Distribution-independent hierarchical N-body methods

Aluru, Srinivas, Ph.D.

Iowa State University, 1994

Distribution-independent hierarchical N-body methods

by

Srinivas Aluru

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1994

DEDICATION

To my parents

For keenly providing me the

opportunities that they never had

and

my wife Maneesha,

For fighting against all odds to be a part of my life

For her love, patience, inspiration and encouragement

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I wish to thank my advisors Dr. G.M. Prabhu and Dr. John Gustafson, for taking personal interest in my professional advancement and for presenting challenging problems throughout my graduate study. I am thankful to Dr. John Gustafson for establishing the Scalable Computing Laboratory and providing a wonderful work environment. He took me as a research assistant so that I could concentrate on research and provided me the flexibility to work on problems that interest me. It has been a pleasure to work under such an exemplary researcher. Dr. Prabhu has been both a mentor and a friend. He has spent significant amount of time in research-related discussions and through motivation and an unwavering faith in my abilities has helped me during difficult times. Their role in helping me acquire the skills required to pursue a life-time research career cannot be overemphasized.

I am thankful to Dr. David Fernandez Baca for expressing keen interest in my research and for providing valuable suggestions and guidance from time to time. Thanks to Dr. Shashi Gadia and Dr. Charles Wright for their guidance and their role as members of my POS committee.

Throughout my stay at Iowa State University, I have been fortunate to receive guidance and encouragement from several faculty members including Dr. Giora Slutzki, Dr. Suraj Kothari, Dr. Soma Chaudhuri and Dr. Vasant Honavar. Special thanks are due to Dr. Giora Slutzki for bringing to my attention and loaning new books and research articles of interest to me.

I wish to thank the staff and fellow students at the Scalable Computing Laboratory including, but not limited to - Nan Ripley, Joe Metzger, Charles S. Shorb, Yogesh Agrawal, Rekha Pai, Dr. Thomas L. Marchioro, Venu Padakanti, Quinn Snell, Dr. Susan X Ying, Sairam, Anthony M. Baker and Steve Heistand for their assistance, discussions and friendship. A special note of thanks to our secretary Nan Ripley for her prompt, efficient and useful service.

I wish to express my gratitude to my parents and sisters for their assistance and encouragement throughout my life and to my wife Maneesha, for enduring me all these years and for being a valuable source of support.

This research is supported by the Applied Mathematical Sciences program of the Ames Laboratory which is operated by the U.S. Department of Energy under contract No. W-7405-ENG-82. The United States Government has assigned the DOE number IS-T 1643 to this dissertation.

ABSTRACT

The N-body problem is to simulate the motion of N particles under the influence of mutual force fields based on an inverse square law. The problem has applications in several domains including astrophysics, molecular dynamics, fluid dynamics, radiosity methods in computer graphics and numerical complex analysis. Research efforts have focused on reducing the $O(N^2)$ time per iteration required by the naive algorithm of computing each pairwise interaction. Widely respected among these are the Barnes-Hut and Greengard methods. Greengard claims his algorithm reduces the complexity to $O(N)$ time per iteration.

Throughout this thesis, we concentrate on rigorous, distribution-independent, worst-case analysis of the N-body methods. We show that Greengard's algorithm is not $O(N)$, as claimed. Both Barnes-Hut and Greengard's methods depend on the same data structure, which we show is distribution-dependent. For the distribution that results in the smallest running time, we show that Greengard's algorithm is $\Omega(N log^2 N)$ in two dimensions and $\Omega(N log^4 N)$ in three dimensions. Both algorithms are unbounded for arbitrary distributions.

We have designed a hierarchical data structure whose size depends entirely upon the number of particles and is independent of the distribution of the particles. We show that both Greengard's and Barnes-Hut algorithms can be used in conjunction with this data structure to reduce their complexity. Apart from reducing the complexity of the Barnes-Hut algorithm, the data structure also permits more accurate error estimation. We present two- and three- dimensional algorithms for creating the data structure. The multipole method designed using this data structure has a complexity of $O(N \log N)$ in two dimensions and $O(N \log^2 N)$ in three dimensions.

CHAPTER 1. INTRODUCTION

The N-body Problem

A large number of physical systems can be studied by simulating the interactions between the particles constituting the system. In a typical system each particle influences every other particle, often based on an inverse square law such as Newton's law of gravitation or Coulomb's law of electrostatic interaction. Examples of such physical systems can be found in astrophysics, plasma physics, molecular dynamics and fluid dynamics. Since the simulation involves following the trajectories of motion of N particles, the problem is termed the N-body problem. Apart from traditional applications in the study of physical systems, some problems in numerical complex analysis and elliptic partial differential equations can also be solved using this approach [14]. Applications of the problem are also found in the radiosity method, which attempts to create images by computing the equilibrium distribution of light for complex scene geometries [17, 35].

Since no closed form expression is known for the equations of motion for a collection of four or more particles, iterative methods are used to solve the N-body problem. In each iteration, the force on each particle due to every other particle is computed using the inverse square force law. This is used to compute the acceleration of the particle, which is presumed to be constant over a small interval of time δt.

The approximate position and velocity of the particle at the end of the time interval is calculated using this acceleration. The position of each particle after an arbitrary length of time is calculated by many iterations of this method. A straightforward computation of all the pairwise forces requires $O(N^2)$ work per iteration. The rapid growth with N effectively limits the number of particles that can be simulated by this method.

Early Approaches

To facilitate the study of large systems, several approaches have been tried to reduce the $O(N^2)$ work per iteration required by the naive algorithm of pairwise force computation. All the approaches are based on the following principle: In performing N-body simulations, there is an error introduced in assuming that the accelerations remain constant during the time intervals corresponding to the iterations. Even if the exact force on each particle is computed, this force changes during the time interval of the iteration and the computed force is only an approximation of the force acting on the particle during the time interval. Therefore, it is sufficient to compute the approximate force acting on each particle to a high degree of accuracy. This observation can then be used to reduce the complexity per iteration.

One of the approaches used is to represent the problem in a position-velocity phase space and to transform the force field using a fast Fourier transform into a form in which it can be applied in linear time [29, 30]. The time per iteration is now dominated by the computation of the Fourier transform, which requires $O(N \log N)$ time. To use this method, the phase space must be discrete - all velocities must be less than some maximum and all positions must be multiples of some fixed lattice size. Hence, the method is not useful for non-uniform distributions.

Another approach is to use variable time steps depending on the distance between the particles [1]. Recall that the force on a particle is assumed to be constant over a small interval of time δt. The time interval has to be very small for nearby particles since the force could change significantly with a small change in the position. A larger time interval can be chosen to approximate the interaction between faraway particles. For each particle, the force due to nearby particles is computed every iteration but the force due to faraway particles is computed using larger time-steps depending on their distance from the particle. For a non-uniform distribution, the complexity of the method degenerates to $O(N^2)$.

Another alternative is to impose a grid on the system of particles [20] and to use a *fast Poisson solver* to obtain the potential values at the mesh points. The forces are then computed from the potential and interpolated to the particle positions. Such methods are applicable if the potential satisfies Poisson's equation, which is true for gravitational and electromagnetic interactions. The complexity of these methods is $O(N + M \log M)$, where M is the number of mesh points. The number of mesh points should be proportional to the number of particles, resulting in an asymptotic complexity of $O(N \log N)$. Unfortunately, the method is useful only for uniform distributions since the mesh provides limited resolution otherwise. It is possible to compute the forces due to the nearby particles directly and to compute the forces due to faraway particles by extrapolating from the mesh points. In highly inhomogeneous systems, the number of nearby particles may be of the same order as the total number of particles, resulting in $O(N^2)$ complexity.

All of the methods discussed above are useful only for relatively uniform distributions. Except for some applications in plasma physics, most N-body simulations involve non-uniform distributions. In such cases, the methods described have the same worst-case complexity as directly computing all pairwise interactions.

Hierarchical Methods

Recently, a new class of particle simulation methods have emerged to solve the N-body problem efficiently for arbitrary distributions. These methods are characterized by an organization of the particles into a hierarchy of clusters, starting from a cluster containing all the particles to clusters containing the individual particles. These methods are usually referred to as *hierarchical methods*, or *tree methods* since a tree naturally represents a hierarchical organization of clusters. Such a hierarchical method was first proposed by Appel [4, 5], whose scheme allows for clusters with arbitrary shapes.

Appel's Method

Appel's method [4, 5] is based on the approximation that a cluster of particles can be treated as a single particle of equivalent mass located at the center of mass of the cluster, for the purpose of force calculation with a faraway particle. More formally, consider two particles m_1 and m_2, each at a distance of no more than $|\vec{dr}|$ from their center of mass (see Figure 1.1). The acceleration imparted due to the two particles at a point situated at a distance $|\vec{r}|$ from the center of mass is

$$\vec{a} = \frac{Gm_1(\vec{r} + \vec{dr}_1)}{|\vec{r} + \vec{dr}_1|^3} + \frac{Gm_2(\vec{r} + \vec{dr}_2)}{|\vec{r} + \vec{dr}_2|^3}$$

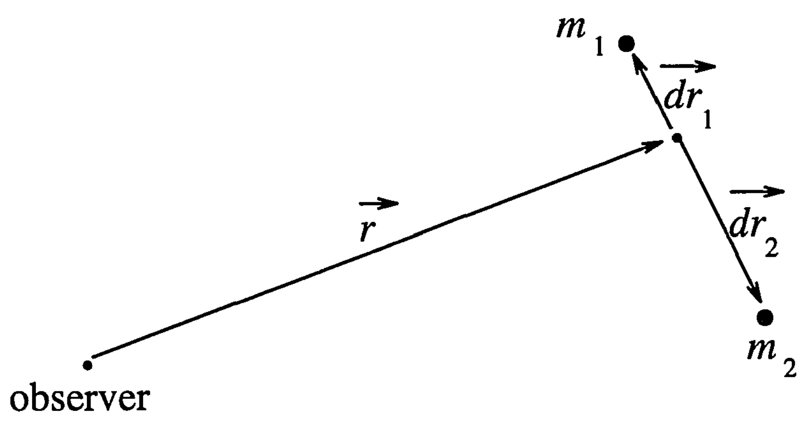

Figure 1.1: The monopole approximation

Expanding the denominators of the two terms using Taylor series expansion, the acceleration turns out to be

$$\vec{a} = \frac{G(m_1 + m_2)\vec{r}}{|\vec{r}|^3} + O((\frac{|d\vec{r}|}{|\vec{r}|})^2)$$

Since the various Taylor series are all expanded around the center of mass, the first order terms vanish and the approximation is good to a second order.

Let the radius of a cluster of particles be the largest distance from the center of mass to any particle in the cluster and let $0 \leq \theta < 1$ be a prespecified accuracy criterion. Consider two disjoint clusters of particles with radii dr_1 and dr_2, located at a distance r from each other. If $\frac{dr_1}{r} < \theta$ and $\frac{dr_2}{r} < \theta$, the acceleration on any particle in one cluster due to the particles in the other cluster is approximated with the acceleration at the center of mass of the first cluster resulting from treating the second cluster as a point mass located at its center of mass. Otherwise, the cluster with the larger radius is split and the interaction is computed by summing the interactions of the smaller cluster with each of the subclusters of the larger cluster,

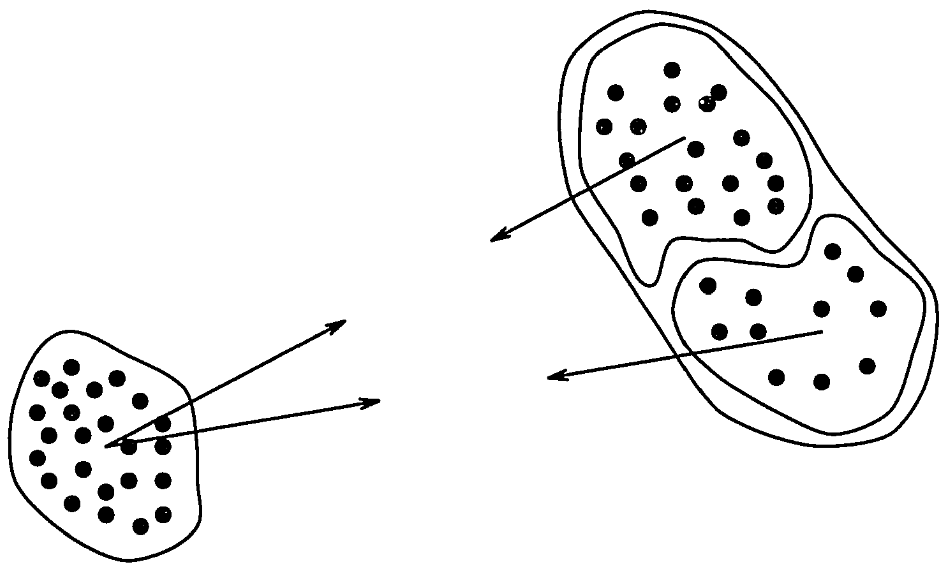

Figure 1.2: An example of force calculation using clusters

computed recursively (see Figure 1.2).

