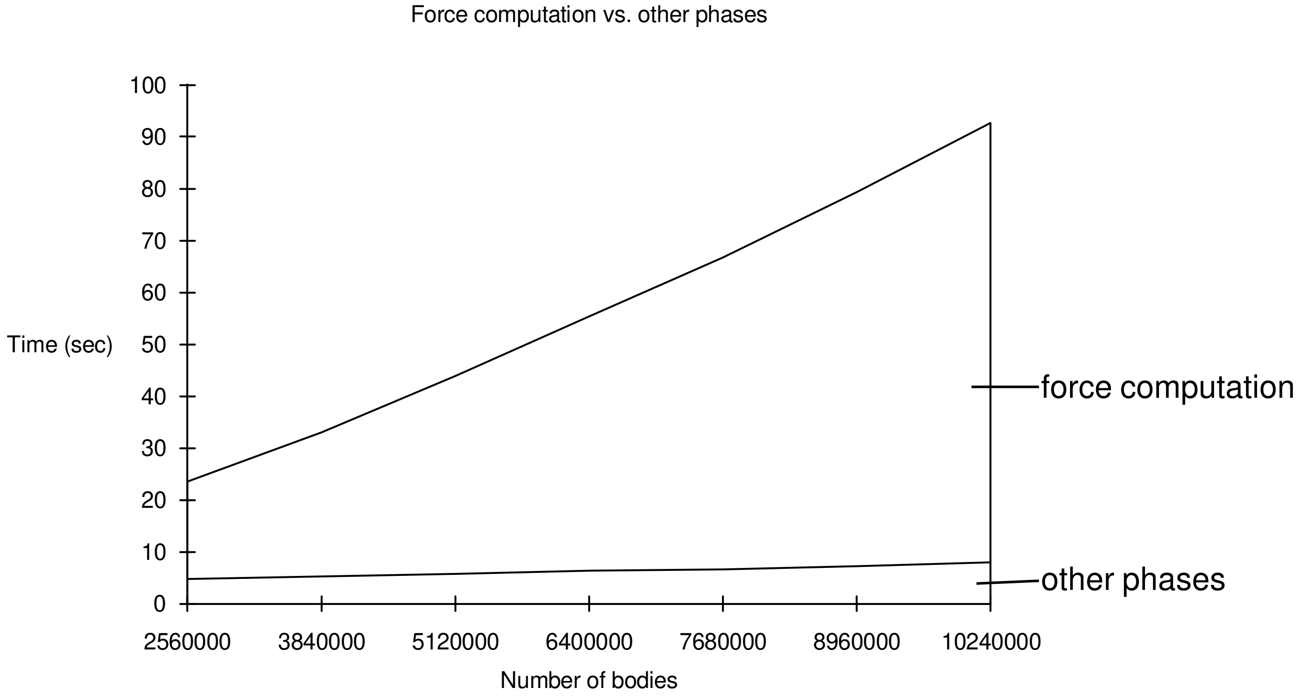
Although level adjustment in the construction of global BH-tree involves two-way communication, it takes very little time to complete for the following reasons. First, our implementation adjusts only the bodies in broken leaves – leaves that span the domains of multiple processors. In very large scale simulations the average size of a leaf is very small and the percentage of broken leaves is under 1% (Figure 5.5).

Figure 5.5 shows the percentage of bodies that require level adjustment as a function of granularity. As the number of bodies per processor increases, the percentage drops because the BH tree partition is further refined. As a result it is less likely for a leaf to span over domains of multiple processors.

Secondly, since the average size of a broken leaf is small, it is very unlikely that a broken leaf can overlap with more than two processor domains. Consequently it is likely for a requester to send requests to only one responder for a broken leaf. The complicated case of more than two covering processors are required to determine the level seldom happens.

Finally, a responder can modify its local tree based on requests it receives. In large scale simulations, most of the bodies that need adjustment will be pushed to the correct level during the request-and-answer communication. The case where a body has to be adjusted after the communication (Figure 3.4) will not happen very often in large scale simulations.

The time to construct representatives is also small for similar reasons. In large-scale



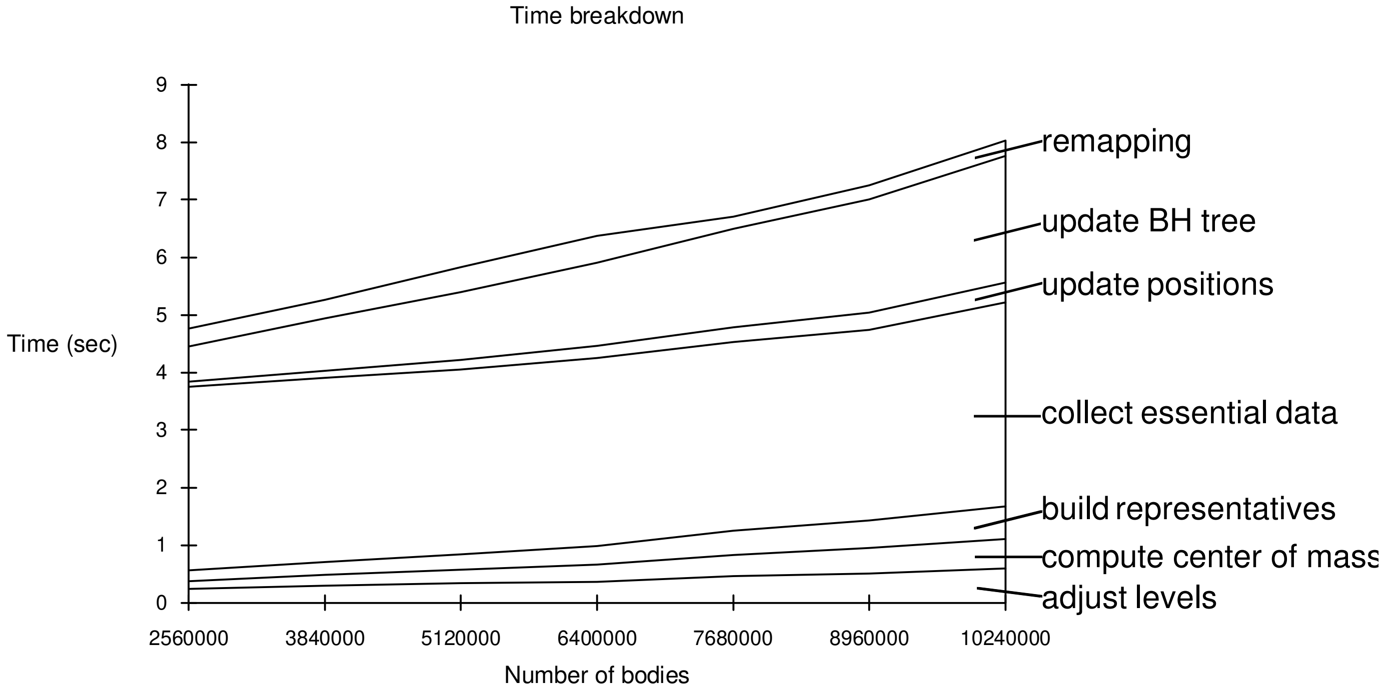
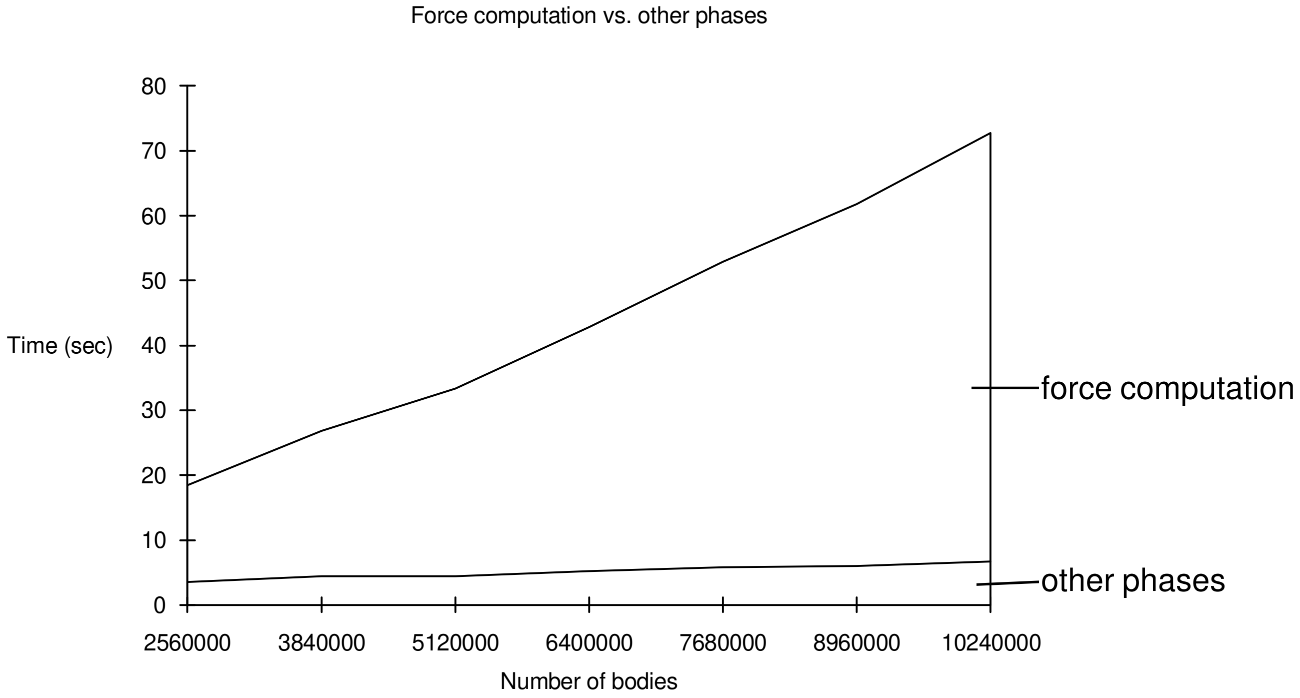


Figure 5.1: Time breakdown for each phase on a 256 node CM-5. The input is a Plummer model with up to 10 million bodies. The "build representatives" phase combines partial contributions into global node-info. The "update BH-tree" phase moves bodies into their new BH-nodes, or to a new processor domain.



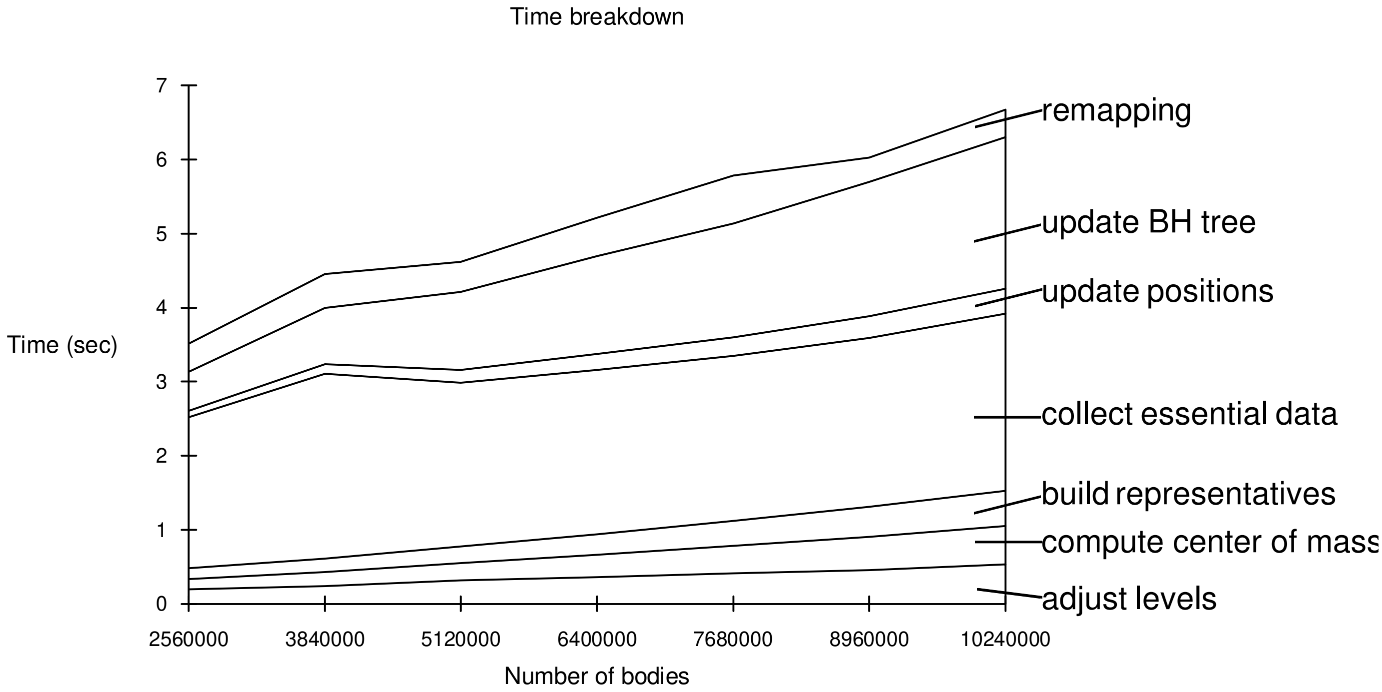


Figure 5.2: Time breakdown the simulation of two colliding Plummer models on a 256 node CM-5.