In Appel's method, the space is subdivided into a hierarchy of clusters with each cluster split into two subclusters. The subdivision is naturally represented by a binary tree. The root of the tree represents a cluster consisting of all the particles and the leaves represent individual particles. The children of a node represent the subclusters of the cluster represented by the node. The acceleration calculations are performed by traversing the tree starting at the root. The accelerations of all particles in a cluster are computed by computing the accelerations due to interactions within each of the subclusters and the accelerations due to interactions between the subclusters. A heuristic is used to update the clusters as the particles move and a k-d tree [5] is used as the hierarchy of clusters to begin with. Appel estimates the complexity of his algorithm to be $O(N \log N)$ based on arguments which apply only to a uniform distribution.

Barnes-Hut Method

The Barnes-Hut method [7] is similar to the Appel's method except for two differences. First, Barnes and Hut use a fixed hierarchical cubical subdivision of the space. The resulting tree is a quadtree in two dimensions and an octree in three dimensions, popularly referred to in the literature as the Barnes-Hut (BH) tree. Adopting a fixed structure for the clusters facilitates the possibility of rigorous error analysis.

Barnes and Hut do not approximate cluster-to-cluster interactions but approximate particle-to-cluster interactions only. The Barnes-Hut method consists of traversing the BH tree once for every particle to determine the force on it. The same criterion as in Appel's method is used to decide if the interaction of the particle with a cluster should be computed directly or by summing the interactions of the particle with the subclusters of the cluster, obtained recursively.

Barnes and Hut give arguments to support an $O(N \log N)$ complexity for their algorithm. The arguments apply only to uniform distributions. Salmon [31] studies the Barnes-Hut algorithm in great detail.

Greengard's Method

Greengard's algorithm [14] computes the potential induced on each particle by the rest of system and obtains the force as a gradient of this potential. Greengard's method, also knows as the fast multipole method (FMM), uses a series expansion to describe the potential induced by a cluster of particles at a given position. The series expansion is called the *multipole expansion* and it accurately describes the potential due to the cluster of particles at a given point. A finite number of terms of the series are used depending on the accuracy of the answer required.

Greengard's algorithm uses the same hierarchical subdivision of space as the Barnes-Hut algorithm. The algorithm is a two-pass procedure on the BH tree. The first pass is a bottom-up traversal of the BH tree to compute the multipole expansions at all nodes. The second pass is a top-down traversal of the tree to compute a series expansion at every node for the potential induced on the cluster represented by the node due to the rest of the system, termed the *local expansion.* The local expansions at the leaf nodes are then evaluated once for each particle. Greengard develops a detailed mathematical formalism and estimates the complexity of his algorithm to be $O(N)$ irrespective of the distribution of the particles.

Other Methods

Most of the literature on hierarchical N-body methods consists of a detailed study of the three methods described above, new methods with minor variations or adapting these methods to parallel architectures.

Esselink [12] argues that the complexity of Appel's method is $O(N)$. Salmon [31] studies the Barnes-Hut algorithm in great detail and incorporates the computation of multipoles into the Barnes-Hut algorithm. Zhao [39] presents a multipole algorithm based on cartesian coordinates as opposed to the spherical harmonics used by Greengard. Katzenelson [23] introduces a formulation of the N-body problem as a set of recursive equations based on a few elementary functions. His formulation encompasses both Barnes-Hut and Greengard's algorithms.

A Unified Framework for Hierarchical N-body Methods

All the hierarchical N-body methods are based on approximating the interactions of clusters of particles instead of dealing with the individual particles. There are two different aspects to consider in such an approach.

1. The determination of the clustering scheme.

2. The method used to approximate the interactions due to a cluster.

The different algorithms can be thought of as different choices exercised in the two aspects. At one extreme, we have the naive algorithm which performs no clustering. The resulting method is an $O(N^2)$ algorithm of computing the exact pairwise interactions. Appel's algorithm uses a clustering scheme based on heuristics while the Barnes-Hut and Greengard algorithms use a clustering scheme based on fixed cubical subdivision. The interactions due to a cluster can be written in the form of a Taylor series. The interactions of a cluster can be approximated by taking either one term of the Taylor series (the monopole method) or a finite number of terms (the multipole method). In the monopole method, the desired accuracy is achieved by splitting the cluster into subclusters recursively until the monopole terms can describe the interaction to the required accuracy. The multipole method achieves the same by taking as many terms as needed.

Outline of the Dissertation

Throughout this thesis, we concentrate on rigorous worst-case analysis of the complexity of the N-body methods. Even though the hierarchical algorithms are claimed to be efficient for non-uniform distributions, researchers have often used arguments based on uniform distributions to justify their claims. With the notable exception of Greengard, most researchers paid little attention to a rigorous worst-case complexity analysis.

In Chapter 2, we analyze the Greengard and Barnes-Hut methods. Both the methods are based on a fixed hierarchical cubical subdivision of the space, represented by the BH tree. We analyze the characteristics of the BH tree to determine the lower and upper bounds on the size of the tree. The results are used to determine the complexity of the Greengard and Barnes-Hut methods. We show that Greengard's algorithm is not $O(N)$, as claimed. We prove that Greengard's algorithm is $\Omega(N \log^2 N)$ in two dimensions and $\Omega(N \log^4 N)$ in three dimensions and that the actual complexity matches this lower bound only for uniform distributions. We also show that both algorithms are unbounded for arbitrary distributions.

In Chapter 3, we describe a distribution-independent data structure for the N-body problem. The data structure is presented as a modification to the BH tree to remove its distribution-dependency. We show that the modified tree has a size of $O(N)$ and contains the same information as the BH tree. We prove that the Barnes-Hut and Greengard's methods can be run on the modified tree. Greengard's multipole method can be run on the modified tree to obtain the force calculations in $O(N)$ time.

In Chapter 4, we describe an algorithm to construct the modified tree in two dimensions. We show a lower bound of $\Omega(N \log N)$ and present an algorithm to construct the tree in time matching the lower bound. Chapter 5 consists of an algorithm to construct the modified tree in three dimensions, requiring $O(N \log^2 N)$

time.

The time per iteration of the N-body problem using the multipole method is dominated by the time required to create the distribution-independent data structure. The multipole algorithm on the new data structure computes the forces in $O(N \log N)$ time in two dimensions and in $O(N \log^2 N)$ time in three dimensions irrespective of the distribution of the particles.

In Chapter 6, we conclude the dissertation and briefly outline some possibilities for future work.

CHAPTER 2. ANALYSIS OF GREENGARD AND BARNES-HUT METHODS

In this chapter, we analyze the Greengard and Barnes-Hut methods to determine their worst-case running times for arbitrary distributions. Both algorithms use a clustering scheme in which the subclusters of a cluster are determined independent of the location of the particles in the cluster. Since the clusters are to be recursively subdivided until each cluster contains only one particle, the number of subdivisions with such a fixed clustering scheme is dependent on the distribution of the particles. As a result, the running time of these algorithms is sensitive to the distribution of the particles.

The Barnes-Hut and Greengard Methods

The Greengard and Barnes-Hut methods for computing N-body interactions consist of two alternating phases, repeated every time step:

1. Computing a hierarchical tree data structure with the leaves representing the particles and the root of the tree representing the entire system

2. Traversing this data structure to compute the force on each particle to a specified accuracy.

The same tree data structure is used in both the methods, known as the BH tree. In the Barnes-Hut method, the BH tree is traversed once for every particle according to the force calculation scheme given by Appel [5]. The Greengard's method is a two-pass procedure on the BH tree. In the first pass, the tree is traversed bottom-up to compute the multipole expansions at every node. In the second pass, the tree is traversed top-down to compute the local expansions. The local expansions are finally evaluated to compute the approximate force on each particle.

Analysis of the BH Tree

The BH tree is constructed as follows: Begin with a cell (square in two dimensions and cube in three dimensions) large enough to contain all the particles, called the *root cell.* Let d be the number of dimensions. Subdivide this cell into 2^d cells of half the side length of the original cell. For each of these subcells:

1. If the subcell does not contain any particles, discard it.
2. If the subcell contains exactly one particle, do not subdivide this subcell further.
3. If the subcell contains more than one particle, recursively subdivide this subcell.

This recursive subdivision of the space into cells is naturally represented by a tree, which is the BH tree. Such a physical subdivision of a system of 16 particles in two dimensions is shown in Figure 2.1.

A characteristic of the BH tree is that each node in the tree represents a cell of length exactly half that of its parent cell. This is true irrespective of the number of particles of the parent cell contained in the child cell. In particular, a child cell may contain the same particles as its parent cell. As we shall see, this feature makes it

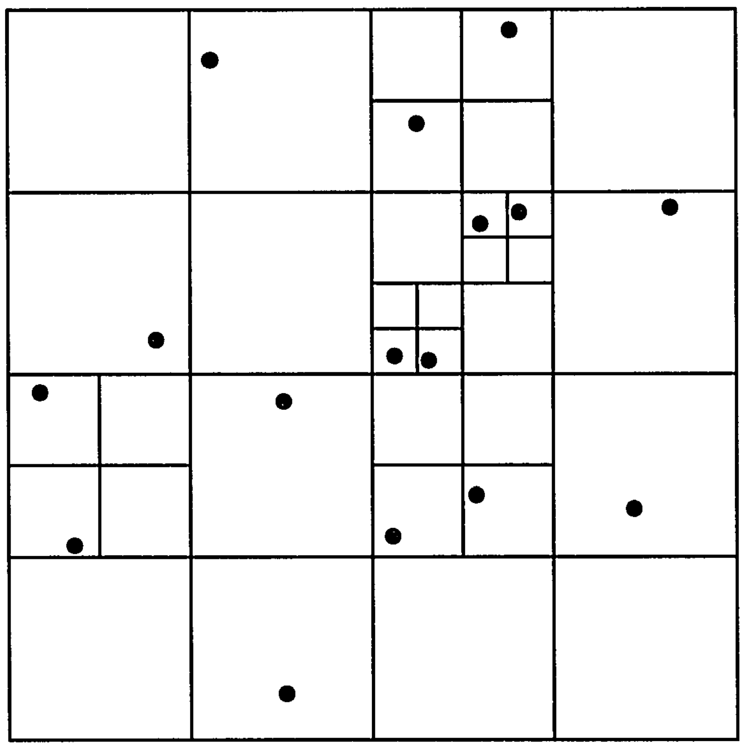

Figure 2.1: The Barnes-Hut physical subdivision of a system of 16 particles in two dimensions

impossible to establish a bound on the size of the tree as a function of the number of particles. For convenience and simplicity, a two-dimensional problem is discussed but the results carry over to three-dimensional problems as well.

Figure 2.2 shows the Barnes-Hut physical subdivision for a collection of three particles in two dimensions. The corresponding BH tree is shown in Figure 2.3. In the example shown, the first subdivision separates particle 1 from particles 2 and 3. The next three subdivisions performed to separate particles 2 and 3 are not successful as one of the child cells at every level of the subdivision contains both the particles and the other three contain none. The recursive subdivision is continued until the particles 2 and 3 are separated.