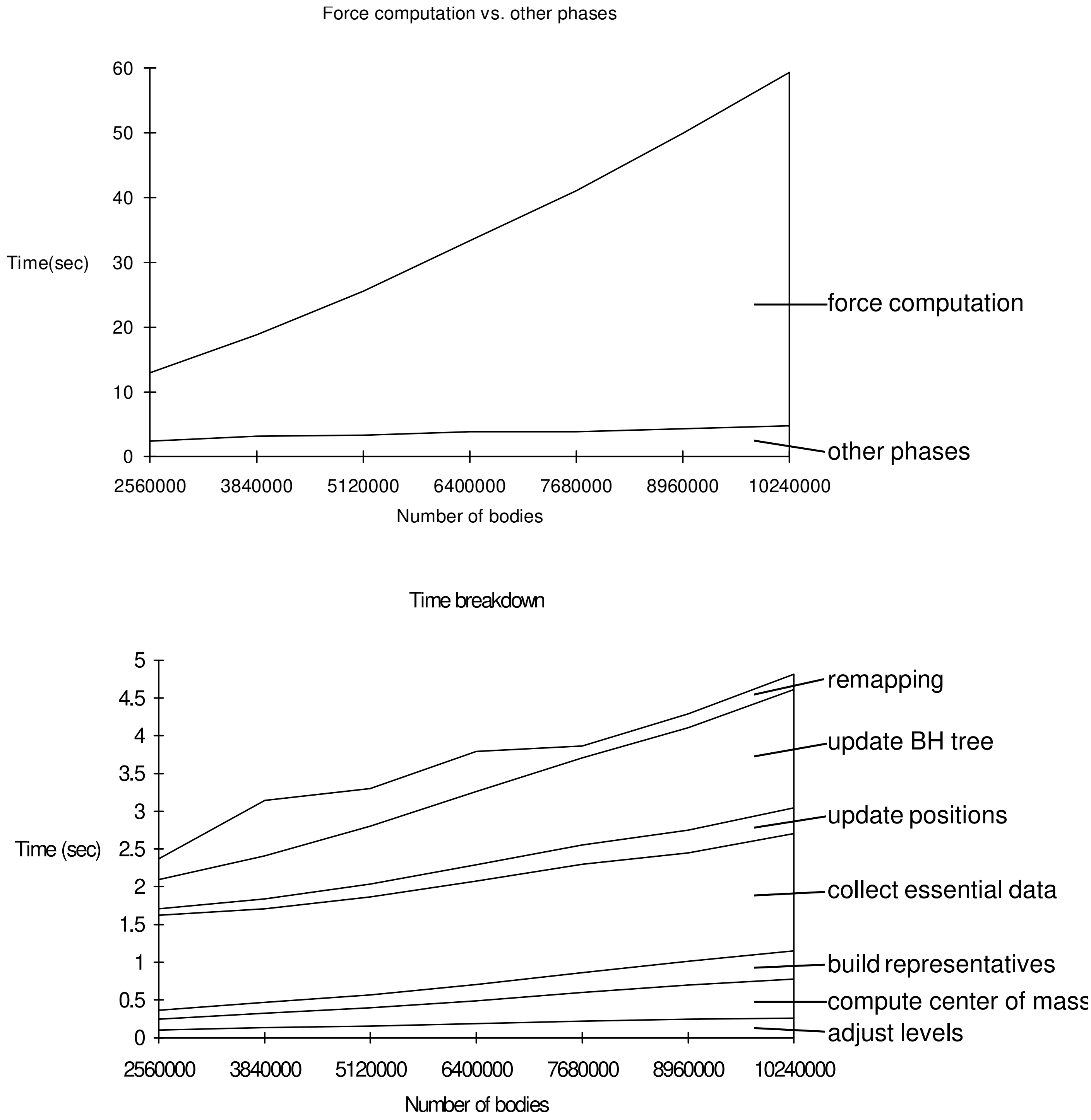


Figure 5.3: Time breakdown for each phase of the simulation on a 256 node CM-5. The bodies is uniform distributed in a sphere.

	Warren-Salmon-92	Warren-Salmon-93	Liu-Bhatt
machine configuration	512 node Delta	512 node Delta	256 node CM-5
number of bodies	8.8 million	8.8 million	10 million
input distribution	uniform	uniform	uniform
time per simulation step (sec)	77	114	59
time % of interaction computation	85%	47%	91%
other overhead	15%	53%	9%

Figure 5.4: Comparison of Warren and Salmon's implementations [53, 54] and ours. The time percentage of force computation in [53] includes tree traversal and the actual percentage of overhead is higher.

simulations, the percentage of tree nodes that span more than one processor domain is small. Intuitively when the tree nodes decreases their sizes as they go deeper in the BH-tree, it becomes very unlikely that any ORB bisector can cut through these very small tree nodes. Only those tree nodes in the top levels of BH-tree need to construct their representatives via communication. Those on the deeper levels of the BH-tree have correct global information right in the local tree, and do not have to construct representative nodes via communication.

Incremental BH-tree Update

Our incremental tree structure is more efficient than the conceptually simpler method of [54]. 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 [44]. 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.

Figure 5.6 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 tree node is inserted into the BH-tree, the implementation must allocate memory for it. The memory management routine (malloc()) provided by UNIX 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.

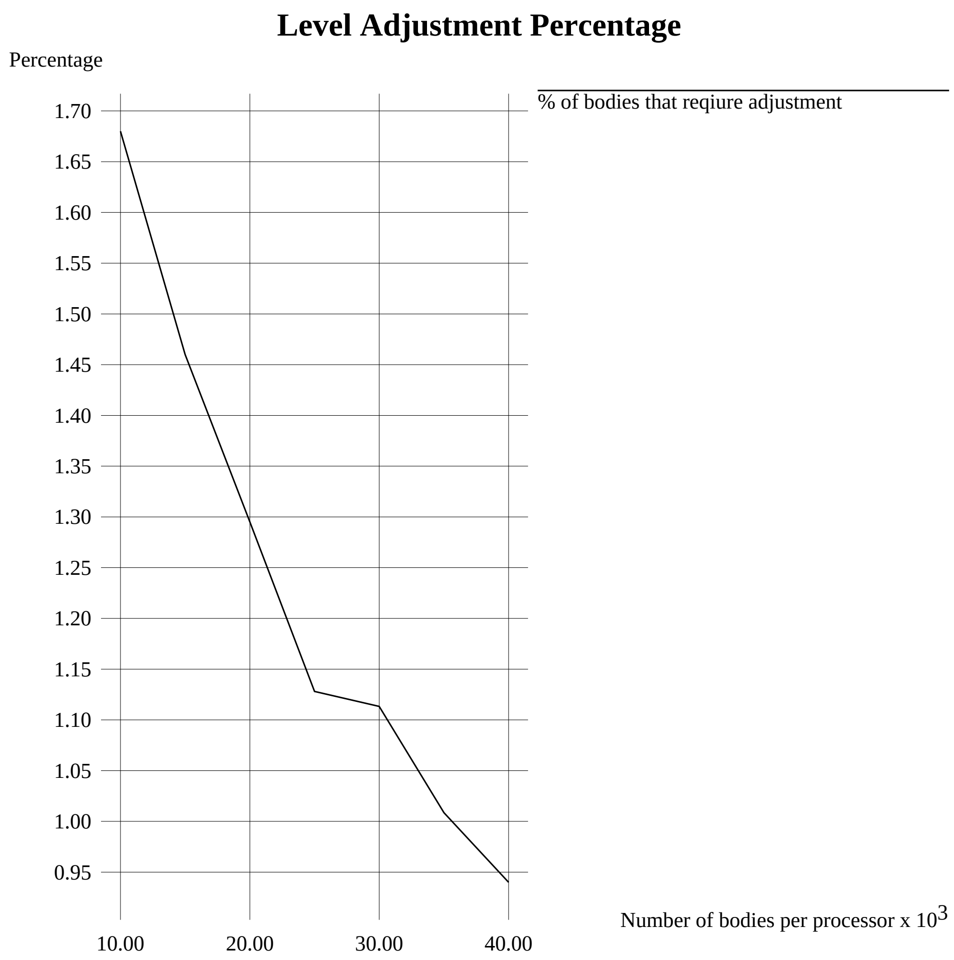


Figure 5.5: The percentage of bodies that require level adjustments for different granularity. The number are taken from a Plummer model simulation with up to 10 millions bodies on a 256 node CM-5.

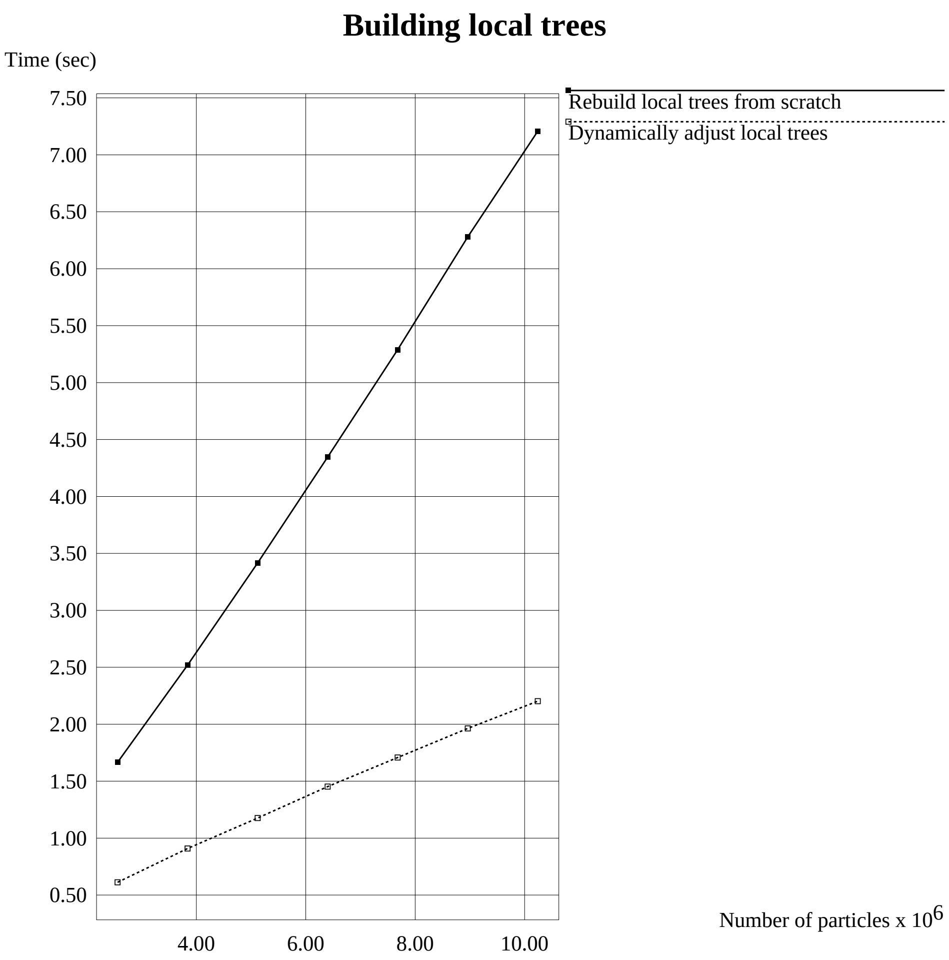


Figure 5.6: The time to adjust or rebuild the BH-tree under different granularity. The data is taken from a Plummer model simulation with up to 10 million bodies on a 256 node CM-5. The maximum number of bodies in a leaf is 8.

5.3 Communication and Locally Essential Trees

In [44] 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, and the overhead is very small with respect to the overall execution time, even on a large 256-node CM-5 configuration.

Singh suggested that the total size of locally essential trees will be much larger than the size of actual global tree tree [44]. From the experimental results the amount of duplicated information in the locally essential trees is not significant compared with the global tree size. Figure 5.7 shows that the ratio is relatively small and decreases as the granularity increases.

The main reason for this small locally essential tree to global tree ratio is that most of the tree nodes are bodies, and the majority of them are not duplicated in the locally essential trees. In our implementation a leaf node can have up to L bodies. This effectively reduces the number of internal nodes which are more likely to be duplicated than a single body, as well as the total tree size. Although some bodies close to ORB bisectors will be duplicated in locally essential tress, they constitute only a very small fraction of total number of bodies. And the fraction decreases as the granularity and the percentage of bodies far away from bisectors increases.

Salmon [42] reported that the complexity overhead, namely the locally essential trees construction, is the major source of inefficiency. We do not see this happen in our experiments; the implementation spends less than 9% of the time to build the essential trees. The reason may be that Salmon's method requires two tree traversals for each of the $\log P$ dimensions in a P processors system, and the hypercube protocol use "store-and-forward" communication. In contrast we use only one tree traversal to locate essential data that have to be sent out, and the information is sent to the destinations directly.

5.4 Force Computation

Figure 5.8 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, so does the size of essential data cache. Therefore the time for the 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 the uniform 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 increase of G. Finally, the

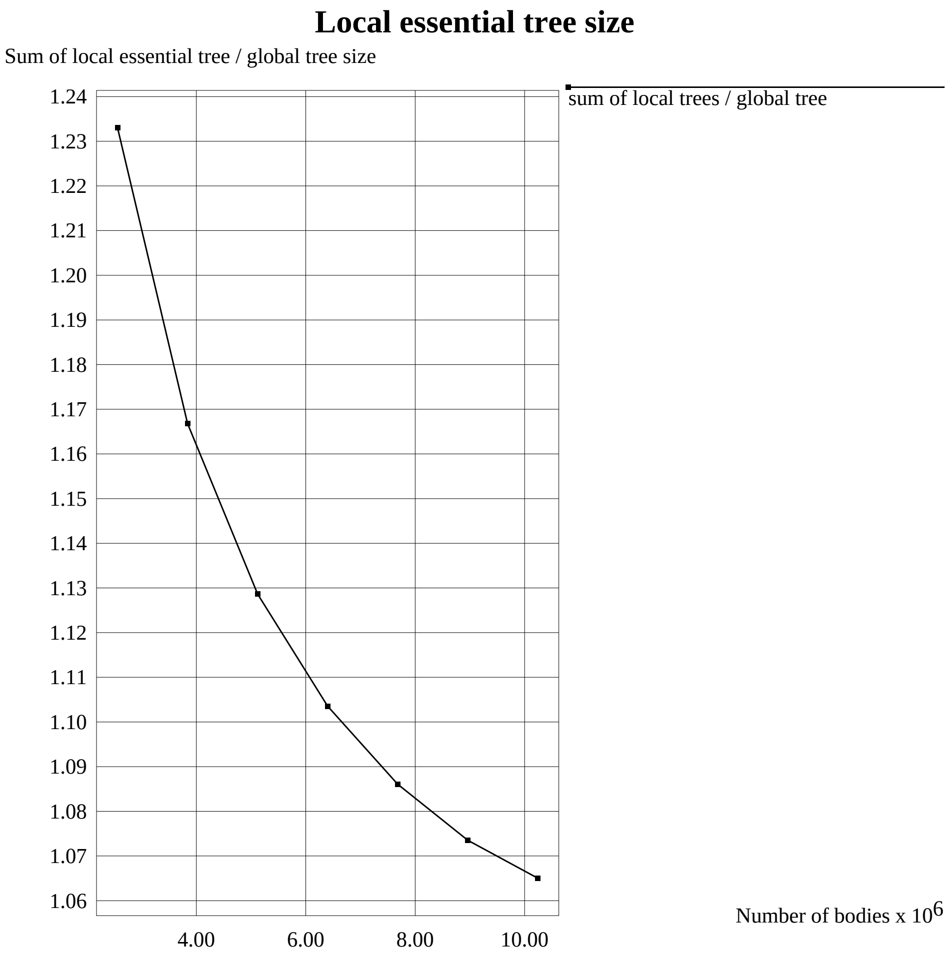


Figure 5.7: The ratio between the sum of the sizes of locally essential trees and the size of the global tree. Each body is counted as a tree node. The data is taken from a Plummer model simulation with up to 10 million bodies on a 256 node CM-5.

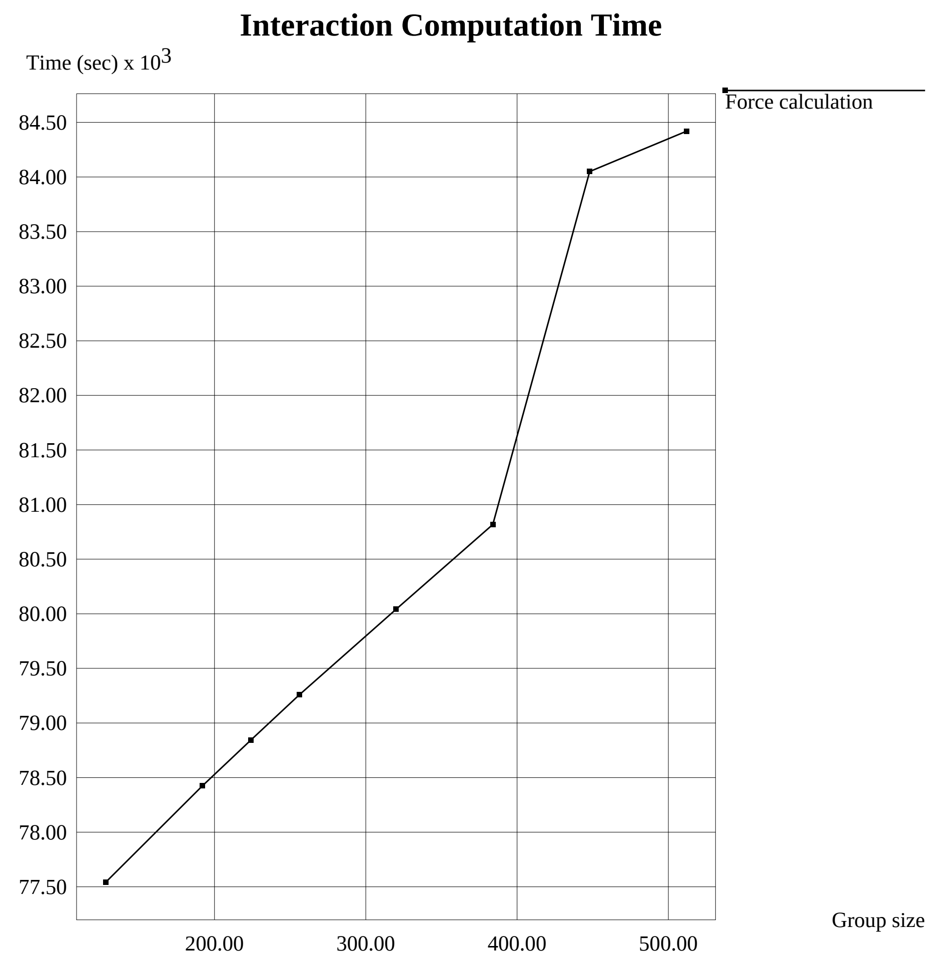


Figure 5.8: The time for vector units to compute interactions for different group sizes. The simulation ran on a 256 CM-5 with a Plummer model with 10 million bodies as the input distribution.

vector units process essential data in blocks of 16, so a small increase in G may not affect the total time for vector units to compute interactions.

Figure 5.9 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 modifications decreases as more bodies are processed at a time, so the time for cache modification decreases 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.

From the experimental results we conclude that the effect of reducing the number of cache modifications is more significant than the increased cost per cache modification, and the time for cache modification decreases as G increases.

Figure 5.10 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.

Figure 5.11 shows that the cache hit rate decreases as more bodies are processed as a group. When the group size G increases, the average size of the bounding box of a group also increases. As a result the cache hit rate decreases since the average distance between consecutive groups becomes larger.

5.5 Workload Balancing

Singh [44] reported that rebuilding ORB for each iteration degrades overall performance, and alternative partitioning method should be used. However, our dynamic load balancing adjusts ORB bisectors with negligible cost. From the experimental data, we conclude that *adjusting*, not rebuilding, ORB bisectors can balance workload efficiently. The imbalance is kept under 5% throughout the simulation with negligible remapping costs.

In order to reduce the number of broken BH-nodes, the remapping places the new bisectors on BH-node boundary as often as possible. The binary search for finding the new bisector location always start from the boundary points of a complete BH-node. This precaution reduces the number of broken leaves that require level adjustment, and the broken internal tree nodes that require representatives. Consequently the cost in both stages is reduced without any extra programming complexity.

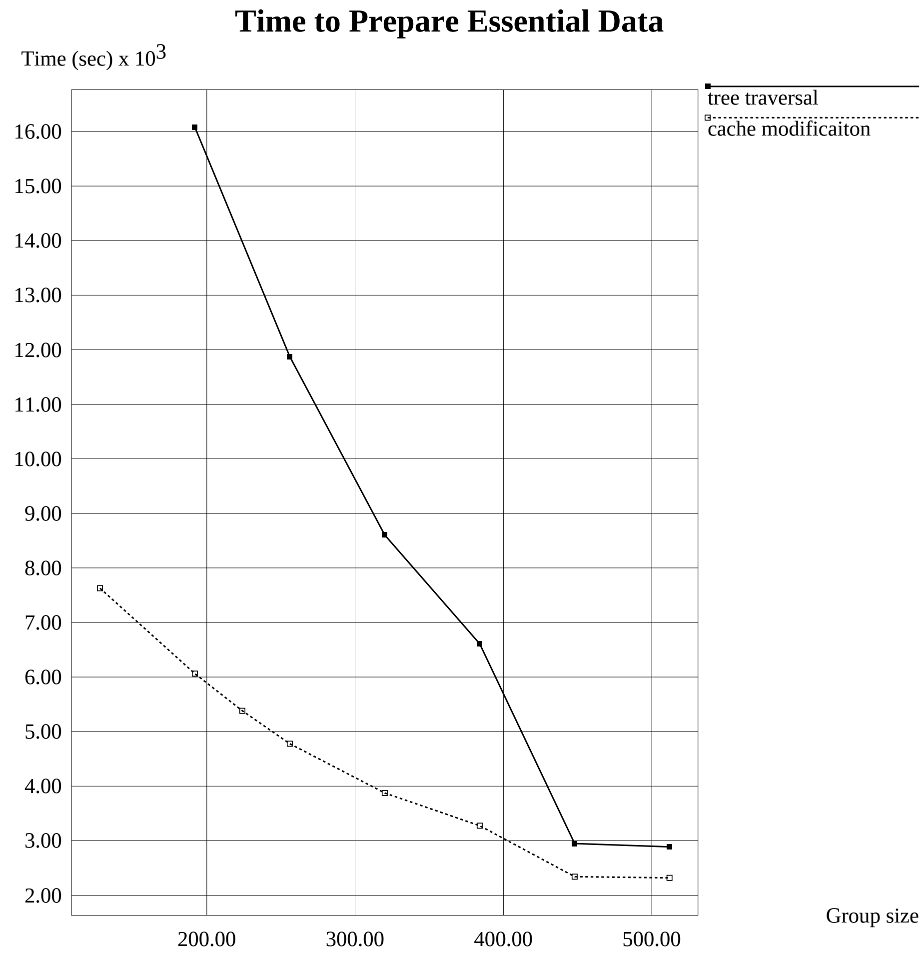


Figure 5.9: The time for caching (tree traversal) to collect essential data for different group sizes, with the same input distribution as in Figure 5.8

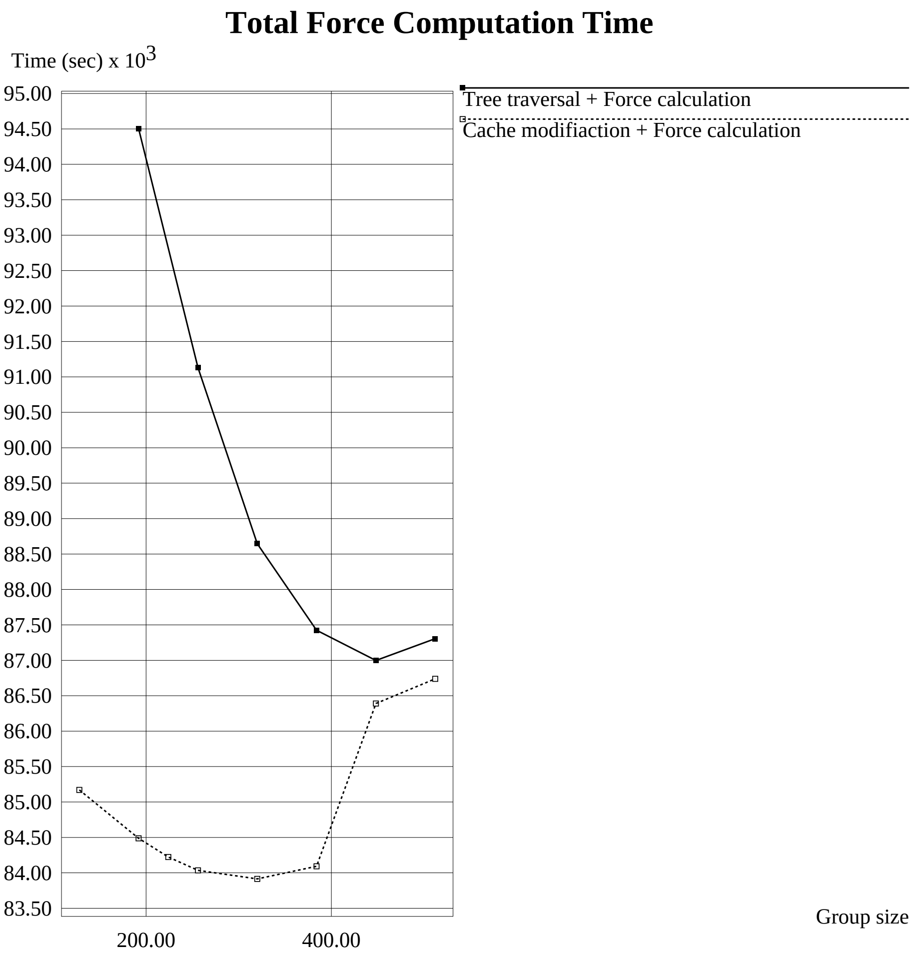


Figure 5.10: The total force computation time which is the sum of the time for vector units to compute interactions (Figure 5.8) and the time for caching (or tree traversal) to collect essential nodes for each group (Figure 5.9).

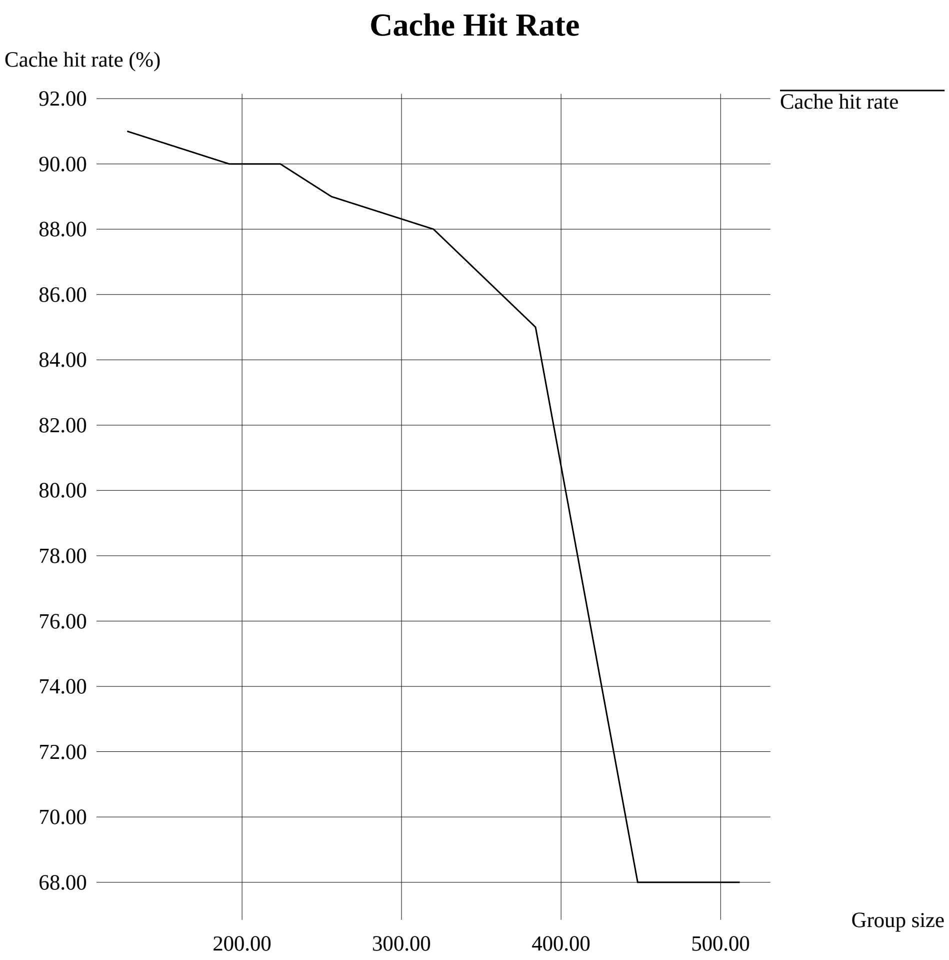


Figure 5.11: The cache hit rates under different values of group size. The accelerations of bodies are evaluated in the order of Peano-Hilbert space-filling curve. The data is taken from a Plummer model simulation with 10 million bodies.