From this example, it is intuitively clear that a large number of recursive subdivisions may be required to separate particles that are very close to each other. It is not true that two particles can be separated only when the cell size is small enough such that a single cell cannot contain both the particles. Figure 2.4 illustrates this point. The distance between the two particles shown in the figure is much smaller than the length of the cells separating them. However, they are positioned in such a way that a single subdivision separates them, even though the size of the child cells in the subdivision is large enough to contain both the particles. In the worst case, the recursive subdivision continues until the cell sizes are so small that a single cell positioned anywhere cannot contain both the particles. Subdivision is never required beyond this point, but the particles may be separated sooner.

Let N be the number of particles in the system and let s be the smallest inter-particle distance. We require $s > 0$ to avoid infinite interaction force. Let D be the length of a cell that can contain all the particles. Clearly, the worst-case path length

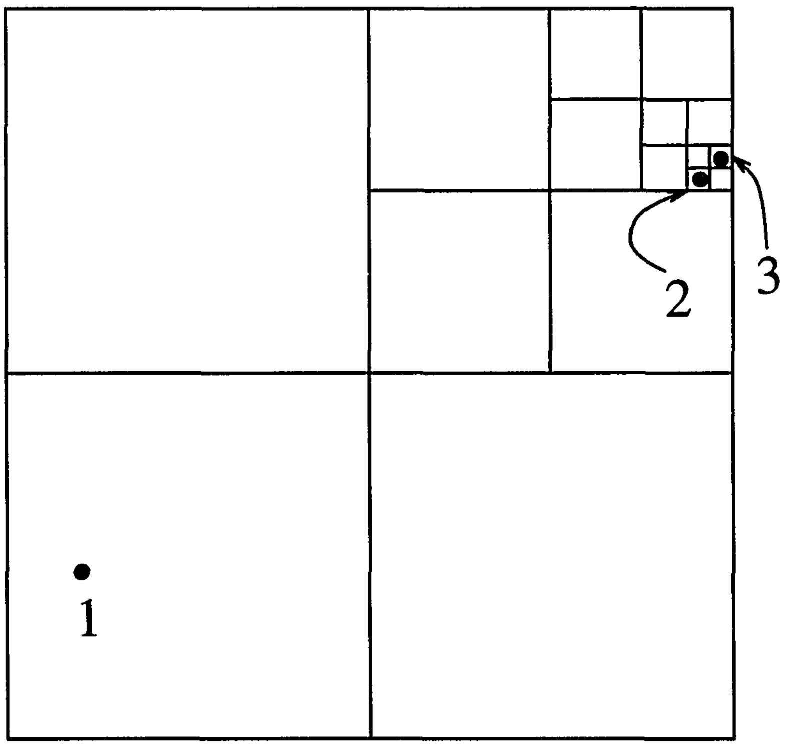

Figure 2.2: The Barnes-Hut subdivision of a system of particles positioned such that a subcell contains the same particles as its parent cell

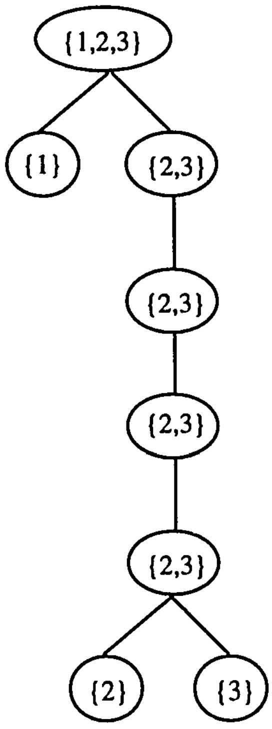

Figure 2.3: The BH tree corresponding to the subdivision of Figure 2.2

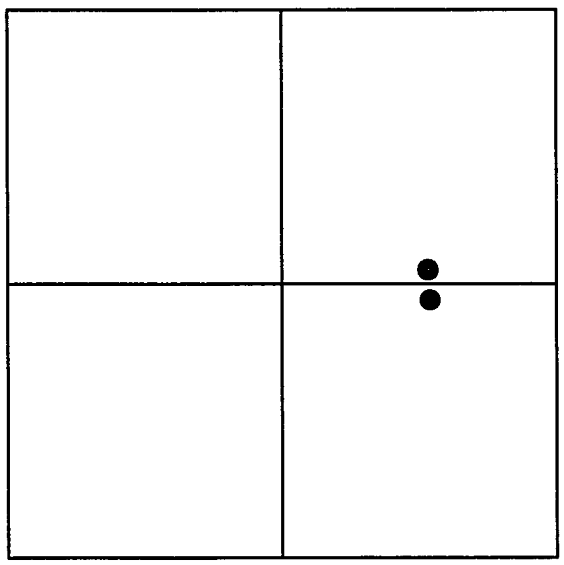

Figure 2.4: A configuration where two close particles are separated by cells large enough to contain both

of the BH tree is given by the worst-case path needed to separate the two particles which are closest to each other. The length of the smallest cell that can contain two particles s apart in two dimensions is $\frac{s}{\sqrt{2}}$ ($\frac{s}{\sqrt{3}}$ in three dimensions; see Figure 2.5).

The paths separating the closest particles in a two-dimensional problem may contain recursive subdivisions until a cell of length smaller than $\frac{s}{\sqrt{2}}$ reached. Since each subdivision halves the length of the cells, the maximum path length is given by the smallest k for which

$$\frac{D}{2^k} < \frac{s}{\sqrt{2}}$$

$$k = \lceil \log \frac{\sqrt{2}D}{s} \rceil$$

In three dimensions,

$$\frac{D}{2^k} < \frac{s}{\sqrt{3}}$$

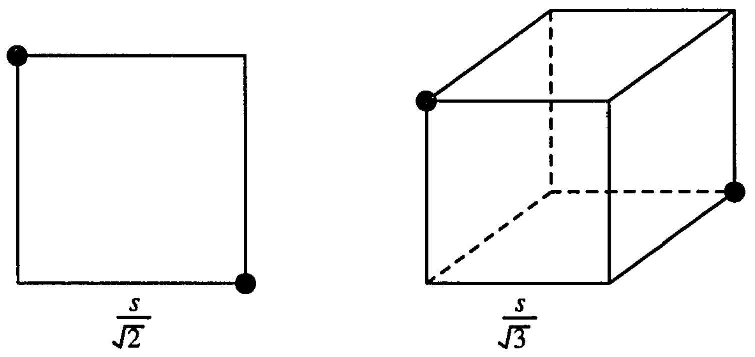

Figure 2.5: Smallest cells that could possibly contain two particles that are s apart in two and three dimensions

$$k = \lceil \log \frac{\sqrt{3}D}{s} \rceil$$

In either case, the worst-case path length is $O(\log \frac{D}{s})$. Since the tree has N leaves, the number of nodes in the tree is bounded by $O(N \log \frac{D}{s})$.

The absence of N in the expression determining the worst-case path length may seem rather strange. One might be curious to ask if N is related to D and s and thus implicitly determines the worst-case path length. Let us examine the behavior of $\frac{D}{s}$ as a function of N. In particular, we shall investigate the upper and lower bounds for $\frac{D}{s}$ as a function of N.

To minimize the ratio $\frac{D}{s}$ for a fixed N, all the particles should be at a distance of s from their nearest neighbors. To see why, suppose this is not true. We can reduce D by 'moving-in' particles that are farther than s from each other, while keeping s the same. Or, we can increase s by increasing the distance between particles that are s apart, keeping D unchanged. In either case, $\frac{D}{s}$ decreases, contradicting minimality. Furthermore, the particles must be packed as closely as possible. Figure 2.6 shows the configuration minimizing the ratio $\frac{D}{s}$ for a fixed N in two dimensions. Each particle has six nearest neighbors, all at a distance s. The particle is at the center of the hexagon formed by its nearest neighbors. The particles do not fit in a cell smaller than $D \times D$. Adding the particles column-wise,

$$N = \lfloor \frac{D}{s} + 1 \rfloor + \lfloor \frac{D}{s} \rfloor + \lfloor \frac{D}{s} + 1 \rfloor + \ \ldots \qquad (\lfloor \frac{2D}{\sqrt{3}s} + 1 \rfloor terms)$$

$$N \leq \frac{D}{s} \left[\frac{2D}{\sqrt{3}s} + 1 \right] + \frac{D}{\sqrt{3}s} + 1$$

$$N \leq \frac{2}{\sqrt{3}} \frac{D^2}{s^2} + \left(1 + \frac{1}{\sqrt{3}} \right) \frac{D}{s} + 1$$

$$\frac{D}{s} \geq c_1 \sqrt{N}$$

for some constant c_1. Since this is computed using the configuration minimizing the ratio $\frac{D}{s}$, the worst-case path length ($\log \frac{D}{s}$) is $\Omega(\log N)$. In three dimensions,

$$\frac{D}{s} \geq c_2 N^{\frac{1}{3}}$$

In either case,

$$\log \frac{D}{s} = \Omega(\log N)$$

Next, let us investigate how large $\frac{D}{s}$ can be for a fixed N. For any $N \geq 3$ particles, $\frac{D}{s}$ can be made arbitrarily large by reducing the distance between the closest particles (thus reducing s), or by increasing the spread of the particles (thus increasing D). Hence, the worst-case path length does not have an upper bound as a function of the number of particles and is entirely dependent upon the spatial distribution of the particles. This immediately implies that the size of the BH tree is unbounded and can be arbitrarily large for a fixed N. Since both Greengard's and

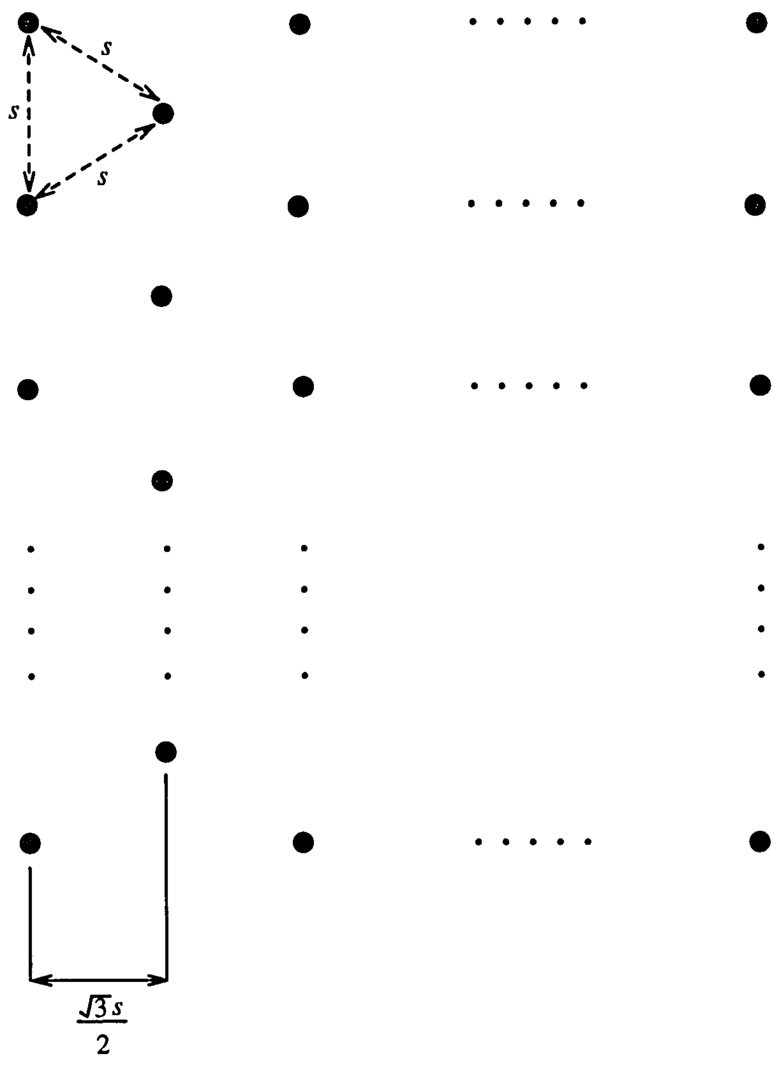

Figure 2.6: The configuration minimizing the ratio of the cell length containing all the particles and the smallest interparticle distance in two dimensions

Barnes-Hut algorithms construct and visit each node in the BH tree at least once, these algorithms are unbounded for arbitrary distributions.