5.6 Summary

Our experimental results show that N-body simulation can be implemented efficiently in distributed memory using explicit message-passing communication. The slightly more complicated programming requirement does not translate into performance penalty. Our message-passing implementation of Barnes-Hut's algorithm sustains a high speed of 36 Mflop/s with very little overhead in distributed data structure management and communication.

We show that ORB can efficiently preserve space locality and balance workload at the same time. The complication in BH-tree building can be solved by level adjustments and representative construction with very small overheads. The ORB bisectors can be adjusted to balance workload continuously with negligible remapping cost.

Barnes's technique [6] of grouping bodies together for tree traversal is extremely effective. The extra computation due to unnecessary opening of internal nodes is much smaller than the time saved in preparing essential data (by either tree traversals or caching). From our experiments, the performance improves up to the point that the group size reaches 320.

The essential data caching provides better performance than tree traversal for all group sizes up to 512. The space filling curve method for sequencing groups of bodies gives provably good performance of cache modification, and requires almost no extra costs for its implementation.

Finally, the principle of incremental update is very important in implementing N-body simulations. Our implementation adjusts instead of rebuilds all the data structures so that they can adapt to the dynamic movements of bodies. The BH-tree is incrementally updated to reflect the new positions of bodies. The ORB tree is incrementally updated in the remapping to reflect the new distribution of workload. And the essential data cache is incrementally updated to reflect the change in position of the current group of bodies during force calculation. We conclude from the experimental results that incremental update does provide better performance than rebuilding the data structures.

Chapter 6

Atomic Message Model

Each communication phase in the N-body simulation can be abstracted as the "all-to-some" communication. 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. The goal is to send all messages to their destinations in minimum time. Studying the "all-to-some" problem in distributed memory necessitates further understanding of message-passing systems, and a communication model that captures the important factors of efficient communication.

The message-passing style of programming is widely used on almost all parallel computers. The primitives to *send* and *receive* messages hide low-level architectural details and are ideal for programming many large applications. While message-passing systems have been in use for over a decade, relatively few results concerning the complexity of message-passing protocols are available. One reason for this discrepancy is the lack of theoretical models that appropriately capture issues related to communication; as stated in [14], most theoretical models "encourage exploitation of formal loopholes, rather than rewarding development of techniques that yield performance across a range of current and future parallel machines."

We propose an atomic model [33] to study the performance of message-passing programs. The model is simple and much more restricted in its capabilities in comparison with existing systems. Nevertheless, we show that it allows simple and efficient solutions (linear speed-up) for message scattering, backtrack and branch-and-bound searches.

6.1 Message-passing Systems

Message-passing instructions appear in two varieties: *blocking* and *non-blocking*. Blocking instructions require synchronization between the sender and receiver: a send instruction terminates only when the corresponding receive is executed by a remote process. One advantage of blocking instructions is that no system buffering is required. However, the delay in waiting for a send instruction to complete means that computation and communication cannot overlap; this can reduce overall performance significantly. Another disadvantage is that the programmer must carefully arrange send/receive instruction pairs to avoid deadlock.

Non-blocking instructions allow a process to execute multiple send instructions before any of the corresponding receive instructions is executed. This allows for the possibility of increased efficiency since communication and computation can overlap. However, more system resources, buffering and bandwidth for example, are required for a non-blocking scheme otherwise *pending* messages (those sent but not yet received) will be excessively delayed or potentially lost. Moreover, since system resources are finite, the programmer must ensure that the number of pending messages is bounded at all times.

Underneath the message-passing abstractions, a message goes through several phases before it is absorbed at its destination. During each phase it requires some critical system resource to continue its journey. For example, a memory buffer is required to compose a message. When a message buffer is sent, it goes through the network interface connecting the processor to the network. Before the message arrives at the destination, it travels in the network and occupies network buffers. On reaching its destination the message occupies a buffer at the network interface before it is removed and processed. Whenever a message cannot get the critical resource it needs, it must wait. When messages wait a long time, there is the danger that communication delays can cause processor idling, thereby reducing overall performance greatly.

In many applications it is also common practice to reduce communication costs (due primarily to system overheads) by aggregating data into fewer but longer, atomic, messages [11].[1] First the sender notifies the receiver of the message length. Upon receipt of this notification, the receiver allocates sufficient buffer space and sends back an acknowledgment. This establishes a link between the sender and the receiver and the message is transmitted in the third step. Once again, there is ample opportunity for delay from the time the protocol is initiated to the time the data is actually transferred.

Given the limited resources of multicomputer systems, it is natural to ask whether the efficiency gained by using non-blocking instructions is lost if the number of pending messages is severely limited.

We investigate this question formally within the atomic model which permits only one pending message per processor. In brief, each processor is given one send buffer and one receive buffer, each capable of holding one atomic message. The system alternates between message transmission and computation cycles. During a computation cycle a processor retrieves a message from its receive buffer, performs a computation, enques newly generated messages into a message queue, and writes the first message in the queue into the send buffer if the send buffer is empty. During the transmission cycle, the network attempts to transmit every message in each send buffer to the receive buffer of the destination. If more than one message is destined for the same processor, exactly one is successfully transmitted. The rest remain in their send buffers. The one which is transmitted is chosen by a *network arbiter*. The *worst-case* arbiter makes choices to maximize the running time. The *FIFO* arbiter gives priority to messages with smaller time-stamp; messages with the same time-stamp can be delivered in arbitrary order.

[1] Atomic messages travel through a critical resource as a single entity; different messages do not co-exist inside the critical section.

The atomic model is motivated by the desire to analyze the performance of message-passing programs in an architecture-independent manner. For this reason, we have chosen to abstract the network as an arbiter which takes one unit of time to transfer messages from send buffers to receive buffers at the destination. We believe this is reasonable in applications that involve the atomic transfer of large data sets. Unit-delay assumptions are also made in the literature on PRAMs and complete networks [28, 29]. Unlike these models however, we explicitly account for message contention and do not allow multiple messages to be received in one step by a processor. The issue of contention at the receiving module is also addressed in models for optical communication [21] and module parallel computers [27, 35]. A key feature which distinguishes the atomic model is that once a message has been sent it cannot be retrieved; the sending processor must wait for the network to clear the send buffer after the message has been copied into the receive buffer at the destination. Finally, the atomic model can be viewed as the limiting version of the LogP model [14]; with long messages of equal length the latency, overhead and gap parameters of the LogP model can be lumped into a single, unit time delay.

Inspite of the restriction on the rate at which the network can deliver messages to a destination, as well as the adversarial nature of the arbiter, we show that simple randomized algorithms can attain linear speed-up for branch-and-bound and backtrack tree searching. And all-to-some message passing can finish within constant factor of optimal time with high probability if the destinations are uniformly distributed among processors.

6.2 The Atomic Message-passing Model

We model a message-passing multicomputer as a collection of p nodes connected via an interconnection network [43]. For convenience of analysis we require that the system be synchronous, and operate in discrete time steps.[2] Each time step is divided between one communication step and one node computation step.

Each node consists of a receive buffer, a processor, local memory, a queue manager, a message queue, and a send buffer. Each buffer can hold one atomic message. Every node can perform local computation using its processor and local memory. It can also receive a message using the receive buffer and enque messages into the message queue. The message queue is maintained by a queue manager which may be under the control of the processor or the system. A message from the message queue is injected into the network by placing it into the send buffer. For our purposes, it is convenient to model the actions at a node as repeated executions of the following *reactive cycle* which occurs during one synchronous time step:

1. **The send phase** (performed by the processor or system):
   - MAINTAIN QUEUE: Put newly enqueued messages into an appropriate place in the message queue.

[2] This assumption is not required for termination, but simplifies the analysis of throughput.

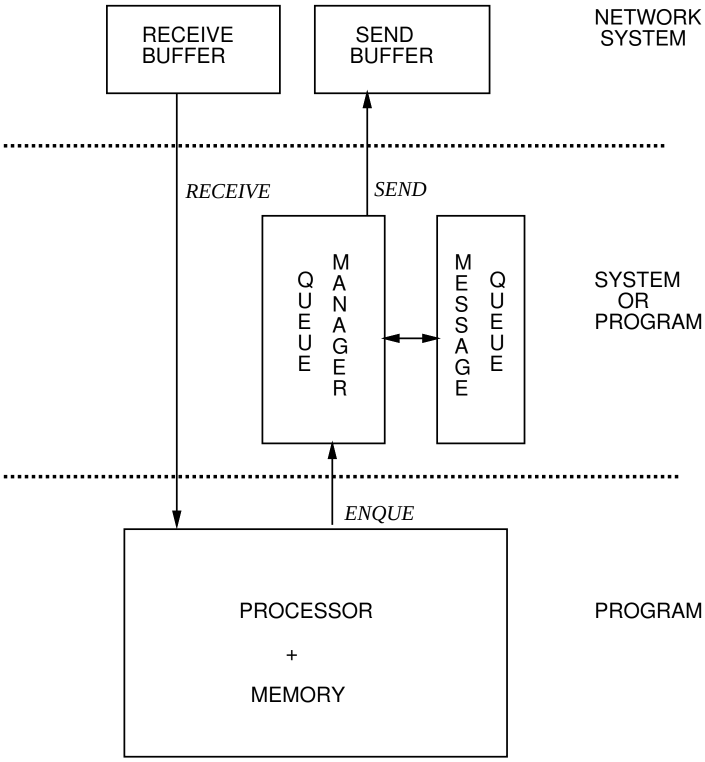


Figure 6.1: The structure of a node.

- SEND: Inject the message at the head of the queue into the send buffer if it is empty.

2. **The transmission phase** (performed by the network system):

   - TRANSMIT: Take messages from send buffers to receive buffers according to message destinations. If more than one message is destined for the same receive buffer, the one which succeeds is selected by the network arbitration policy.

3. **The computation phase** (performed by the processor):

   - RECEIVE: Probe the receive buffer to receive an incoming message, if any, into local memory.
   - COMPUTE: Perform local computation, possibly on the newly received message, and generate new messages.
   - ENQUE: Pass the newly generated messages to the queue manager.

Observe that there are two ways a message can be delayed. First, a message may have to wait in the message queue until it is selected to be placed in the send buffer. Second, once a message is in the send buffer, it may be delayed in the network. We call the second kind a *receive delay.*

When more than one message, occupying send buffers of different nodes, are simultaneously destined for the same node, the network must deliver one message. Since every node executes a RECEIVE instruction during its reactive cycle, this requirement of the network satisfies the network contract of the CM-5 [32]: "The data network promises to eventually accept and deliver all messages injected into the network by the processors as long as the processors promise to eventually eject all messages from the network when they are delivered to the processors." With the reactive cycle and the network contract we are assured that deadlocks cannot occur.

We wish to make as few assumptions as necessary on the message queue. Our results for backtrack search are independent of queue maintenance. Our result for branch-and-bound depends on maintaining the message queue as a priority queue.

We also wish to make as few assumptions as necessary about the network arbitration policy when multiple messages are destined for the same node. We will consider two different network arbitration policies. The *worst-case* policy selects the message which maximizes the overall time to complete the task at hand. The *FIFO* policy dictates that, for any pair of messages with the same destination, they will be accepted in the order of earliest occupancy of their respective send buffers. In other words, if the messages reach their send buffers at different time steps, then the earlier one will be delivered first. If two messages reach their send buffers at the same time, then the order of delivery is arbitrary. Optimal speedup for backtrack search can be achieved even with worst-case arbitration; whereas, we require FIFO arbitration to prove optimal speedup for branch and bound search.

6.3 Overview of Results

We will study three problems under the atomic message setting: all-to-some message scattering, backtrack search, and branch-and-bound search. For each of these problems we analyze the case when all messages are destined for independently chosen random nodes. Our intuition is that when messages are headed for random destinations, the number of conflicting messages is unlikely to become too large. However, when the size of the computation is much larger than the number of processors, this is not always true and one has to prove that the effects of the conflicts do not add up significantly.

The message scattering problem is informally stated as follows: suppose that each node has a list of m messages to send (in order) to remote nodes. How much time does it take, under the worst-case (adversarial) arbitration policy, until all messages are received at their destinations? This problem arises naturally in several applications. In fact, the message scattering problem and the atomic message model are motivated by the "all-to-some" communication in our parallel N-body implementation (Section 3.5).

In the backtrack search problem, each internal vertex of a search tree $\mathcal{T}$ corresponds to a partial solution to a problem while each leaf represents a solution with a certain cost. The goal of backtrack search is to find the minimum cost leaf in the search tree. The search tree is not given in advance, rather it is spawned on-line as the search proceeds. The search begins with the root of the tree in a given node; when each internal vertex is expanded two (or any bounded number of children) are spawned and must each be examined. When a leaf is examined, the cost is calculated and no further expansion along that branch is possible. If the total number of vertices in the search tree is n, and the maximum depth of any leaf is h, it is easy to see that the time to examine all leaves is at least $\Omega(n/p + h)$, where p denotes the number of processors.

Branch-and-bound search is similar to backtrack search, except that only a subtree of the search tree must necessarily be explored. Following Karp and Zhang [28, 29], we model a branch-and-bound tree as a binary search tree, each of whose vertices has an associated cost. The cost of each vertex is strictly less than the cost of each of its children (for simplicity we assume that all vertex costs are distinct). The problem is to find the leaf with minimum cost in the tree. Clearly, every tree vertex whose cost is less than the minimum cost leaf must be expanded because one of its children could potentially be the minimum cost leaf. These vertices form a critical subtree, call it $\mathcal{T}$ of the overall search tree.

As before, the time to complete the search is $\Omega(n/p + h)$ where n is the number of vertices in the *critical* subtree, and h is the height of the critical subtree. Non-critical vertices can, in principle, be pruned by the search process and need not be explored.

Tight upper bounds for branch-and-bound, and hence for backtrack search, were given by Karp and Zhang [28] on the complete network which allows multiple messages to be simultaneously received at each node, and on the concurrent PRAM which essentially allows unsuccessful writes to be detected. The basic idea was to send each node to a random processor for further exploration. Ranade [39] gave an elegant alternative proof of the Karp-Zhang result. By extending Ranade's techniques we show that the random destination strategy yields linear speedup for backtrack search in the atomic model.

Theorem 4 *Using random destinations, the probability that a binary backtrack search tree of size n and depth h takes time more than $k(n/p+h)$ in the atomic transmission model with worst-case arbiter is polynomially small in n, for k sufficiently large.*

Achieving linear speedup for branch-and-bound in the atomic model is a little harder. The subtle distinction is that pending non-critical vertices can delay pending critical vertices. In the Karp-Zhang model this can never happen. Since we have no control over the number of non-critical vertices, and we do not know the shape of the critical subtree, it is conceivable that the delays can become arbitrarily large under the worst-case arbiter which consistently favors non-critical vertices over critical vertices. However, under a FIFO arbiter we establish the following result.

Theorem 5 *Let the critical subtree, $\mathcal{T}$ of a branch-and-bound search tree have size n and depth h. Using randomized destinations, the probability that the time, in the atomic model with FIFO arbiter, exceeds $k(n/p+h)$ is polynomially small in n when $n > p^2 \log p$, and k is sufficiently large.*

We present the proof of Theorem 4 and 5 in Chapter 7.

6.4 Message Scattering

The off-line version of the message scattering problem in which the lists can be reordered is easily solved using standard bipartite graph edge-coloring techniques [9, 20]. If r is the maximum number of messages received by any node, then $\max\{r, m\}$ steps are necessary and sufficient.

However, the distributed version of the problem, without reordering, is not as simple. We show that with each of p nodes sending m messages (m can be arbitrarily larger than p) the worst-case time is $\Omega(mp)$. In other words, the average throughput of the system is $O(1)$ messages received per time step, independent of the size of the system.

On a positive note, we show that when each of the messages is destined for a randomly chosen node (all destinations independent and uniformly drawn) then, with high probability, the time to completion is $O(m)$. This means that the average throughput is $\Omega(p)$ messages received per time step, asymptotically the maximum possible.

6.4.1 Lower Bounds

Suppose every processor sends n messages to every processor in ascending processor index order. We show that a simple FIFO network arbiter increases the communication time to $\Omega(n^2p)$ so that in average only a constant number of messages are received in one time step. The network arbiter ensures that the messages are received in FIFO order; the message sent first is received first. The messages sent at the same time are received in increasing processor index order.

Figure 6.2 shows the history of four processors sending two messages to each destination in ascending processor index order. A square in the intersection of row i and column j indicates that processor p_i successfully sends a message at time step j. The numbers in the squares are the processor index of the destination. Notice that two successful sends to the same destination are p time steps apart because the messages are received in FIFO order. The total number of time steps is therefore $((n-1)p+1)p+(p-1) = \Omega(np^2)$.

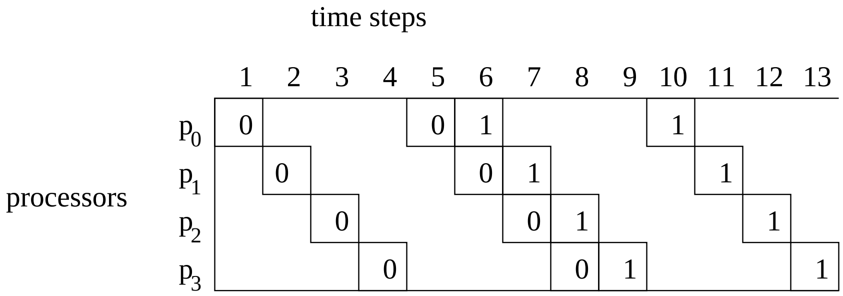


Figure 6.2: The situation when the messages are sent in ascending processor address order.

6.4.2 Randomized Scattering

Formally, we establish the following theorem.

Theorem 6 *Suppose that each node sends m messages, and that for each message all destinations are equally likely and independent of choices of all other messages. The probability that the time until all messages have been received exceeds km is bounded by $O(e^{-m})$, for sufficiently large k and $m > \log p$.*

Proof. We adapt Ranade's proof [39] of the result of Karp and Zhang [28].

Let T be the completion time of the protocol, the last time step at which a message is received. Let message $\mathcal{M}_m$ be a message received at time step T, and let S be the node which was the source of message $\mathcal{M}_m$. Let $\mathcal{M}_i$ denote the ith message sent by node S, and let T_i denote the time step at which $\mathcal{M}_i$ was received at its destination Q_i.

Definition. Suppose that message m is selected for transmission, i.e., m enters the send buffer at time step τ, and is destined for node q. Then we say that m became *ready for q at time step τ*.

Lemma 2 *There exists a partition $\Pi = \Pi_1, \ldots, \Pi_m$ of the interval $[1, T]$ and a set R of $T-m$ messages (not including those sent by S) each of which satisfies the following property: if the message became ready during Π_i its destination node is Q_i.*

Proof of Lemma. Message $\mathcal{M}_i$ is received at Q_i at time step T_i. Let $T_i^{nr} < T_i$ be the maximum time step at which Q_i does not receive a message. At each time step of the interval $\Delta_i = [T_i^{nr}+1, T_i]$ Q_i receives a message. Each of these messages became ready during the same interval Δ_i.

Observe that message $\mathcal{M}_{i-1}$ was received at time step T_{i-1}, and message $\mathcal{M}_i$ became ready at time step $T_{i-1}+1$. Therefore, $T_i^{nr} \leq T_{i-1}$. This means that there is no gap between any pair of consecutive intervals Δ_i, Δ_{i+1}. Given the intervals $\Delta_1, \ldots, \Delta_m$, we construct a partition Π as follows:

$$\begin{aligned}\Pi_m &= \Delta_m \\ \Pi_i &= \Delta_i - \bigcup_{j>i} \Pi_j, \quad 1 \leq i < m.\end{aligned}$$

By construction, it follows that every message received by Q_i during Π_i became ready during Π_i, and at least $T-m$ messages received by Q_i's during Π_i's were not sent by S. This establishes the lemma. ∎

To complete the proof of the theorem, we sum, over all possible partitions, choice of source S, and choice of $T-m$ messages, the probability that these $T-m$ messages chose their destinations in accordance with the partition.

The probability that a message which becomes ready during Π_i chooses Q_i as its destination equals $\frac{1}{p}$. The probability that each of $T-m$ messages makes the right choice is $p^{-(T-m)}$. The number of choices for S, the partitions and the $T-m$ messages equals $p\binom{T+m}{m}\binom{(p-1)m}{T-m}$. The probability that $T \geq km$ is at most

$$\begin{aligned}& p\binom{T+m}{m}\binom{(p-1)m}{T-m}p^{-(T-m)} \\ \leq\ & 2^{T+2m}\binom{(p-1)m}{(k-1)m}p^{-((k-1)m)} \\ \leq\ & 2^{(k+2)m}(\frac{e}{k-1})^{(k-1)m} \\ \leq\ & (2^{k+2}(\frac{e}{k-1})^{k-1})^m.\end{aligned}$$

For k sufficiently large, this quantity is smaller than $O(e^{-m})$. ∎

Chapter 7

Efficient Tree Search

This chapter presents the randomized reactive protocol for backtrack and branch-and-bound tree search. We prove that the reactive protocols yield linear speedup in the atomic message model.

7.1 Techniques for Tree Searches

7.1.1 Algorithmic Issues

This section outlines the algorithmic and proof strategies for backtrack and branch-and-bound search in the atomic message model. The branch-and-bound strategy is essentially that of Karp and Zhang [28]; their model allows any number of messages to be received at a node in one time step. Our technical contribution is to extend their result to the weaker atomic transmission model. The proofs of both results extend the techniques of the previous section.

While the goal of both search procedures is to find the minimum-cost leaf, there is an essential difference. Backtrack search examines every vertex of the search tree. In branch-and-bound search the cost associated with each vertex increases monotonically with the distance from the root, so that only the *critical subtree*, consisting of vertices with cost no greater than the minimum-cost leaf, need be examined. We call such vertices *critical vertices*. For efficient branch-and-bound search, the time devoted to examining non-critical vertices must not dominate that for examining the critical subtree.

Within each synchronous reactive cycle, each processor: (1) receives a tree vertex, if any, from its receive buffer, (2) examines and expands the vertex, and (3) puts the children onto the message queue, headed for an independently chosen random destination. For backtrack search we place no requirements on the message queue discipline. However, for branch-and-bound search we require that the message queue be a priority queue, so that the tree vertex selected for transmission is the one with minimum cost.

Using priority message queues for branch-and-bound search means that non-critical vertices cannot be selected for transmission when there is at least one critical vertex inside the message queue. However, a critical vertex can arrive inside the message queue while a non-critical vertex occupies the send buffer. In this case, the critical vertex will have to wait for selection, but it is easy to see that a critical vertex can be delayed by a non-critical vertex in this manner at most once.

Once a message has been selected for transmission, it is still subject to receive delays. Receive delays depend on the network policy and are beyond the control of the programmer, so we would like to make as few assumptions as necessary. For backtrack search we are able to carry out the analysis without making any assumptions on network arbitration. For branch-and-bound however, our analysis requires that the network observes a FIFO arbitration policy.

In conclusion, our analysis for branch-and-bound search makes stronger assumptions on both the message queue discipline, and the network arbitration policy. The first assumption is required to guarantee that progress is made on the critical subtree and is reasonable from an algorithmic viewpoint. The second assumption, concerning network arbitration, is required for technical reasons: we bound the running time as a function of the size of the critical subtree, not the entire search tree which can be arbitrarily larger. Currently we do not know if the FIFO assumption can be weakened, and it is conceivable that it can.

7.1.2 Proof Techniques

In this section we describe some of the ideas and terminology common to the analysis for both backtrack and branch-and-bound search. In both problems our goal is to analyze the time to expand a critical tree[1] of size n and depth h on a p-node system. For branch-and-bound search the quantities n and h can each be much smaller than the size and depth of the complete search tree.

In the analysis of the running time we proceed as follows. At time $t = 1$ the root is assumed to be in the send buffer of some node, and is received at its destination within that time step. Suppose that the running time is T, i.e., the last time step at which a critical vertex is received. Pick one of the critical vertices received at time T – it must be a leaf in the critical subtree. Call the path $s_1, s_2, \ldots, s_h$ from the root (s_1) to this critical leaf (s_h) the *special* path and the vertices along this path the *special vertices*. Let Q_i denote the destination queue of special vertex s_i.

The first step of the proof is similar to the proof of Lemma 2. For a fixed run of the algorithm we construct a partition, $\Pi = \{\Pi_1, \ldots, \Pi_h\}$, of the time interval $[1, T]$. Next, we construct a *signature set* R of non-special, critical vertices each of which became ready for some Q_i during the corresponding time interval Π_i. Roughly speaking, the signature set, R, is constructed such that the receive delay periods of its children are disjoint, and the sum of these receive delays is large, i.e., close to T.

There are two cases to consider: the signature set R is either large or small, compared with the threshold αT, where α is some suitably chosen constant. We first show that it is unlikely that R is large when T is large. The proof closely follows Lemma 2.

[1] Every vertex in backtrack search is critical.

Lemma 3 *For suitable constants k, α, the probability that $T > k(\frac{n}{p} + h)$ and $|R| \geq \alpha T$ is polynomially small in n.*

Proof.

We estimate $Pr(|R| \geq \alpha T)$ by summing the probability of the event $|R| \geq \alpha T$ under all possible combinations of partition Π and queue sequence Q. Given Π and Q each critical vertex appears in R with probability $1/p$, independent of other vertices. The probability that $|R| \geq \alpha T$ is therefore bounded by $\binom{n}{\alpha T} p^{-(h+\alpha T)}$.

The number of choices for Π, Q and the special path S is no more than $\binom{T+h}{h} p^h n$. Thus, the probability that $|R| \geq \alpha T$ is bounded by

$$
\begin{aligned}
& \binom{n}{\alpha T} p^{-\alpha T} \binom{T+h}{h} n \\
< \; & (\frac{ne}{\alpha T p})^{\alpha T} ((1 + T/h)e)^h n \\
< \; & (\frac{e}{\alpha k})^{\alpha k(\frac{n}{p}+h)} ((k(\frac{n}{ph} + 1) + 1)e)^h n, \quad \text{when } T > k(\tfrac{n}{p} + h) \\
< \; & [(\frac{e}{\alpha k})^{\alpha k(\frac{n}{ph}+1)} ((k(\frac{n}{ph} + 1) + 1)e)]^h n \\
< \; & [(\frac{e}{\alpha k})^{\alpha k} ((2k + 1)e)]^h n
\end{aligned}
$$

which, for appropriately chosen k, is polynomially small in n, the size of the critical subtree.
■

The second part of the proof argues that it is unlikely that R is small when T is large. The intuition is that the expected receive-delay of any vertex is a small constant; therefore, it would seem unlikely for the children of a small number of signature vertices to suffer a large total receive-delay. Unfortunately, the delays of the children of the signature vertices are not independent random variables, so that Chernoff bounds cannot be immediately invoked.