In practice, the simulations have to be run on a machine with finite precision. With finite precision, there is a largest expressible number and a smallest expressible positive number. Once the precision is fixed, $\frac{D}{s}$ is bounded as a function of this precision. Greengard assumes the precision to be a constant in analyzing the complexity of his algorithm. The problem with this approach is discussed in detail in the following section.

On the Complexity of Greengard's Algorithm

Greengard assumes the length D of the cell containing all the particles to be one. His arguments can be summarized as follows: For a fixed machine precision ϵ, only certain classes of particle distributions can be modeled, independent of the algorithm used. In order to make the simulation possible, Greengard requires that the smallest distance s between any pair of particles be greater than ϵ. Thus, $\log \frac{D}{s}$ is bounded by $p = \lceil \log \frac{1}{\epsilon} \rceil = \lceil -\log \epsilon \rceil$. Greengard's algorithm takes the precision parameter ϵ as input. The force acting on each particle is also computed to the same precision. It turns out that the first p terms in the multipole and local expansions are enough to achieve the desired accuracy in force calculation. The algorithm, therefore, computes p-term multipole and local expansions. Since ϵ is a constant, p is a constant. Greengard estimates the running time of his algorithm to be $N(\alpha p^2 + \beta p + \gamma)$ in two dimensions and $N(\alpha p^4 + \beta p^2 + \gamma)$ in three dimensions, where α, β and γ are constants. Since p is taken to be a constant, Greengard [14] claims his algorithm runs in $O(N)$ time in two or three dimensions.

If we need a cell of length D to contain all the particles, we can force it to be one by appropriate scaling. Since this scaling does not change the ratio of the size of the cell containing all the particles and the smallest distance between any pair of particles, without loss of generality, $\log \frac{D}{s}$ is bounded by p.

The above arguments imply that the height of the tree is bounded by $O(p)$, a constant. Yet, we know that the height of a tree with N leaves and at most a constant number of children per node is $\Omega(\log N)$. How can this disparity be explained?

To further highlight the discrepancy, consider the first step in Greengard's algorithm - the construction of the tree representing the hierarchical subdivision. At every level of the tree, the nodes containing more than one particle (or more than a fixed number of particles) are subdivided and particles in each parent box are distributed among its child boxes. Since each particle is assigned to a box at every level and there are at most p levels, the work involved is proportional to Np. Since p is a constant, the complexity is computed to be $O(N)$.

Consider running this algorithm on a uniform distribution. Each child cell contains exactly a fourth of the particles of its parent cell. The resulting tree is a quadtree with $\log N$ levels and the work involved in constructing the tree is easily seen to be $O(N \log N)$, not $O(N)$.

The problem lies in the assumption that the parameters D and s are entirely dependent on the spatial distribution of the particles and not related to the number of particles N. We have seen that for any $N \geq 3$ particles, $\frac{D}{s}$ can be made arbitrarily large. This validates the argument that for a fixed machine precision, only certain classes of particle distributions can be modeled, independent of the algorithm used.

In the previous section, we have shown that $\log \frac{D}{s}$ has a lower bound of $\Omega(\log N)$.

Since $\log \frac{D}{s}$ is bounded by p, p is also $\Omega(\log N)$.

How does this translate to what classes of particle distributions can be modeled with a machine precision ϵ? It is already noted that not all distributions can be modeled for any given $N \geq 3$ because of precision limits. However, unless $p = \lceil -\log \epsilon \rceil \geq c \log N$ (c a constant), no distribution can be modeled for that N. The very fact that we are able to run an N-body problem for a collection of N particles with precision ϵ implies that $p = \lceil -\log \epsilon \rceil \geq c \log N$. Thus, p cannot be taken as a constant in the analysis of the running time of the algorithm and Greengard's algorithm is not $O(N)$. Greengard's time complexity is $\Omega(N \log^2 N)$ in two dimensions and $\Omega(N \log^4 N)$ in three dimensions. The running time matches the lower bound only for a uniform distribution. For arbitrary distributions, the running time is unbounded.

Two different precisions are involved in the simulation of an N-body problem. The first is the precision used to represent the input: the positions and velocities of the particles etc.. The second precision is the accuracy to which the force acting on each particle should be approximated. Greengard's algorithm computes the force to the same precision as used to represent the input. This results in a lower bound of $\Omega(\log N)$ for the precision parameter p. The precision used for the force calculation need not be the same as the precision used to represent the input. With a better precision to represent the input, a larger number of particles and/or a larger variety of distributions can be modeled. The precision used to compute the forces should be related to the duration of the time steps used in the simulation and the accuracy of the final answer required. In a reasonable simulation, the precision to which the force on each particle is computed should be of the same order as the change in the force on the particle during the time step. Computing the force to a higher precision than the change in the force during the time step is not useful.

On the Complexity of the Barnes-Hut Algorithm

Barnes and Hut [7] estimate that the tree construction can be accomplished in $O(N \log N)$ time and that the force on each particle can be computed in $O(\log N)$ time. The cost per iteration is thus estimated to be $O(N \log N)$. Their arguments apply only to a uniform distribution of particles. Unfortunately, running times based on uniform distributions are often extrapolated to be valid for non-uniform distributions. It should be noted that several techniques outlined in Chapter 1 are applicable to uniform distributions and can be used to solve the N-body problem in $O(N \log N)$ time per iteration. The hierarchical methods are designed to be efficient for arbitrary distributions of the particles. It is therefore important to analyze the complexity of these algorithms for arbitrary distributions.

Salmon [31] studies the Barnes-Hut algorithm in great detail. He shows that the Barnes-Hut algorithm takes $O(N^2)$ time for an exponential distribution. However, this does not represent the worst-case for the Barnes-Hut algorithm. Since the BH tree is unbounded, the Barnes-Hut algorithm is unbounded for arbitrary distributions.

Clearly, not all particle distributions can be modeled on a given machine due to precision limits. But, an algorithm whose running time depends upon the distribution is undesirable. An analogy can be drawn to a sorting algorithm whose running time depends on the size of the numbers to be sorted. The complexity of a sorting algorithm is $O(n \log n)$, provided basic operations on the numbers to be sorted (like comparison, copying) can be accomplished in constant time. The complexity of the algorithm does not remain $O(n \log n)$ if this assumption is not valid. However, there is a distinct advantage to having a sorting algorithm in which the number of operations is independent of the size of the input numbers. Such an algorithm can sort 128-bit numbers on a machine with 128-bit words with the same speed as it sorts 32-bit numbers on a 32-bit word machine.

Similarly, it is reasonable to assume that the distribution of the particles is representable in a given machine but algorithms whose running times depend on the distribution are undesirable.

CHAPTER 3. A DISTRIBUTION-INDEPENDENT DATA STRUCTURE FOR THE N-BODY PROBLEM

The distribution-dependency of the Barnes-Hut and Greengard's algorithms is due to the clustering scheme in which the subclusters are fixed relative to the parent cluster and irrespective of the location of the particles. In this chapter, we describe a distribution-independent clustering scheme for the N-body problem. The resulting data structure is presented as a modification to the BH tree to remove its distribution-dependency. We show that the Barnes-Hut and Greengard's algorithms can be run in conjunction with this modified tree structure. We prove that the force computations of the Barnes-Hut and Greengard's algorithms can be accomplished by a traversal the modified tree. The construction of the modified tree itself is postponed until the next chapter.

A Modified Data Structure

The BH tree can contain a path on which every node represents the same set of particles, though each node represents a cell of a different size. Such a path can be arbitrarily large irrespective of the total number of particles. Each node on the path represents a cell of exactly half the length of the cell represented by its parent. Our intent is to rectify this unbounded nature of the BH tree.

Let $v_1, v_2, ..., v_k$ $(k \geq 2)$ be a maximal path in the BH tree such that each node of the path represents the same set of particles. The maximality of the path ensures that v_1's parent has more particles than v_1 and no child of v_k has the same particles as v_k. Since only cells having more than one particle are subdivided, it is imperative that v_k is not a leaf and has at least two child nodes. If v_k is a leaf, v_k and hence v_1 have exactly one particle. In such a case, v_1 is not further subdivided and is a leaf, a contradiction. We can also assume without loss of generality that v_1 has a parent. Otherwise, v_1 has to be the root of the tree, thus containing all the particles in the system. By the property of the path $v_1, v_2, ..., v_k$, v_k also contains all the particles in the system. This simply means that our choice of the initial cell is too large for the system of particles and a cell $\frac{1}{2^k}^{th}$ length of it (this is the cell represented by v_k) can contain the entire system. In this case, we can safely make the subtree rooted at v_k to be the BH tree. Therefore, it can be assumed that v_1 always has a parent. Furthermore, v_i is the only child of v_{i-1} $(1 < i \leq k)$.

We define the modified tree as follows: Let $v_1, v_2, ..., v_k$ $(k \geq 2)$ be any maximal path in the BH tree as described above. Let v_0 be the parent of v_1. The modified tree is obtained by deleting the nodes $v_1, v_2, ..., v_{k-1}$ and making v_k the child of v_0. Since v_i is the only child of v_{i-1} $(1 < i \leq k)$, the resulting structure is a tree. The BH tree for a collection of 5 particles and the corresponding modified tree are shown in Figure 3.1.

The modified tree is obtained from the BH tree by collapsing paths representing the same particles using cells of different sizes, into a single node. Nodes in the BH tree are used to store aggregate information on the collection of particles they represent. For example, the Barnes-Hut method keeps track of the total mass and

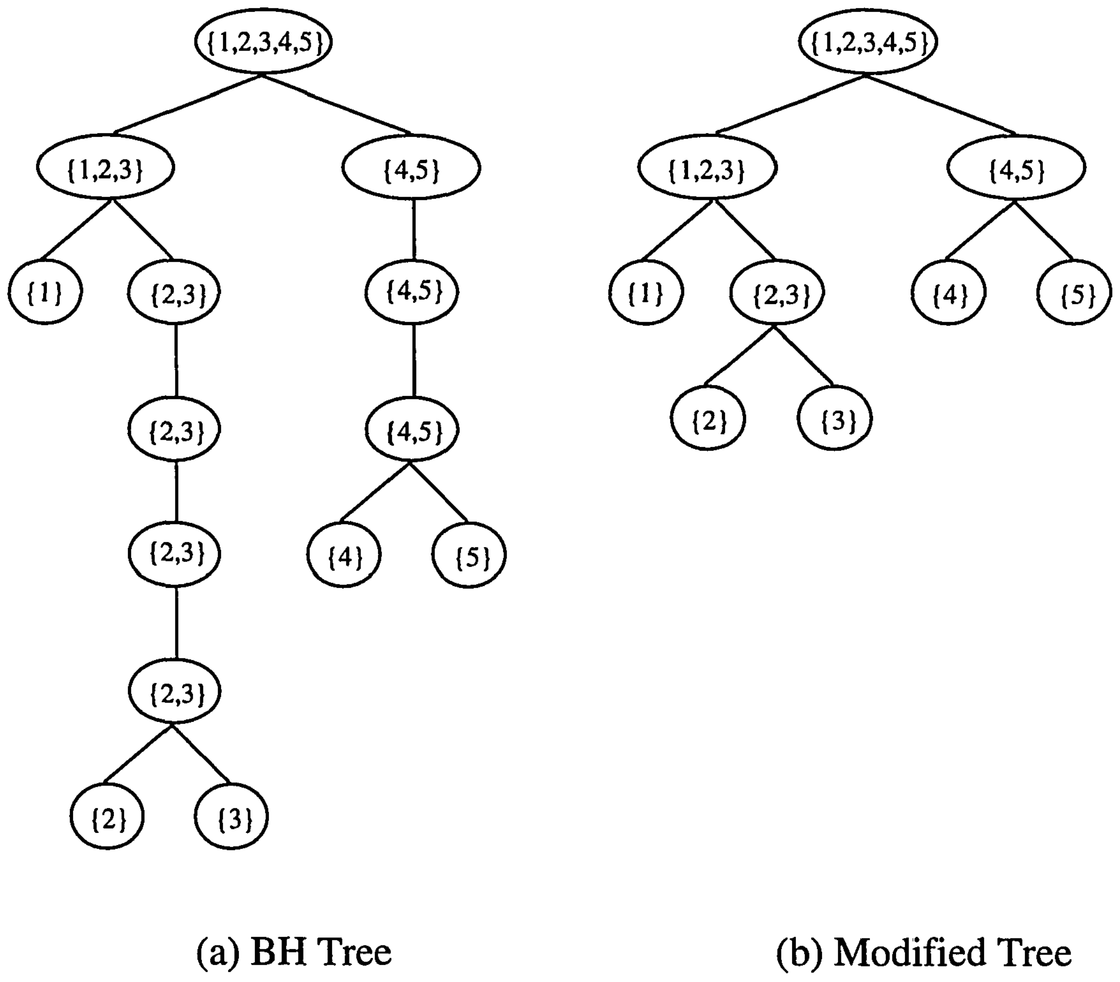

Figure 3.1: BH tree for a collection of 5 particles and the corresponding modified tree

the center of mass of the collection of particles. Greengard's method computes the multipole and local expansions of the collection of particles. Since every node on such a path represents the same particles, they all contain the same information, perhaps in a different form. Therefore, the modified tree obtained by eliminating this redundancy should contain the same information as the BH tree and it should be possible to modify Barnes-Hut and Greengard's algorithms to run on the modified tree.

For convenience of understanding, the tree is presented as a modification to the BH tree, obtained by collapsing paths representing the same particles. This should be taken as a definition of the modified tree rather than as a way of computing the modified tree. Since the BH tree is unbounded, one should not build the modified tree by first building the BH tree and deriving the modified tree from it. Algorithms for creating the modified tree directly in two and three dimensions are discussed in the following chapters.

Analysis of the Modified Tree

In this section, we show that the size of the modified tree is $O(N)$, irrespective of the distribution of the particles and the number of dimensions.

Lemma 3.1 *Let $S(N)$ be the number of nodes in the modified tree for N particles. $S(N) \leq 2N - 1$.*

Proof: By induction on the number of particles N. If $N = 1$, the modified tree is a single node representing a cell containing the particle. $S(1) = 1$, clearly satisfying the lemma.

Consider any $N > 1$. The root of the modified tree represents all the N particles. The root has at least 2 and at most 2^d children (where d is the number of dimensions). Let k be the number of children of the root node and let N_i be the number of particles contained in the cell represented by the i^{th} child. Let $S(N_i)$ be the size of the subtree rooted at the i^{th} child. We have

$$S(N) = 1 + \sum_{i=1}^{k} S(N_i) \qquad (2 \leq k \leq 2^d)$$

$$\sum_{i=1}^{k} N_i = N$$

By induction,

$$S(N_i) \leq 2N_i - 1$$

Therefore,

$$\begin{aligned} S(N) &= 1 + \textstyle\sum_{i=1}^{k} S(N_i) \\ &\leq 1 + \textstyle\sum_{i=1}^{k} (2N_i - 1) \\ &= 1 + \left(2 \textstyle\sum_{i=1}^{k} N_i\right) - k \\ &= 2N - (k-1) \\ &\leq 2N - 1 \end{aligned}$$

■

By the lemma, the size of the tree is bounded by $O(N)$ for any dimension d. Since any tree containing N leaves has at least $O(N)$ nodes, the modified tree is an optimal representation of the hierarchical clustering scheme. Since each child contains at least one particle less than its parent, the worst-case path length is also bounded by $O(N)$.

We now show that the force computation phase of the Barnes-Hut and Greengard's algorithms can be accomplished by a traversal of the modified tree instead of a traversal of the BH tree. It is assumed that the tree is already built.

The Barnes-Hut Method Using the Modified Tree

In the Barnes-Hut method, the BH tree is traversed once for every particle in the system to approximate the force acting on the particle due to the rest of the system. The force on any particle P is approximated using the following recursive calculation: Let l be the length of the cell currently being processed. Let d be the distance between the particle and the center of mass of the cell under consideration. If $\frac{l}{d} < \theta$, where $0 \leq \theta < 1$ is a pre-specified accuracy criterion, the cell is treated as a single particle of equivalent mass located at the center of mass for the purpose of force calculation. Otherwise, the children of the cell are examined recursively to compute the force on P. The force on P due to the particles in the cell is obtained by a vector summation of the forces on P due to the particles in each of the child cells. The force calculation starts by examining the *root cell.* This calculation is repeated once for every particle in the system.

Force Calculation

We show that performing force calculations on the modified tree yields exactly the same results as the force computations on the BH tree.

Theorem 3.2 *Let P be any particle. The approximate force acting on P as computed by a traversal of the modified tree is the same as the force computed by a traversal of the corresponding BH tree.*

Proof: Consider any maximal path $v_1, v_2, ..., v_k$ ($k \geq 2$) in the BH tree where all nodes represent the same particles and let v_0 be the parent of v_1. In the modified tree, v_k is the child of v_0. If v_1 is never reached (for any such maximal path) in the traversal of the BH tree, the force computation gives the same answer on either tree because the same nodes are traversed. Therefore, suppose that v_1 is reached during the traversal of the BH tree. Let $l(v_i)$ be the length of the cell, $cm(v_i)$ be the center of mass and $M(v_i)$ be the total mass of the particles in the cell represented by node v_i. Note that

$$M(v_1) = M(v_2) = ... = M(v_k)$$

$$cm(v_1) = cm(v_2) = ... = cm(v_k)$$

$$l(v_1) = 2l(v_2) = 2^2 l(v_3) = ... = 2^{k-1} l(v_k)$$

Case I: The traversal stopped at some v_i $(1 \leq i \leq k)$ in the BH tree.

Since the traversal stopped at v_i,

$$\frac{l(v_i)}{d(p, cm(v_i))} < \theta,$$

where $d(p, cm(v_i))$ is the distance from P to the center of mass of the cell represented by v_i and θ is the accuracy criterion. Since v_j is the only child of v_{j-1} $(1 < j \leq k)$, the force contributed by the subtree rooted at v_1 is the force between P and a mass of $M(v_i)$ located at $cm(v_i)$, given by

$$\frac{Gm_p M(v_i)}{d(p, cm(v_i))^2}.$$

In traversing the modified tree, v_k is reached instead of v_1. Since $k \geq i$,

$$\frac{l(v_k)}{d(p, cm(v_k))} = \frac{1}{2^{k-i}} \frac{l(v_i)}{d(p, cm(v_i))} < \theta.$$

The traversal stops at v_k and the force is computed to be

$$\frac{Gm_p M(v_k)}{d(p, cm(v_k))^2} = \frac{Gm_p M(v_i)}{d(p, cm(v_i))^2}.$$

The force contributed by the subtree rooted at v_k is the same as the force contributed by the subtree under v_1 in the BH tree, as needed.

<u>Case II:</u> The traversal proceeds to the children of v_k in the BH tree.

In this case, the traversal proceeds to the children of v_k in the modified tree also. The force contributed by the subtree rooted at v_1 in the Barnes-Hut tree is the force contributed by the subtree rooted at v_k, which is the same for both the trees.

Hence, the force computations give the same result on both trees. ■

The worst-case time to compute the force on a particle P using the BH tree is unbounded since the BH tree is unbounded. On the modified tree, this force computation is bounded by $O(N)$, the size of the modified tree.

Error Estimation

The error in approximating the force between a particle P and a cluster of particles by treating the cluster as a single particle of equivalent mass located at the center of mass is proportional to $\left(\frac{dr}{r}\right)^2$, where dr is the radius of the cluster and r is the distance of its center of mass from P. In the Barnes-Hut algorithm, each cell represents a cluster of particles. If l is the length of the cell containing the particles, the radius of this cluster of particles is at most $l\sqrt{d}$ where d is the number of dimensions. In two or three dimensions, the error introduced by treating the cell represented by node v_i as a single particle is therefore proportional to

$$\left(\frac{l(v_i)}{d(p, cm(v_i))}\right)^2.$$

If $v_1, v_2, ..., v_k$ is a maximal path in the Barnes-Hut tree with every node containing the same particles and the Barnes-Hut tree traversal stopped at some v_i $(1 \leq i \leq k)$, the error made is computed to be proportional to

$$\left(\frac{l(v_i)}{d(p, cm(v_i))}\right)^2 .$$

This is an overestimation of the error because the length of the cell that can contain the particles is taken to be $l(v_i)$ whereas the length is in fact bounded by $l(v_k) = \frac{l(v_i)}{2^{k-i}}$ $(k \geq i)$. A traversal on the modified tree computes the same force with an error estimate proportional to

$$\left(\frac{l(v_k)}{d(p, cm(v_k))}\right)^2 = \frac{1}{2^{2(k-i)}} \left(\frac{l(v_i)}{d(p, cm(v_i))}\right)^2 .$$

The error estimate at this node is thus improved by a factor of $2^{2(k-i)}$.

Greengard's Method Using the Modified Tree

Greengard's fast multipole algorithm is a two-pass procedure on the BH tree. The first pass is a bottom-up traversal of the tree in which a p-term multipole expansion is formed at every node of the tree, where p is a precision parameter. The multipole expansions at the leaves are computed directly. At any internal node, the multipole expansion is formed by shifting the multipole expansions of the child nodes to the center of the cell represented by the node and adding them together. In the second pass, the tree is traversed top-down to compute the local expansions at every node. The local expansion at a node is formed by shifting the local expansion at the parent node to its center, shifting the multipole expansions of the *well-separated* children of the nearest neighbors of the parent of the node to its center and adding them together. Finally, the local expansions at every leaf are evaluated to compute the approximate cumulative force on each particle. For a detailed description of Greengard's algorithm, see [14].

Consider a run of Greengard's algorithm on the BH tree containing a path $v_1, v_2, ..., v_k$, where each node represents the same particles. Since v_i is the only child of v_{i-1} $(1 < i \leq k)$, the multipole expansion at v_{i-1} is formed by shifting the multipole expansion of v_i to the center of the cell represented by v_{i-1}. The multipole expansions at these nodes are merely translations of one another. Since $v_1, v_2, ..., v_k$ is a chain, the multipole expansions at these nodes are useful only to compute the multipole expansion of v_1's parent. However, the contribution by v_1's multipole expansion to the multipole expansion of its parent can be directly obtained by shifting the multipole expansion of v_k to the center of the cell represented by the parent of v_1. Thus, computing the multipole expansions at $v_1, v_2, ..., v_{k-1}$ is unnecessary and is avoided by the modified tree. A similar argument shows that the correct local expansions at the leaves can be obtained using the modified tree.

In the multipole algorithm designed to run on the modified tree, the precision parameter p is a constant since it can be chosen independent of N. In Greengard's algorithm, p has a lower bound of $\log N$. This is because p is also used as an upper bound on the worst-case path length ($\log \frac{D}{s}$) of the BH tree, which has a lower bound of $\log N$. Therefore, p cannot be chosen independent of N and is also a function of the distribution of the particles. In the multipole algorithm on the modified tree, the precision parameter is merely a function of the desired accuracy of the force calculations chosen independent of the number and distribution of the particles.

The new algorithm consists of two traversals of the modified tree. Computing the p-term multipole/local expansion at each node takes constant time per node. Evaluating a p-term local expansion for every particle also takes constant time. Since the number of nodes in the modified tree is $O(N)$, running the multipole algorithm once the modified tree is constructed takes $O(N)$ time. This is irrespective of the distribution of the particles.

The running time of this algorithm depends on the complexity of the tree creation and the complexity of performing the force calculations. It is already noted that the force computations can be performed in $O(N)$ time on the modified tree. In the next section, we show that the modified tree can be created in $O(N \log N)$ time in two dimensions and in $O(N \log^2 N)$ time in three dimensions. Thus, the new multipole algorithm has a complexity of $O(N \log N)$ in two dimensions and $O(N \log^2 N)$ in three dimensions.

CHAPTER 4. TWO-DIMENSIONAL ALGORITHMS

In this chapter, we discuss algorithms for creating the modified tree in two dimensions. First, we show that the construction of the tree requires $\Omega(N \log N)$ time. Then, we present an algorithm with running time matching this lower bound.

A Lower Bound for the Construction of the Modified Tree

We show that constructing the modified tree requires $\Omega(N \log N)$ time by reducing sorting to the construction of the modified tree.