Briefly, in analyzing backtrack search we track the destinations of the children of the signature vertices to construct a new set of queues, a new partition of time, and a new signature set. The new signature set is guaranteed to be large; consequently, the remainder of the proof follows the proof of Lemma 3. The analysis of branch-and-bound is based on the observation that, under FIFO arbitration, the delays of the children of the signature set can essentially be treated as a martingale, thereby allowing us to use Chernoff bounds.

7.2 Analysis of Backtrack Search

In this section we demonstrate that when each vertex chooses its destination randomly and independently, then with high probability the completion time of backtrack search is optimal within a constant factor.

Following the outline of the previous section, we proceed in two stages. In the first stage we identify the required signature set R; the second stage establishes the unlikelihood of the event that T is large while R is small.

7.2.1 The Signature Set

7.2.2 Signature Set

We begin with some terminology and definitions. As before, let $S = \{s_1, \ldots, s_h\}$ denote the special vertices along the special root-to-leaf path and Q_i be the destination queue which receives special vertex s_i at time T_i.

Definition. A node Q is *empty at time* t if neither the send buffer nor the message queue of Q contains a vertex *at the end of time step* t.

By definition a node cannot be empty at time t if it receives an internal vertex at time t. However, it is possible that a node which is empty at t receives a leaf at time t.

Definition. For any time interval $I = [s, e]$, $I^+ = [s+1, e+1]$ is the interval obtained by shifting each end-point of I by one, and $I^{\rightarrow} = [s, e+1]$ by shifting only the right endpoint.

Note that any node which is non-empty throughout an interval I attempts to inject a vertex into the network at every time step of I^+.

Definition. Suppose that node Q receives a vertex at each time-step during time interval W. We call the interval W an *arrival window* for node Q.

Note that if time interval W is the receive delay of vertex v, then W is an arrival window for the destination queue of v.

Definition. For $1 \leq i < h$, let T_i^e denote the maximum time $t < T_{i+1}$ such that Q_i is empty at t, and T_i^{nr} denote the maximum time $t \leq T_i^e$ at which Q_i does not receive a message. Finally, let N_i denote the interval $[1 + T_i^e, T_{i+1} - 1]$, $A_i = [1 + T_i^{nr}, 1 + T_i^e]$, and $\Delta_i = A_i \cup N_i = [1 + T_i^{nr}, T_{i+1} - 1]$ (Figure 7.1).

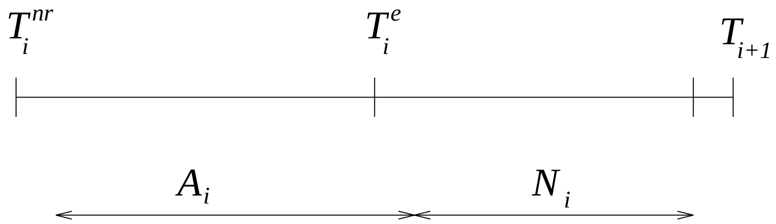


Figure 7.1: The interval Δ_i

The following lemma summarizes three properties which are a straightforward consequence of the definitions above.

Lemma 4

1. *Q_i receives an internal vertex at time $1 + T_i^e$,*

2. *Q_i is non-empty throughout N_i and Q_i attempts to inject a vertex at every step of N_i^+, and*

3. *A_i is an arrival window for Q_i.*

Let c_i denote the set of vertices that are injected into the network from Q_i during N_i^+. We obtain the following lemma.

Lemma 5

1. *The parent of every vertex in c_i becomes ready for Q_i during Δ_i,*

2. *$\Delta_i^{\rightarrow}$ can be partitioned into a set of arrival windows, and*

3. *the interval Δ_i contains $[T_i, T_{i+1})$, and $\bigcup_{i=1}^{h-1} \Delta_i = [1, T-1]$.*

Proof.

Let w be the parent of a vertex $v \in c_i$. In contradiction to (1), suppose w is ready before T_i^{nr}. Two cases follow: either w is received before T_i^{nr} or after T_i^{nr}.

In the first case, if w is received before T_i^{nr}, then v will stay in the message queue of Q_i until it becomes ready. However, from the definition, v cannot become ready until $T_i^e + 1$ or later. This contradicts the fact that Q_i is empty at T_i^e.

In the second case, if w is received after T_i^{nr}, then the receive delay of w is an arrival window for Q_i. This contradicts the fact that Q_i does not receive a vertex at T_i^{nr}. As a result w must be ready after T_i^{nr}.

From part 2 of Lemma 4 N_i^+ can be partitioned into arrival windows for the destinations of c_i. Therefore, $\Delta_i^{\rightarrow}$ can be partitioned into arrival window A_i for Q_i and a set of arrival windows in N_i^+ for the destinations of c_i.

Finally, for part 3 we observe that $s_{i+1} \in c_i$, so s_i must be ready after T_i^{nr} (from part 1 of this lemma) and is received at $T_i > T_i^{nr}$. Therefore Δ_i contains $[T_i, T_{i+1})$ and part 3 follows. ■

From part 3 of Lemma 5 the union of all Δ_i covers the time interval $[1, T-1]$, consequently we can define a partition $\Pi = \{\Pi_1, \ldots, \Pi_{h-1}\}$ of the interval $[1, T-1]$ as follows.

$$\begin{aligned} \Pi_{h-1} &= \Delta_{h-1} \\ \Pi_i &= \Delta_i - \bigcup_{j>i} \Pi_j, \quad 1 \leq i < h-1. \end{aligned}$$

Definition. Let R_i be the set of critical vertices which are not special ($v \notin S$) but are ready for Q_i during the interval Π_i. Also, let $R = \cup_{i<h} R_i$. We call the set R the *signature set.*

Having identified the signature set, it remains to estimate the probability that the signature set is small when T is large.[2] This estimation is completed in the following section.

7.2.3 A Refined Partition

As mentioned in Section 7.1, our strategy will be to find a new partition of the interval $[1, T]$ and a corresponding signature set which is guaranteed to be large.

In this section we identify $O(|R| + h)$ arrival windows which cover the interval $[1, T]$. We will find arrival windows to cover each $\Pi_i^{\rightarrow}$ $(1 \leq i < h)$ and argue that the sum of number of windows in each $\Pi_i^{\rightarrow}$ is $O(|R| + h)$. The next section will identify the new partition and signature set.

Definitions. Let C_i denote those k_i children of $R_i \cup s_i$ that are ready within $N_i^+ \cap \Pi_i^{\rightarrow}$. We sort C_i into a list $v_{i1}, \ldots, v_{ik_i}$ according to the time they become ready. Also let Q_{ij} denote the destination of vertex v_{ij} and W_{ij} denote the receive delay of v_{ij}.

Each $\Pi_i^{\rightarrow}$ is the union of two disjoint intervals $N_i^+ \cap \Pi_i^{\rightarrow}$ and $A_i \cap \Pi_i^{\rightarrow}$. The interval $N_i^+ \cap \Pi_i^{\rightarrow}$ is an initial segment of N_i^+ which, from part 2 of Lemma 5, can be partitioned into arrival windows. Each such arrival window W is the receive delay of a vertex v which is enqueued after T_i^e and becomes ready in Q_i at the beginning of W. From part 1 of Lemma 5 the parent of v must be ready for Q_i after T_i^{nr}. In other words, the parent of v must be ready during Π_i and consequently $v \in C_i$. As a result the interval $N_i^+ \cap \Pi_i^{\rightarrow}$ can be partitioned into the receive delays of the k_i vertices in C_i[3]. Each such receive delay is an arrival window W_{ij} for Q_{ij} $(1 \leq j \leq k_i)$.

From part 3 of Lemma 4 $A_i \cap \Pi_i^{\rightarrow}$ is an arrival window for Q_i. Let $W_{i0} = A_i \cap \Pi_i^{\rightarrow}$ and $Q_{i0} = Q_i$, so that the interval $\Pi_i^{\rightarrow}$ can be partitioned into arrival windows W_{ij} for Q_{ij} $(0 \leq j \leq k_i)$. Finally we consider $\Pi_i^{\rightarrow}$ for all i and it follows that $\bigcup_{1 \leq i < h} \bigcup_{0 \leq j \leq k_i} W_{ij} = [1, T]$.

From the discussion above the receive delays, W_{ij}, of the k_i vertices $v_{ij} \in C_i$, and W_{i0} cover $\Pi_i^{\rightarrow}$. Moreover the parent of every vertex in C is in either R or S; consequently, the number of arrival windows is at most $h + |C| \leq h + 2(|S| + |R|) = 3h + 2|R|$.

We need one more definition to derive the refined partition Π^* and queue sequence Q^* where we can find T vertices that become ready for Q^* according to Π^*.

Definition. For the arrival window $W_{ij} = [t_1, t_2]$ for Q_{ij}, let $t \leq t_1$ be the maximum time step at which Q_{ij} does not receive a message. Define $W_{ij}^* = [t+1, t_2]$ so that (W_{ij}^*, Q_{ij}) is the *maximal backward extension* of the arrival window.

[2] The case when R and T are both large is covered by Lemma 3.

[3] The receive delay of the last vertex v_{ik_i} may not fall entirely within $N_i^+ \cap \Pi_i^{\rightarrow}$ in which case we truncate it accordingly

Let Q^* be the sequence of all the queues Q_{ij}, $1 \leq i < h$, $1 \leq j \leq k_i$. Notice that every message received by Q_{ij} during the interval $W^*_{ij} = [t+1, t_2]$ necessarily become ready for Q_{ij} during the same interval. Otherwise Q_{ij} would have received a message at time t, a contradiction.

From the extended windows W^*_{ij}, $1 \leq i < h$, $0 \leq j \leq k_i$, we next obtain the partition Π^* as follows:

$$\begin{aligned} \Pi^*_{h-1\ k_{h-1}} &= W^*_{h-1\ k_{h-1}} \\ \Pi^*_{ij} &= W^*_{ij} - \bigcup_{l>i \vee (l=i \wedge m>j)} \Pi^*_{lm} \end{aligned}$$

Now we can find T vertices that become ready for Q^* according to Π^*. Let X_{ij} denote the set of vertices v such that $v \notin C \cup R \cup S$, and v is received by Q_{ij} during Π^*_{ij}. From the discussion above every vertex in X_{ij} must become ready for Q_{ij} during Π^*_{ij}. Finally, let $X = \cup X_{ij}$ and $V = C \cup R \cup S$. Since the arrival windows cover the interval $[1, T]$, it follows that $|X| \geq T - |V|$.

7.2.4 Execution Templates

Our goal in this section is to estimate the probability of the event that T and R are both large. We proceed in two stages; first we characterize the completion time in terms of an *execution template*. Then we show that execution templates corresponding to large completion times are unlikely. This follows the delay-sequence arguments used in the literature [39, 40].

Definition. An execution template $\mathcal{E}$ is an octuple $(S, R, C, \Pi, \Pi^*, X, Q, Q^*)$ whose elements are defined as follows.

- $S = \{s_1, \ldots, s_h\}$ denotes the set of vertices along a path from the root to a leaf,
- R_i, $1 \leq i < h$, are disjoint sets of non-special critical vertices that become ready for Q_i during Π_i, and $R = \cup_{i=1}^{h-1} R_i$ is the signature set,
- C_i, $1 \leq i < h$, are disjoint sets of tree vertices that are children of $s_i \cup R_i$ and become ready within $N_i^+ \cap \Pi_i^{\rightarrow}$; $|C_i| = k_i$, and $C = \cup C_i$,
- $\Pi = \{\Pi_1, \ldots, \Pi_{h-1}\}$ is a partition of $[1, T-1]$,
- $\Pi^* = \{\Pi_{11}, \ldots, \Pi_{1k_1}, \ldots, \Pi_{h-1\ 1}, \ldots, \Pi_{h-1\ k_{h-1}}\}$ is a partition of $[1, T]$,
- X_{ij}, $1 \leq i < h$, $0 \leq j \leq k_i$ are sets of tree vertices that are disjoint from V and ready for Q^*_{ij} during Π^*_{ij}, $X = \cup X_{ij}$ and $|X| \geq T - |V|$,
- $Q = \{Q_1, \ldots, Q_h\}$ is a set of queues, such that for every $1 \leq i < h$, Q_i is the destination queue of s_i and also of every vertex in R_i and X_{i0}, and Q_h is the destination of s_h,

- $Q^* = \{Q^*_{11}, \ldots, Q^*_{1k_1}, \ldots, Q^*_{h-1\ 1}, \ldots, Q^*_{h-1\ k_{h-1}}\}$ is a set of queues, such that for every $1 \leq i < h, 1 \leq j \leq k_i$, Q^*_{ij} is the destination for the jth element in C_i and every vertex in X_{ij}.

From the earlier discussion, when the backtrack search takes T time steps to complete, there exists an execution template where the destination of vertices in S, R, C, and X satisfy the following conditions (let $D(v)$ be the random destination of vertex v):

1. $D(s_i) = Q_i$, $1 \leq i \leq h$.

2. $\forall v \in R_i$, $D(v) = Q_i$, $1 \leq i < h$.

3. Let v_{ij} be the jth element in C_i, $D(v_{ij}) = Q^*_{ij}$, $1 \leq i < h, 1 \leq j \leq k_i$.

4. $\forall v \in X_{ij}$, $D(v) = Q^*_{ij}$, $1 \leq i < h, 1 \leq j \leq k_i$.