Let $x_1, x_2, ..., x_N$ be the input to the sorting problem. Without loss of generality, assume that $x_i \geq 0$ $(1 \leq i \leq N)$. Otherwise, let $x_{min} = min\{x_1, x_2, ..., x_N\}$ and create a new sequence $x'_1, x'_2, ..., x'_N$ where $x'_i = x_i - x_{min}$. Let $y'_1, y'_2, ..., y'_N$ be the output of sorting this sequence. The output of the original sorting problem is $y'_1 + x_{min}, y'_2 + x_{min}, ..., y'_N + x_{min}$. The extra effort required is linear and does not change the complexity of sorting since producing the output to the sorting problem takes at least linear time.

Assume that the input $x_1, x_2, ..., x_N$ to the sorting problem is non-negative. Position N particles such that the i^{th} particle is at location (x_i, x_i^2). The points lie on the parabola $y = x^2$ to the right side of y-axis. Construct the modified tree for this collection of N particles (see Figure 4.1). The output of the sorting problem can

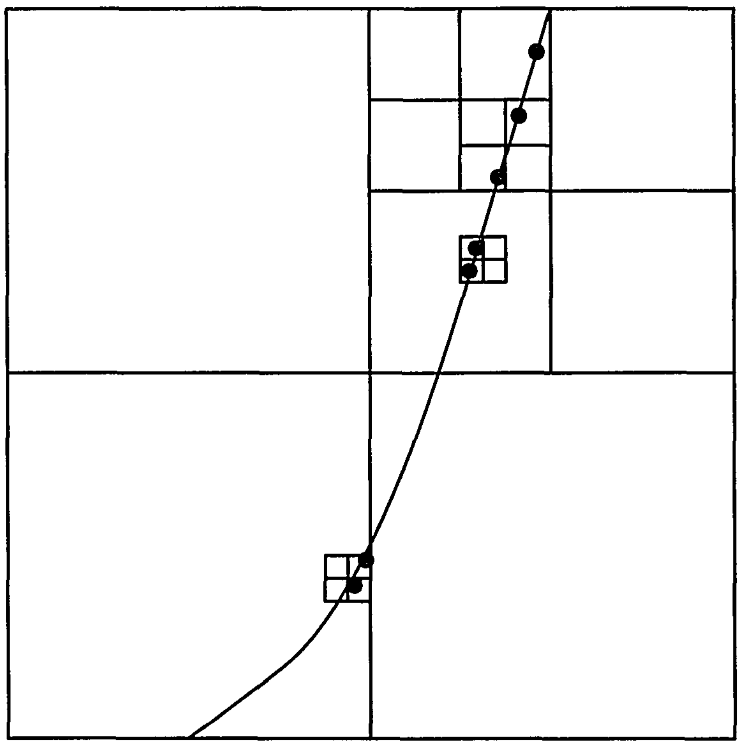

Figure 4.1: Reduction of the sorting problem to the construction of the modified tree in two dimensions

now be read off from the modified tree as follows: Let c be the cell represented by a node in the modified tree. Taking the center of this cell to be the origin, we can label the children of the node as I, II, III or IV according to the quadrant containing the subcell represented by the child. Let (x_i, x_i^2) be the position of any particle in the subtree of child III or child IV and let (x'_i, x'^2_i) be the position of any particle in the subtree of child I or child II. $x_i^2 \leq x'^2_i$ and hence $x_i \leq x'_i$. Therefore, points in the subtrees of child III and child IV appear before the points in the subtrees of child I and child II in sorted order. Any point in the subtree of child II (child III) has a smaller x coordinate than any point in the subtree of child I (child IV). Thus, the sorted order can be read off from the modified tree by starting at the *root cell* and recursively enumerating the particles in subcells represented by the children labeled III, IV, II and I in that order.

Constructing the input to the tree construction problem from the input of the sorting problem requires $O(N)$ time. The sorted order can be read off from the tree in $O(N)$ time since the tree contains $O(N)$ nodes and each node is traversed exactly once. Therefore, a lower bound of $\Omega(N \log N)$ for sorting implies the same lower bound for the construction of the tree.

The lower bound also applies to the construction of the BH tree. The same reduction can be used but the cost of reading the sorted order from the BH tree cannot be bound since the the number of nodes in the BH tree is not bounded. If the BH tree has $O(N \log N)$ or more nodes, the construction of the tree clearly takes $\Omega(N \log N)$ time. Otherwise, a lower bound of $\Omega(N \log N)$ for sorting implies the same lower bound for BH tree construction. This should be contrasted with Greengard's estimation that the BH tree can be constructed in $O(N)$ time.

We now present an algorithm to construct the modified tree in $O(N \log N)$ time for a collection of N particles in two dimensions.

Notation

The physical space containing the particles is subdivided into cells. The cells represent square regions of space in two dimensions. A cell is completely determined by the length of an edge of the cell and the position of one of the corners of the cell. Without loss of generality, choose the lower, leftmost corner. Let D be the length of the *root cell.* We also use the term *cell* to refer to the node in the modified tree representing the cell, for convenience.

Let l be any cell. In order to describe the subcells of this cell, choose the corner of the cell to be the origin. The cell contains 2^{2k} cells of length $\frac{l}{2^k}$. The cells are positioned at $(i\frac{l}{2^k}, j\frac{l}{2^k})$ $(0 \leq i, j < 2^k)$.

Definition 4.1 *A line is called a k-boundary if it contains an edge of a cell of length* $\frac{l}{2^k}$.

Any *boundary* is parallel to one of the axes. A *boundary* can be specified by the axis to which it is parallel and the distance of the *boundary* from the axis. A *k-boundary* is at a distance of $i\frac{l}{2^k}$ $(0 \leq i \leq 2^k)$ from the axis parallel to it.

Fact 4.2 *Any k-boundary is also a j-boundary for every* $j > k$.

There are $2^k + 1$ lines parallel to each axis and spaced $\frac{l}{2^k}$ apart that are *k-boundaries*. The intersections of the *k-boundaries* determine the cells of size $\frac{l}{2^k}$. Subcells and *boundaries* of the *root cell* are shown in Figure 4.2. Note that the description of the subcells and the boundaries is relative to the cell.

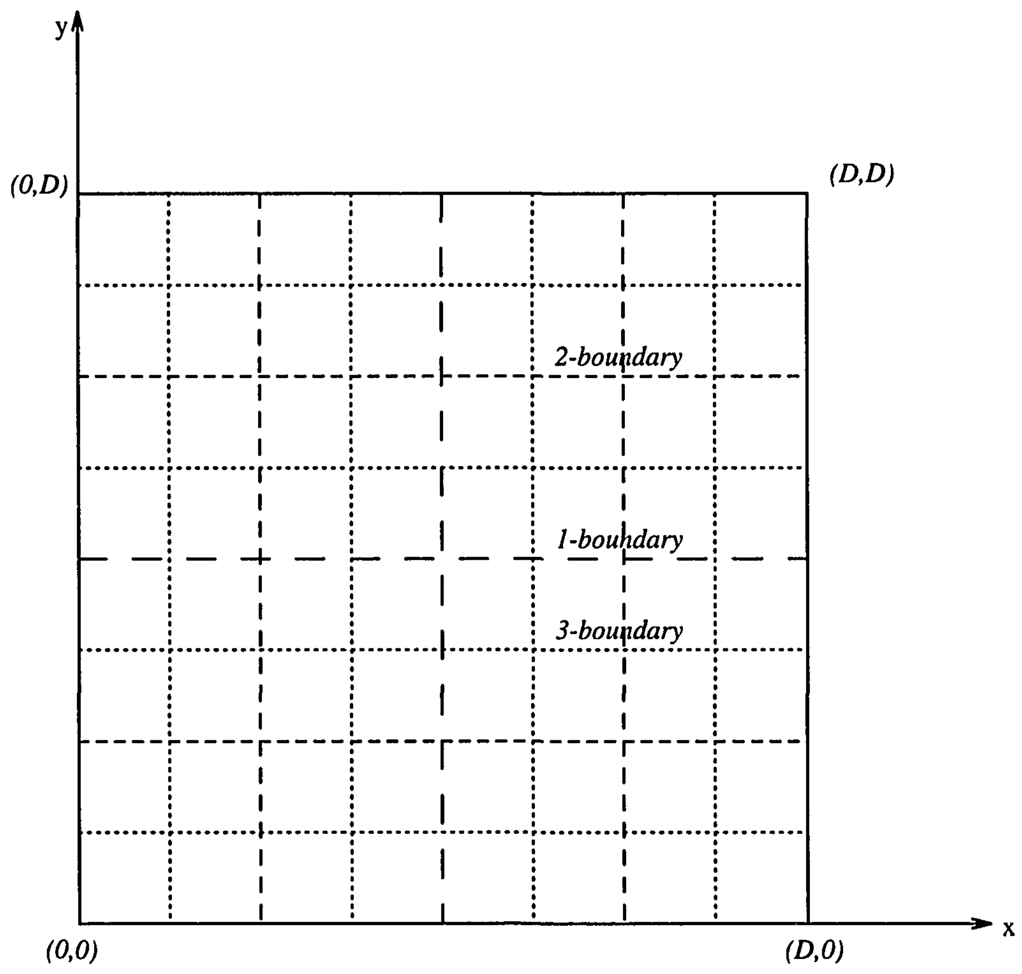

Figure 4.2: A *root cell* of length D. The big dashed lines are 1-*boundaries*, the small dashed lines are 2-*boundaries* and the dotted lines are 3-*boundaries*. 2-*boundaries* are also 3-*boundaries* and 1-*boundaries* are also 2-*boundaries* and 3-*boundaries*

Algorithm for Constructing the Modified Tree

A simple recursive algorithm for creating the modified tree for a cell c containing a collection of particles can be informally stated as follows:

BuildTree(c)

1. Find the smallest cell c' contained in c that still contains all the particles contained in cell c.
2. If c contains no particles, return '*empty tree*'.
3. If c contains exactly one particle, return the one node tree c.
4. Split the cell c' into 4 subcells.
5. For each subcell sc of c', *BuildTree(sc)*.
6. Return the tree obtained by joining all the trees obtained in the previous step, with c' as the root of the tree.

BuildTree is initially called with a cell large enough to contain all the particles in the system. The description of the positions of the particles contained in cell c is not passed as input to the function BuildTree. Otherwise, calling the function on each of the subcells would require distributing the particles among the child cells resulting in $O(N)$ work at every level of the tree. Since there can be $O(N)$ levels, such a distribution itself would require $O(N^2)$ work.

The input to BuildTree is just a description of the cell c - the length of c and the position of its lower, leftmost corner. The running time of the algorithm can be computed by the amount of work done at every node of the modified tree, which is steps 1-4 and 6. Steps 4 and 6 require a constant amount of work at every node of the modified tree. Determining if the input cell c does not contain any particles or if it contains exactly one particle can be determined as a byproduct of Step 1, as we shall see later. Step 1 can be accomplished as follows:

Let l be the length of the cell c passed as input to *BuildTree*. Any cell smaller than c but contained in c has length $\frac{l}{2^k}$ for some $k > 0$. By a suitable transformation, the corner of c is chosen to be the origin. Let b be the smallest rectangle containing all the particles in c. The rectangle is specified by $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$, where x_{min} is the smallest x coordinate of all the particles in c etc. The smallest cell in c containing all the particles of c should also contain the box b.

Fact 4.3 *A cell of size $\frac{l}{2^k}$ encloses b iff no k-boundary passes through b.*

The smallest subcell of c enclosing b is of size $\frac{l}{2^{k-1}}$, where k is the smallest integer such that a *k-boundary* passes through b (Figure 4.3). To determine this, we can examine boundaries parallel to each coordinate axis in turn.

Consider boundaries parallel to the y-axis. These can be specified by their distance from the y-axis. The family of *k-boundaries* is specified by $i\frac{l}{2^k}$, $0 \leq i \leq 2^k$. Let k be the smallest integer such that a *k-boundary* parallel to y-axis passes through b, i.e. k is the smallest integer such that $x_{min} < i\frac{l}{2^k} < x_{max}$ for some i.

Lemma 4.4 *Exactly one k-boundary passes through b.*

Proof: Suppose not. Consider any two consecutive *k-boundaries* that pass through b. These are given by $i\frac{l}{2^k}$, $(i+1)\frac{l}{2^k}$ for some i. Let i' be the even integer among i and $i+1$. Let $i'' = \frac{i'}{2}$. One of the *k-boundaries* is specified by $i'\frac{l}{2^k} = i''\frac{l}{2^{k-1}}$. Therefore,

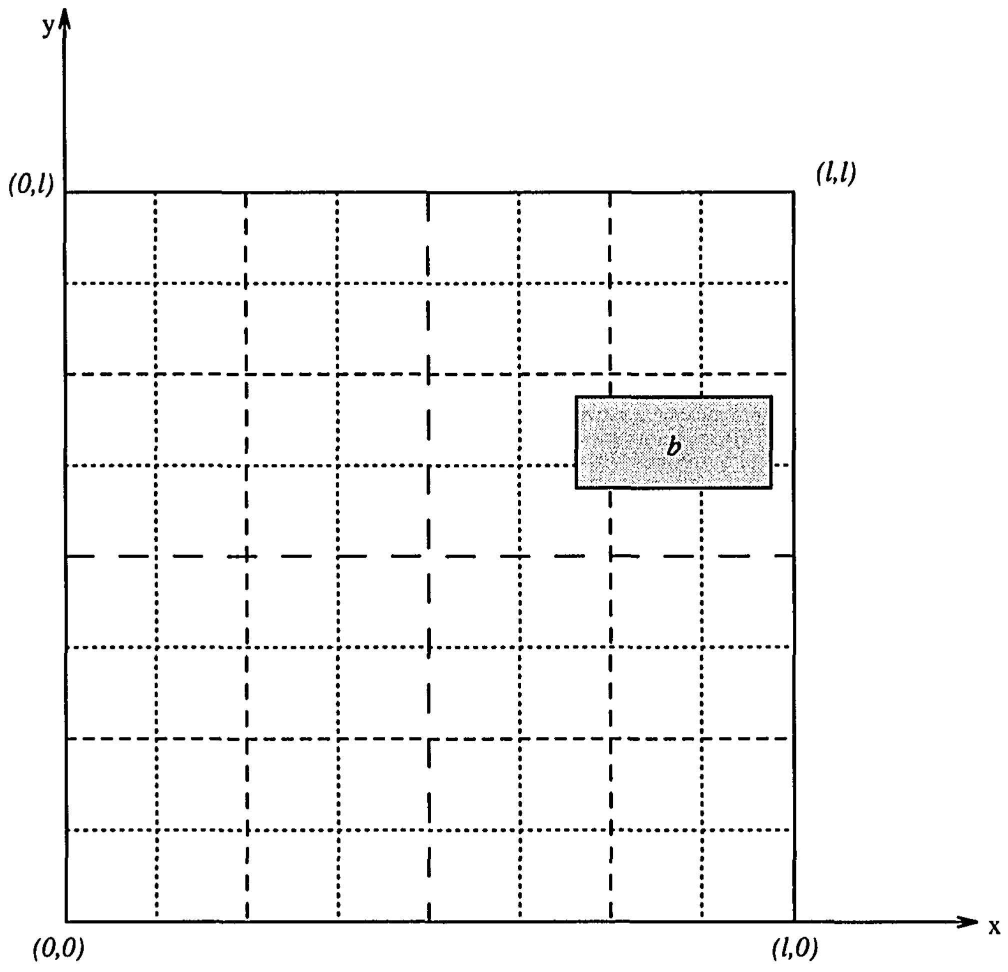

Figure 4.3: A cell of length l and the smallest box b enclosing all the particles in this cell

this is also a $(k-1)$-*boundary* and a $(k-1)$-*boundary* passes through b, contradicting the minimality of k. ■

Let j be the smallest integer such that $\frac{l}{2^j} < (x_{max} - x_{min})$, i.e., set $j = \lceil \log_2 \frac{l}{x_{max}-x_{min}} \rceil$.

Lemma 4.5 *There is at least* 1 *and at most* 2 *j-boundaries passing through b.*

Proof: Suppose that no *j-boundary* passes through b. Since the distance between two consecutive *j-boundaries* is $\frac{l}{2^j}$, this would require $\frac{l}{2^j} > (x_{max} - x_{min})$, a contradiction. Therefore, at least one *j-boundary* crosses b.

If more then 2 *j-boundaries* cross b, let $i\frac{l}{2^j}$, $(i+1)\frac{l}{2^j}$ and $(i+2)\frac{l}{2^j}$ be three consecutive *j-boundaries* passing through b. We have $i\frac{l}{2^j} > x_{min}$ and $(i+2)\frac{l}{2^j} < x_{max}$. Therefore, $(x_{max} - x_{min}) > (i+2)\frac{l}{2^j} - i\frac{l}{2^j} = 2\frac{l}{2^j} = \frac{l}{2^{j-1}}$, contradicting the minimality of j. ■

The *j-boundaries* passing through b are specified by $h_1 = \lceil \frac{2^j x_{min}}{l} \rceil \frac{l}{2^j}$ and $h_2 = \lfloor \frac{2^j x_{max}}{l} \rfloor \frac{l}{2^j}$. If $h_2 = h_1$, only one j-boundary passes through b. Otherwise, two j-boundaries pass through b. Let a be $\lceil \frac{2^j x_{min}}{l} \rceil$. $h_1 = a\frac{l}{2^j}$ and $h_2 = h_1$ or $(a+1)\frac{l}{2^j}$.

Lemma 4.6 *The k-boundary passing through b is either h_1 or h_2.*

Proof: Suppose not. Since $k \leq j$, by Fact 4.2, the *k-boundary* passing through b is also a *j-boundary*. By Lemma 4.2, h_1 and h_2 are the only *j-boundaries* passing through b. Therefore, the *k-boundary* passing through b must coincide with h_1 or h_2. ■

It is now easy to find k since the *k-boundary* passing through b is narrowed down to either h_1 or h_2. If $h_2 \neq h_1$, let a' be the even integer among a and $a+1$. Otherwise, let a' be equal to a. It is clear that $j - k$ is equal to the highest power of 2 that divides a'. One way to find this is to set $j - k = \log_2(1 + \{a' \oplus (a' - 1)\}) - 1$. Since all the above operations take constant time, the smallest cell contained in c enclosing the box b can be determined in constant time.

It is already established that the modified tree has $O(N)$ nodes. The tree is created top-down starting at the root. At each node, the particles with the smallest and the largest coordinates in each dimension ($x_{min}, x_{max}, y_{min}$ and y_{max} in two dimensions) are computed to identify the smallest box enclosing all the particles represented by the node. The smallest cell enclosing this box is computed and the children of the node determined in constant time. As mentioned before, the particles are not distributed among the child nodes. Such a distribution would result in $O(N^2)$ time for tree creation. Distributing the particles to the child nodes is not necessary provided we can determine the particles with extreme coordinates in the child nodes. Except for this task, the rest of the computations are done in constant time per node, for a total of $O(N)$ time.

Finding the points with extreme coordinates can be translated to a range query problem, stated as follows: Given N points, set up a data structure to answer queries of the form 'which point has the smallest x-coordinate among the points that lie in a given square?' efficiently. The answer to such a query is the point with the smallest x-coordinate or $<$ *none* $>$ if no points exist in the given square. Since the modified tree has $O(N)$ nodes and we require four such queries per node, the number of queries is $O(N)$.

In BuildTree, we also need to determine cases where the cell contains exactly one particle or none. This can be determined as a byproduct of the computation of the smallest box b containing all the particles in the cell. If $x_{min} = x_{max}$ and $y_{min} = y_{max}$, the cell contains exactly one particle. If the answer to any of the 4 queries is $< none >$, the cell is empty and can be discarded.

The time for constructing the tree is $O(N)$ plus the time to set up data structures for range querying and the time to perform $O(N)$ queries. In the next section, we discuss a solution to the range query problem.

Range Queries in Two Dimensions

In this section, we discuss the problem of setting up a data structure to answer range queries. The problem we are interested in is formally described as: Given N points (x_i, y_i) $(1 \leq i \leq N)$, set up a data structure to perform queries of the form 'find the minimum (maximum) x-coordinate x_{min} (x_{max}) of all the points in a given query rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$' efficiently. We first describe a simple solution that requires $O(N \log N)$ set up time and $O(log^2 N)$ time per query. We then describe a solution based on Priority Search Trees [28] to reduce the query time to $O(\log N)$ per query.

A Simple Algorithm

Consider the problem of finding a point with the smallest (largest) x-coordinate in a given query rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$. If we gather all the points in the range $[y'_0, y'_1]$, the y direction becomes irrelevant and the query can be answered by a one-dimensional query on x on these points. If a data structure is designed such that these points are available sorted by their x-coordinate, we can find x_{min} (x_{max}) easily with a binary search.

The data structure is built as follows: Sort the given N points by their y-coordinates. Build a Balanced Binary Search Tree (BBST) with the points as leaves. The internal nodes in this BBST do not correspond to any points. They represent the range of values of y-coordinates of the points in their subtrees. The range assigned to a node is $[y_l, y_u]$ where y_l is the smallest y coordinate and y_u is the largest y coordinate of all the points in the subtree rooted at the node. y_l and y_u correspond to the leftmost and rightmost leaves of the subtree rooted at the node. If the range at the left child of a node is $[y_{l1}, y_{u1}]$ and the range at its right child is $[y_{l2}, y_{u2}]$, the range assigned to the node is $[y_{l1}, y_{u2}]$ (see Figure 4.4).

At every node of the BBST, store a list of all the points in its subtree sorted by the x-coordinate. These lists are easily constructed in a bottom-up traversal of the tree. The sorted list at a node is constructed by merging the sorted lists at its left and right children. The data structure for a collection of 8 particles is shown in Figure 4.4.

Sorting the points according to their y-coordinate requires $O(N \log N)$ time. The BBST over these points is built in linear time. Creating the sorted lists at every node requires $O(N)$ work per level of the tree (since the lists are formed by merging and the total number of points merged at any level of the tree is N), for a total of $O(N \log N)$ work. In fact, the sorted lists at the nodes represent the intermediate lists produced during a merge sort of the points. The time and space requirements for building this data structure are $O(N \log N)$.

Given a query rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$, a list of nodes in the BBST are identified that cover the range $[y'_0, y'_1]$ exactly. The minimum (maximum) x-coordinate in the range $[x'_0, x'_1]$ at each of these nodes is identified using binary search. The minimum