5. $\forall v \in X_{i0}$, $D(v) = Q_i$, $1 \leq i < h$.

7.2.5 Estimating the Probability of Execution Templates

Let $\mathcal{L}$ be the event $T > k(\frac{n}{p} + h)$ and $|R| < \alpha T$. We bound the probability of event $\mathcal{L}$ by summing the probabilities of event $\mathcal{L}$ under all possible execution templates. For a fixed execution template the probability that all vertices in S, R, C, and X choose the right queue according to $\mathcal{E}$ is as follows.

$$p^{-|S \cup R \cup C|} p^{-(T - |S \cup R \cup C|)} = p^{-h} p^{-|R|} p^{-|C-(S \cup R)|} p^{-(T-|S \cup R \cup C|)}$$

Next, we count the number of different execution templates. The destinations of vertices from S and R are specified by Q, so the number of unspecified queues in Q^* is $|C - (S \cup R)|$ and the number of ways to choose Q and Q^* is $p^h p^{|C-(S \cup R)|}$. The total number of execution templates is therefore

$$n \binom{n}{|R|} \binom{2(|S| + |R|)}{|C|} \binom{n}{T - |V|} p^h p^{|C-(S \cup R)|} \binom{T + |C| + h}{|C| + h} \binom{T + h}{h}$$

Lemma 6 *For suitable constants k, α the probability that $T > k(\frac{n}{p} + h)$ and $|R| < \alpha T$ is polynomially small in n.*

Proof.

The probability of $\mathcal{L}$ is no more than the product of the number of different execution templates and the probability that every vertex in S, R, C, and X will actually choose the destination according to $\mathcal{E}$ when $T > k(\frac{n}{p} + h)$ and $|R| < \alpha T$.

$$
\begin{aligned}
& n\binom{n}{|R|}\binom{2(|S|+|R|)}{|C|}p^{-|R|}\binom{n}{T-|V|}p^{-(T-|V|)}\binom{T+|C|+h}{|C|+h}\binom{T+h}{h} \\
\leq\ & 2^{\log n}2^{2(|S|+|R|)}\binom{n}{\lceil\frac{n-p}{p+1}\rceil}p^{-\lceil\frac{n-p}{p+1}\rceil}\left(\frac{ne}{(T-|V|)p}\right)^{(T-|V|)}2^{T+|C|+h}2^{T+h} \\
\leq\ & 2^{(2\alpha+2)T+5h+|C|}\left(\frac{ne}{\lceil\frac{n-p}{p+1}\rceil p}\right)^{\lceil\frac{n-p}{p+1}\rceil}\left(\frac{ne}{(T-|V|)p}\right)^{T-|V|} \\
\leq\ & 2^{(2\alpha+2)T+5h+2h+2\alpha T}\left(\frac{ne}{\frac{n}{2p}p}\right)^{\frac{n}{p}}\left(\frac{ne}{(T-3\alpha T-3h)p}\right)^{T-3\alpha T-3h} \\
\leq\ & 2^{7h+(4\alpha+2)k(\frac{n}{p}+h)}(2e)^{\frac{n}{p}}\left(\frac{ne}{((1-3\alpha)k-3)(\frac{n}{p}+h)p}\right)^{((1-3\alpha)k-3)(\frac{n}{p}+h)} \\
\leq\ & 2^{((4\alpha+2)k+7)(\frac{n}{p}+h)}(2e)^{\frac{n}{p}}\left(\frac{e}{((1-3\alpha)k-3)}\right)^{((1-3\alpha)k-3)(\frac{n}{p}+h)} \\
\leq\ & \left[2^{(4\alpha+2)k+7}(2e)\left(\frac{e}{((1-3\alpha)k-3)}\right)^{((1-3\alpha)k-3)}\right]^{(\frac{n}{p}+h)} \\
\leq\ & 2^{-(\frac{n}{p}+h)},\ \text{for suitably chosen constants } k,\alpha.
\end{aligned}
$$

The first inequality follows from the observation that $|V| \leq 3(\alpha T + h)$ and $\binom{n}{x}p^{-x}$ is maximized when $x = \lceil\frac{n-p}{p+1}\rceil$. Finally, since $h \geq \log n$, the bound in the last step is polynomially small in n. ∎

From Lemmas 3, 6 we have the following theorem.

Theorem 7 *Let $\mathcal{T}$ be any binary backtrack search tree of size n and depth h. Let T be the total time for the random destination backtrack search algorithm to expand $\mathcal{T}$ in a p-node network. The probability that T exceeds $k(\frac{n}{p}+h)$, where k is suitably chosen, is polynomially small in n.*

7.3 Analysis of Branch-and-bound Search

The proof for backtrack search does not apply in the backtrack search case because an adversarial network arbiter can delay a critical node by favoring non-critical nodes. In the backtrack case, every tree node has to be expanded, therefore no matter which tree node the arbiter chooses to be received, some process is made. In the branch-and-bound case, although a critical node cannot be delayed by a non-critical node in the competition for the send buffer, it can be delayed by non-critical nodes in the competition for the same destination. An adversarial arbiter can work against the critical nodes so that they suffer long receive delays.

Our analysis of branch-and-bound search is based on the assumption that the network obeys the first-in-first-out (FIFO) scheduling policy. Under FIFO scheduling incoming vertices are received in time-stamp; a vertex that is ready cannot be delayed by a vertex that becomes ready at a later time-step; vertices that become ready at the same time can be received in arbitrary order.

We prove linear speedup in two steps. First, we prove that the aggregate delay of m non-overlapping receive delays is bounded by $O(m)$ with high probability under FIFO scheduling. Next, we show that for every execution there exists a signature set R and a set of $O(|R|+h)$ non-overlapping receive delays with aggregate delay $\Omega(T)$. As a result, it is very unlikely for T to be large and $|R|$ to be small. The other case, that of large T and large $|R|$ is already covered by Lemma 3.

7.3.1 Martingales

Lemma 7 *Let $X_1, \ldots, X_m$ be m random variables each in the range $[0..p-1]$, and let $X = \sum_{i=1}^{m} X_i$. Suppose that the conditional expectation $E(X_i \mid X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) \leq 1$, for all $1 \leq i \leq m$, and $0 \leq x_1, \ldots, x_{i-1}, \leq p-1$. Then, $Pr(X \geq \alpha m) \leq (\frac{1}{2})^{\alpha \frac{m}{p-1}}$ when $\alpha > 2e$.*

Proof. The analysis is similar to the generalized Chernoff bound given by Leighton etal. in [31]. We first estimate the expectation of e^{tX}.

$$
\begin{aligned}
E(e^{tX}) &= E(e^{tX_1} e^{tX_2} \cdots e^{tX_m}) \\
&= \sum_{x=0}^{p-1} e^{tx} E(e^{tX_2} \cdots e^{tX_m} | X_1 = x) Pr(X_1 = x)
\end{aligned}
$$

We then choose a value x_1^* for X_1 so that $E(e^{tX_2} \cdots e^{tX_m} | X_1)$ is maximized.

$$
\begin{aligned}
E(e^{tX}) &\leq \sum_{x=0}^{p-1} e^{tx} E(e^{tX_2} \cdots e^{tX_m} | X_1 = x_1^*) Pr(X_1 = x) \\
&\leq (\sum_{x=0}^{p-1} e^{tx} Pr(X_1 = x)) E(e^{tX_2} \cdots e^{tX_m} | X_1 = x_1^*) \\
&\leq E(e^{tX_1}) E(e^{tX_2} \cdots e^{tX_m} | X_1 = x_1^*) \\
&= E(e^{tX_1}) \sum_{x=0}^{p-1} e^{x} E(e^{tX_3} \cdots e^{X_m} | X_1 = x_1^*, X_2 = x) Pr(X_2 = x | X_1 = x_1^*) \\
&= E(e^{tX_1}) E(e^{tX_2} | X_1 = x_1^*) E(e^{tX_3} \cdots e^{tX_m} | X_1 \leq x_1^*, X_2 = x_2^*) \\
&\vdots \\
&\leq E(e^{tX_1}) E(e^{tX_2} | X_1 = x_1^*) \cdots E(e^{tX_m} | X_1 = x_1^*, \ldots, X_{m-1} = x_{m-1}^*)
\end{aligned}
$$

Each of these expectation is maximized when the probability is non-zero only at 0 and $p-1$; the endpoints of the range of X_i. In addition, we choose t so that $e^{t(p-1)}$ is larger than 1 and $E(e^{tX_i})$ is maximized when $Pr(X_i = p-1)$ is maximized. From Markov's inequality we can bound $E(e^{tX_i})$ by $Pr(X_i = 0) + Pr(X_i = p-1)e^{t(p-1)} \leq (1 - \frac{1}{p-1}) + \frac{1}{p-1}e^{t(p-1)}$ since $E(X_i)$ is bounded by 1 from the assumptions.

$$\begin{aligned} E(e^{tX}) &\leq ((1 - \frac{1}{p-1}) + \frac{1}{p-1}e^{t(p-1)})^m \\ &= (1 + \frac{e^{t(p-1)} - 1}{p-1})^m \\ &\leq e^{(\frac{e^{t(p-1)}-1}{p-1})m} \qquad (1 + y < e^y) \end{aligned}$$

Then we use Markov's inequality again to bound the probability that X is greater than αm.

$$\begin{aligned} Pr(X \geq \alpha m) &= Pr(e^{tX} \geq e^{t\alpha m}) \\ &\leq \frac{e^{(\frac{e^{t(p-1)}-1}{p-1})m}}{e^{t\alpha m}} \\ &= e^{-\frac{(\alpha \ln \alpha - \alpha + 1)m}{p-1}} \qquad (\text{when } t = \tfrac{\ln \alpha}{p-1}) \\ &\leq 2^{-\frac{\alpha m}{p-1}} \end{aligned}$$

■

Lemma 8 *Let $V = \{v_1, \ldots, v_m\}$ be m vertices with non-overlapping delays. The probability that their aggregate delay exceeds βm is smaller than $(\frac{1}{2})^{\frac{(\beta-1)m}{p-1}}$ when $\beta \geq 2e + 1$.*

Proof. Let Q_i be the destination of v_i and X_i be the number of vertices that will be received by Q_i before v_i when v_i becomes ready. The receive delay of v_i is $X_i + 1$ and the aggregate receive delay of V is $m + \sum_{i=1}^{m} X_i$.

Every vertex chooses its destination independently and uniformly; therefore, given $X_1, \ldots, X_{i-1}$, v_i is equally likely to pick any destination. We will argue that, given $X_1, \ldots, X_{i-1}$, the expected value of X_i is no more than 1. When v_i makes its random choice there are at most $p-1$ other ready vertices in the system whose choices are independent of v's choice. Therefore, the conditional expectation of X_i is less than one.

For the aggregate delay to exceed βm, the sum of all X_i must exceed $(\beta - 1)m$. The bound on the probability of this event follows from Lemma 7. ■

7.3.2 The Signature Set

As before we consider the special vertices $S = \{s_1, \ldots, s_h\}$. Let s_i be received by Q_i at time T_i for $1 \leq i \leq h$. For every $1 \leq i < h$ we seek a set of receive delays which together cover the interval $[T_i, T_{i+1}]$.

Let T_i^{ns} be the largest time step smaller than T_{i+1} at which the send buffer of Q_i is not occupied by a critical vertex, $(1 \leq i < h)$. Note that at each time step during the interval $\Gamma_i = [T_i^{ns}+1, T_{i+1}]$ the send buffer of Q_i is occupied by a critical vertex. Let c_i be the critical vertices that are injected into the network from Q_i during Γ_i. As a result, Γ_i can be partitioned into receive delays of vertices in c_i.

Among all the parents of vertices in c_i let f_i be the one that becomes ready at the earliest time step, say, T_i^f. Since $s_{i+1} \in c_i$ it follows that s_i is received no earlier than T_i^f.

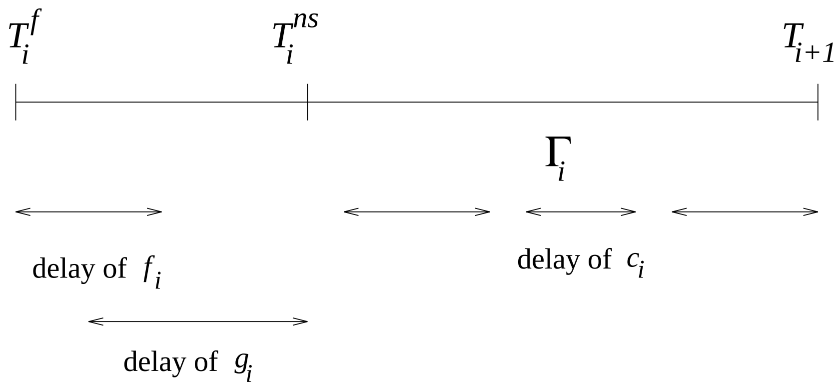


Figure 7.2: The interval Δ_i

As shown in Figure 7.2 it is possible for a gap to exist between the receive delays of f_i and vertices in c_i. In this case, the send buffer of Q_i must be occupied by a non-critical vertex, call it g_i, which is received at its destination at time T_i^{ns}. Observe that g_i cannot be received at its destination any earlier, for otherwise the send buffer of Q_i would have to be occupied by a critical vertex (the message queue contains at least one critical vertex, the child of f_i, and a critical vertex gets priority over non-critical vertices to enter the send buffer). But this contradicts the definition of T_i^{ns}.

Let T_i^g be the time step at which the parent of g_i becomes ready, and let $\Delta_i = [\min(T_i^f, T_i^g), T_{i+1} - 1]$. The following lemma summarizes our observations.

Lemma 9

1. *The parent of each vertex in c_i becomes ready for Q_i during Δ_i,*

2. *$\Delta_i^{\rightarrow}$ is the union of receive delays of vertices f_i, c_i, and g_i (if it exists), and*

3. $\bigcup_{i=1}^{h-1} \Delta_i = [1, T-1]$.

Proof. The parent of each vertex v in c_i must become ready before v; furthermore, it cannot become ready before f_i. For (2), if there is a gap between the receive delay of f_i and c_i, then this gap will be covered by the receive delay of g_i from the discussion above. Finally s_i must be ready at or after T_i^f from the definition of f_i so it cannot be received before T_i^f; thus the interval Δ_i contains $[T_i, T_{i+1})$, and (3) follows. ∎

From (3) of Lemma 9 the union of all Δ_i cover the interval $[1, T-1]$. We can therefore define a partition Π of $[1, T-1]$ as follows,

$$\begin{aligned} \Pi_{h-1} &= \Delta_{h-1} \\ \Pi_i &= \Delta_i - \bigcup_{j>i} \Pi_j, \quad 1 \le i < h-1. \end{aligned}$$

Definition. As before, a critical vertex v is in the *signature set* R if v is not a special vertex ($v \notin S$) and v becomes ready for Q_i during Π_i.

There are three kinds of receive delays in $\Pi_i^{\rightarrow}$: the earliest ready parent f_i, the non-critical vertex g_i, and those vertices in c_i that become ready during $\Gamma_i \cap \Pi_i$. We use F, G, and C to denote the sets of these three kinds of vertices from all Π_i. From (1) of Lemma 9 and the definition of R, the parent of every vertex in C is either in R or S. As a result the number of receive delays in $F \cup G \cup C$ is at most $2h + 2(|S| + |R|) = 4h + 2|R|$.

The receive delays identified thus far cover the interval $[1, T]$ and there are no more than $4h + 2|R|$ in number. They are not necessarily non-overlapping, however. Using a straightforward greedy procedure it is possible to produce a subset of no more than $2h + |R|$ intervals which are disjoint and whose union includes at least $T/2$ time steps. With this observation, we have the following theorem.

Theorem 8 *Let $\mathcal{T}$ be the critical branch-and-bound subtree of size n and depth h. Let T be the total time for the random destination algorithm to expand $\mathcal{T}$ in a p-node network under FIFO scheduling strategy. The probability that T exceeds $k(\frac{n}{p} + h)$ is polynomially small, for suitably chosen k, when $n > p^2 \log p$.*

Proof.

The probability that the signature set R exceeds αT in size is polynomially small by Lemma 3. From the above discussion we can identify a set of $2h + \alpha T$ non-overlapping receive delays whose aggregate delay is at least $T/2$. From the result of Lemma 8 this probability is bounded by $(\frac{1}{2})^{\frac{cT}{p}}$ for a suitable constant c. This quantity is polynomially small in n for $n > p^2 \log p$ and a suitably chosen k. ∎

Chapter 8

Conclusion

This dissertation introduces new techniques for implementing adaptive N-body methods efficiently on distributed memory parallel machines. The techniques overcome the major obstacles in developing efficient parallel N-body codes – dynamic adaptive data structures, non-uniform computational structures, and evolving irregular communication patterns. Experimental results show that these techniques significantly reduce the overhead due to parallelization. In addition, these techniques are general enough to be used in different N-body algorithms and other adaptive tree-structured computations.

The design and implementation of adaptive data structures in distributed memory is one of the most challenging problems in parallelizing non-uniform computations. For example, we spent tremendous effort to improve the implementation of the adaptive Barnes-Hut tree. Our techniques include incremental updates on the data structures and implicit representations of a global structure by combining local structures in every processor. These techniques have been integrated into a C++ library to hide the implementation details from the end users.

We found that incremental modification efficiently maintains slowly evolving data structures in the N-body simulations. Modifications to the data structures are carried out incrementally through explicit message-passing. For example, the Barnes-Hut tree is incrementally updated to conform to the dynamically changing distribution of bodies. Those bodies moving into new processor domains are shifted among processors through message-passing. Similar incremental changes are applied to ORB bisectors to dynamically balance the workload distribution without recomputing the entire ORB decomposition. Experimental data suggest that the incremental update approach is more efficient than rebuilding the data structures.

The method of adjusting local data structures to remove inconsistency, then combining them into a global structure is a generally useful technique. After data are partitioned among processors, every processor has only a local view of the entire data structure. These local views may be inconsistent because processor may not have all the information required to determine the shape of the underlining global structure. Usually these inconsistencies happen at those "boundary points" that are adjacent to the data partitioning hyperplanes. After the inconsistency on the boundary points are removed via communication, the shape of the data structure can be determined. We believe that this general technique can be used to implement adaptive data structures in other computations.

We found that the optimization of computation is one of the most significant factors in improving efficiency. Due to the large amount of computation in N-body simulations, any significant reduction on the force computation time reduces the total execution time considerably.

The major problem in vectorizing force calculations is that the irregular computational structures of adaptive N-body algorithms do not fit directly into the regular SIMD style computation that the CM-5 vector units perform best. To remove the irregularity of the computational structures and explore the performance potential of vector units, we cache essential nodes so that the vector units can work directly on an array instead of an irregular tree structure. In addition, we compute the accelerations for groups of bodies at a time. The irregularity of essential data for different bodies in a group is removed so that the computation becomes more uniform.

We found that the advantages of transforming the irregular computational structure into a regular shape are more significant than its complications. The complications of embedding an irregular data structure into a regular array are resolved by various algorithmic techniques with minute extra costs. However, the vector units are now able to access essential data and perform floating point operations at a much faster rate, as indicated by the experimental results. In addition, the extra computation due to grouping is insignificant compared with the cost to gather essential data for different bodies in the group when the group size is reasonably bounded.

We believe the tension between irregular computational structures found in most of the adaptive algorithms, and the regular computation structures computer hardware performs best, plays a crucial role in parallel efficiency. Parallel programs must bridge these two different endpoints with careful data structure and algorithm designs to achieve good parallel efficiency.

Experimental results indicate that our distributed memory implementation exhibits high efficiency and has very small communication overhead. The explicit message-passing does not translate into significant run-time overhead.

As the communication networks of parallel machines become more powerful, the topologies of the networks become less important. The communication network performs point-to-point message-passing communication so efficient that it can be viewed as fully connected. As a result we believe that the parallel program designers should pursue architecture-independent communication protocols, rather than exploring the advantages of a particular network topology.

The second part of this dissertation introduced a new communication model to capture the resource-contention phenomenon in parallel computers. The atomic message-passing model places significant restriction on resource consumption; each processor cannot send the next long atomic message until the previous one is received. Nevertheless there exist simple randomized protocol for message scattering, backtrack search, and brand-and-bound procedure. We show that when messages are scattered to randomly and independently chosen destinations, the throughput of the system is still optimal under the weak atomic message model. In addition, we show that with high probability, a simple randomized protocol achieves linear speedup for backtrack search and branch-and-bound search.

8.1 Future Work

Currently a parallel code for Fast Multipole Method is under development using the same techniques we introduce in this dissertation. The general techniques for maintaining irregular dynamic data structures and for efficient communication should be able to reduce the development time of the parallel fast multipole method. For example, the tree building stage in Barnes-Hut algorithm is essentially the same as in the fast multipole method. In fact, we believe that these techniques are general enough to be used in other tree-structured computations as well.

To estimate caching efficiency, we are able to bound the number of cache modifications for some special cases, but the general case of arbitrary θ and distribution of bodies remains open. Similar results may be established by restricting the distribution of bodies to some realistic class of distributions, and/or use a space-filling curve that guarantees adjacency of consecutive groups of bodies.

Our atomic message model differs from other communication models by emphasizing resource-efficient communications. However, it is still possible to link the atomic model to various communication models like the LogP model [14], postal model [5], optical models [21], some PRAM models including QRQW PRAM [22] and Distributed Memory Machine [16, 27]. The relations among these models is worth further investigation.

Although the randomized reactive protocol completes the tree search within constant factor of optimal time steps with high probability, the number of tree nodes stored in the message queues may be very large. The breadth-first nature of the protocol maintains a frontier of the tree within the message queues of all processors, and the size of the frontier may be much larger than the number of queues. Leiserson and Blumofe [12] proposed a new search procedure in which each processor performs a depth-first search so that space requirement is significantly reduced. Our reactive protocol for the atomic message model will be able to reduce queue length requirements if it can adopt Leiserson and Blumofe's space-efficient scheduling.

For technical reasons we assume the network arbiter is a FIFO in the analysis of branch-and-bound search. This FIFO assumption may not be realistic because of various delays in the routing process. However, the FIFO assumption may not be necessary to establish the theorem. To what extent the FIFO assumption can be weakened so that the linear speedup of the reactive branch-and-bound protocol still holds remains open.

In some message scattering problems we cannot assume the destinations are randomly and independently chosen (e.g. a h-relation). However, given a set of messages and their destination, we are allowed to reorder the message sending order. The standard bipartite graph edge coloring may not be suitable because of heavy communication overhead. Instead we can reorder the message at random. The intuition is that the messages having the same destination will be spread out in time so they do not block each other. The random ordering algorithm does not require communication and is totally distributed. However, it is still open that the random reordering can send all the message in an h-relation within $O(h)$ time.

Bibliography

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

[2] C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13, 1992.

[3] R. Anderson. Nearest neighbor trees and N-body simulation. *manuscript*, 1994.

[4] A.W. Appel. An efficient program for many-body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6, 1985.

[5] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

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

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

[8] J.J. Bartholdi and L.K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in Euclidean space. *Management Science*, 34, 1988.

[9] C. Berge. *Graphs and Hypergraphs*. North-Holland, 1973.

[10] D. Bertsimas and M. Grigni. Worst-case examples for the spacefilling curve heuristic for the Euclidean traveling salesman problem. *Operation Research Letter*, 8, 1989.

[11] S. Bhatt, M. Chen, C. Lin, and P. Liu. Abstractions for parallel N-body simulation. In *Scalable High Performance Computing Conference SHPCC-92*, 1992.

[12] R. Blumofe and C. Leiserson. Space-efficient scheduling of multi-threaded computations. In *25th Annual ACM Symposium on Theory of Computing*, 1993.

[13] P. Callahan and S. Kosaraju. A decomposition of multi-dimension point-sets with applications to k-nearest-neighbors and N-body potential fields. *24th Annual ACM Symposium on Theory of Computing*, 1992.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Eicken. Logp: Towards a realistic model of parallel computation. In *4th ACM PPOPP*, 1993.

[15] R. Das, J. Saltz, D. Mavriplis, J. Wu, and H. Berryman. Unstructured mesh problems, Parti primitives and the ARF compiler. In *Parallel Processing for Scientific Computation, Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[16] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[17] B. Chapman et. al. Vienna FORTRAN – A Fortran Language Extension for Distributed Memory Multiprocessors. In *High Performance FORTRAN Forum*, 1992.

[18] S. Hiranandani et. al. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12, 1991.

[19] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. In *High Performance FORTRAN Forum*, January 1992.

[20] H. Gabow. Using Euler partitions to edge color bipartite multigraphs. *International Journal of Computer and Information Sciences*, 5, 1976.

[21] M. Gereb-Graus and T. Tsantilas. Efficient optical communication in parallel computers. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[22] P. Gibbons, Y. Matias, and V. Ramachandran. QRQW: Accounting for concurrency in PRAMs and asynchronous PRAMs. Technical Report BL011211-930301-05TM, BellLab, 1993.

[23] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73, 1987.

[24] L. Hernquist. Vectorization of tree traversals. *Journal of Computational Physics*, 87, 1987.

[25] W.J. Kaufmann III and L.L. Smarr. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

[26] J. F. Leathrum Jr. and J. Board Jr. The parallel fast multipole algorithm in three dimensions. *manuscript*, 1992.

[27] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *24th Annual ACM Symposium on Theory of Computing*, 1992.

[28] R.M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *20th Annual ACM Symposium on Theory of Computing*, 1988.

[29] R.M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computations. *Journal of the ACM*, 40, 1993.

[30] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures On Distributed Memory Architectures. Technical report, ICASE, NASA Langley Research Center, 1990.

[31] F. T. Leighton, M. J. Newman, A. Ranade, and E.J. Schwabe. Dynamic tree embeddings in butterfly and hypercubes. In *1st Annual ACM Symposium on Parallel Algorithms and Architecture*, 1989.

[32] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The network architecture of the connection machine CM-5. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

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

[34] J. Makino. Comparison of two different tree algorithms. *Journal of Computational Physics*, 88, 1990.

[35] K. Mehlforn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21, 1984.

[36] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C.*, 1990.

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

[38] L.K. Platzman and J.J. Bartholdi. Spacefilling curves and the planar traveling salesman problem. *Journal of the ACM*, 1989.

[39] A. Ranade. A simpler analysis of the Karp-Zhang parallel branch-and-bound method. Technical Report UCB/CSD 90/586, University of California, 1990.

[40] A. Ranade. Optimal speedup for backtrack search on a butterfly network. In *3rd Annual ACM Symposium on Parallel Algorithms and Architecture*, 1991.

[41] J. Reif and S. Tate. The complexity of N-body simulation. In *International Colloquium on Automata Languages and Programming*, 1993.

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

[43] C.L. Seitz. Multicomputers. In *Developments in Concurrency and Communication, C.A.R Hoare (ed) Addison–Wesley, pp 131-201*, 1990.

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

[45] 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.

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

[47] R.E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[48] Thinking Machine Corporation. *CM-Fortran Programmer's Manual*, 1990.

[49] Thinking Machine Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[50] Thinking Machine Corporation. *CDPEAC: Using GCC to program in DPEAC*, 1993.

[51] Thinking Machine Corporation. *CMMD Reference Manual*, 1993.

[52] Thinking Machine Corporation. *DPEAC Reference Manual*, 1993.

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

[54] M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing*, 1993.

[55] F. Zhao. An $O(N)$ algorithm for three dimensional N-body simulation. Technical report, MIT, 1987.

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

[57] F. Zhao and S.L. Johnsson. The parallel multipole method on the connection machine. *SIAM Journal on Scientific and Statistical Computing*, 1991.