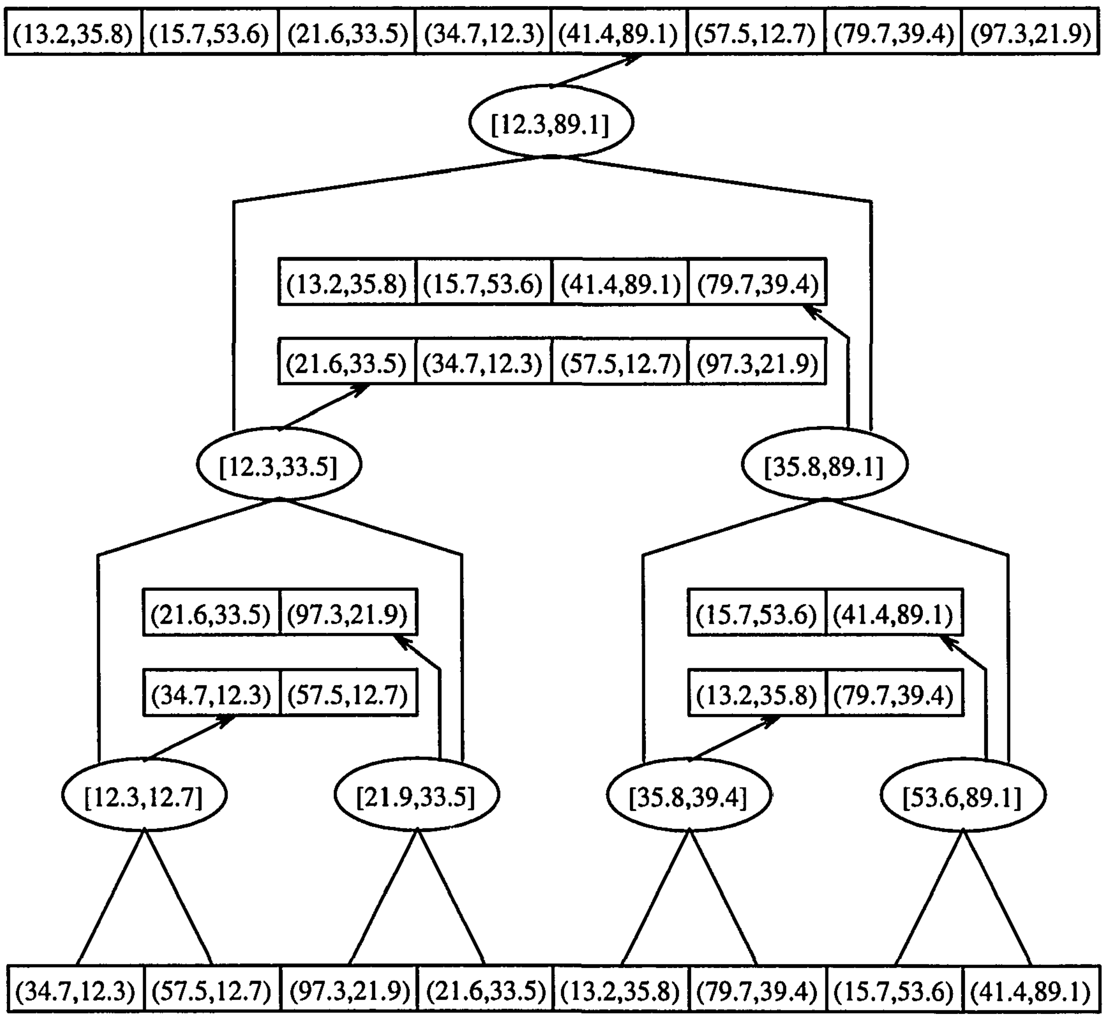

Figure 4.4: A simple data structure for performing range queries in two dimensions

(maximum) of all the values obtained is x_{min} (x_{max}).

The following algorithm identifies a list of nodes in the BBST that cover the range $[y_0', y_1']$.

```
FindNodes(node)
    If $[y_0', y_1']$ does not contain the range at the node,
        If range at left child intersects $[y_0', y_1']$, FindNodes(leftchild(node)).
        If range at right child intersects $[y_0', y_1']$, FindNodes(rightchild(node)).
    Else add node to the list.
```

FindNodes is initially called with the *root* of the BBST. *FindNodes* traverses a subtree of the BBST and its running time is proportional to the size of this subtree. The list of nodes identified by this function exactly cover the range $[y_0', y_1']$ and are the leaves of the subtree traversed by this function.

Lemma 4.7 *The number of nodes in the list formed by FindNodes is at most* $O(\log N)$.

Proof: We first show that no more than 2 nodes are identified at any level of the tree. Suppose that this is not true. Let the *root* be at level 0 and suppose that three or more nodes are identified at level i ($i \geq 2$). The nodes identified at any level are clearly consecutive. Thus, $\exists$ a node v at level $i-1$ such that its left child $l(v)$ and its right child $r(v)$ are both identified by *FindNodes*. By the description of *FindNodes*, $[y_0', y_1']$ contains the range at $l(v)$ and $r(v)$. Since the range at v is the union of the ranges at $l(v)$ and $r(v)$, $[y_0', y_1']$ should contain the range at v also. But then, the node v is identified by *FindNodes* and the children of node v are not traversed at all, contradicting the assumption that $l(v)$ and $r(v)$ are identified.

Therefore, function *FindNodes* identifies at most 2 nodes at any level of the BBST. Since the number of levels is bounded by $O(\log N)$, the number of nodes in the list formed by *FindNodes* is at most $O(\log N)$. ■

Lemma 4.8 *The running time of FindNodes is $O(\log N)$.*

Proof: The running time of *FindNodes* is proportional to the size of the subtree it visits. By lemma 4.7, this subtree has $O(\log N)$ leaf nodes. The number of nodes in a binary tree with $O(\log N)$ leaf nodes is also $O(\log N)$. Thus, the running time of *FindNodes* is $O(\log N)$. ■

Function *FindNodes* identifies a set of $O(\log N)$ nodes that cover exactly all the points with y-coordinates in the range $[y'_0, y'_1]$. We can identify the minimum (maximum) x values in the range $[x'_0, x'_1]$ in the sorted lists at these nodes using a simple binary search. The minimum (maximum) of all these values gives x_{min} (x_{max}) in the query rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$. Since any of these sorted lists contains at most N points, each binary search takes at most $O(\log N)$ time. Performing $O(\log N)$ searches requires $O(\log^2 N)$ time, after which x_{min} (x_{max}) can be found by computing the minimum (maximum) of the results of these searches in $O(\log N)$ time. Thus, the query time is $O(\log^2 N)$.

To query for minimum (maximum) y-coordinate y_{min} (y_{max}), an analogous data structure can be designed which is a BBST on the x-coordinate with each node containing the points in its subtree sorted by their y-coordinates.

This scheme can be used recursively to create a data structure for answering range queries for any dimension d. First, one of the dimensions is chosen and a balanced binary search tree is built on the points sorted by the chosen dimension.

At each node, the data structure for performing $(d-1)$-dimensional range queries on the points in the subtree of the node is stored. To answer a query, a set of nodes in the balanced binary search tree are identified that cover the range along the dimension using which the balanced binary search tree is built. By Lemma 4.7, the number of nodes identified is $O(\log N)$. The query can be answered by performing $(d-1)$-dimensional queries at each of these $O(\log N)$ nodes and taking the minimum (maximum) value obtained. The query time is $O(\log^d N)$.

Using this scheme, the modified tree can be built in $O(N \log^2 N)$ time in two dimensions. This consists of $O(N \log N)$ set up time for creating the data structures to perform the required range queries, $O(N \log^2 N)$ time for $O(N)$ queries at $O(\log^2 N)$ time per query and $O(N)$ time for rest of the work in function BuildTree. We next outline a solution based on Priority Search Trees to reduce the query time to $O(\log N)$ with the same set up time, to reduce the total complexity to $O(N \log N)$.

Priority Search Trees

A Priority Search Tree (referred to as PST hereafter) [28] is a data structure for representing N points in two dimensions such that the following operations can be implemented efficiently.

- *InsertPoint(x, y)* : Insert the point (x, y) into the PST.
- *DeletePoint(x, y)* : Delete the point (x, y) from the PST.
- *MinXInRectangle(x_0', x_1', y_1')* : Find the point with the smallest x coordinate in the rectangle $[x_0', x_1'] \times (-\infty, y_1']$.

- *MaxXInRectangle*(x'_0, x'_1, y'_1) : Find the point with the largest x coordinate in the rectangle $[x'_0, x'_1] \times (-\infty, y'_1]$.
- *MinYInXrange*(x'_0, x'_1) : Find the point with the smallest y coordinate such that $x'_0 \leq x \leq x'_1$.
- *EnumerateRectangle*(x'_0, x'_1, y'_1) : Enumerate all the points in the rectangle $[x'_0, x'_1] \times (-\infty, y'_1]$.

McCreight [28] presents algorithms to perform all the above operations in $O(\log N)$ time except for *EnumerateRectangle*(x'_0, x'_1, y'_1) which requires time proportional to the number of points enumerated. For our purposes, we are interested in the operations *MinXInRectangle*(x'_0, x'_1, y'_1) and *MaxXInRectangle*(x'_0, x'_1, y'_1). Notice that the operations find the minimum (maximum) x in a rectangle with the bottom edge fixed at $-\infty$. We need to use the PST's to create a data structure that allows us to perform queries on bounded rectangles. Also, a PST can be designed such that the queries can be performed in a rectangle with the top edge fixed at $+\infty$.

The PST can best be described as a combination of a binary search tree on x and a priority queue on y. PST is a tree with each node containing two points p and q and two boolean variables *validP* and *duplQ*. The notation $p.x$ refers to the x-coordinate of the point p etc. $v.p$ is used to refer to the point p at node v. Thus, $v.p.x$ refers to the x-coordinate of point p at node v. The tree should satisfy the following properties (see Figure 4.5):

1. The tree is a binary search tree based on $q.x$. Each point appears in the q filed of exactly one node.

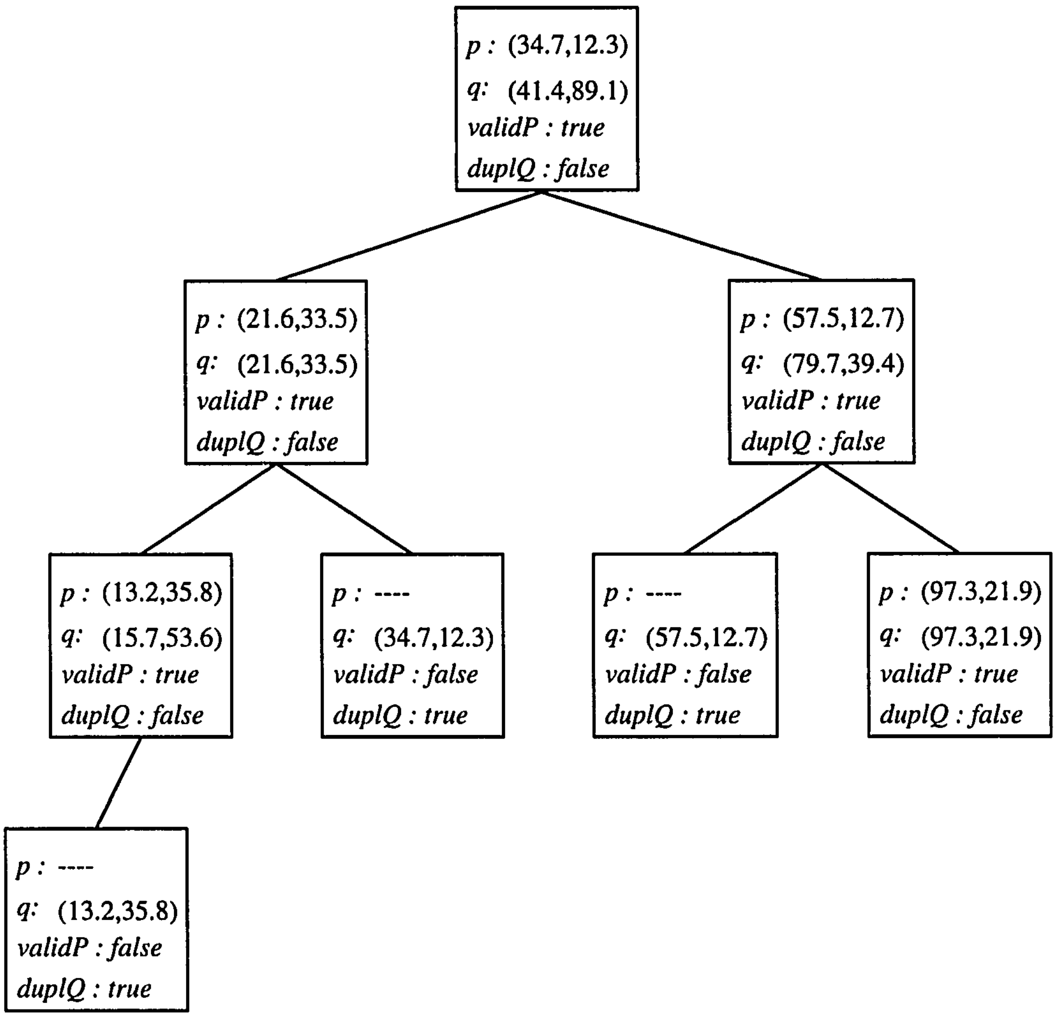

Figure 4.5: A Priority Search Tree for 8 points in two dimensions

2. The p field of each node contains the point with the smallest y coordinate among the points given by the q fields of the descendants of the node such that the point is not the one with the smallest y coordinate among the q fields of the descendants of the parent of the node. The p field is empty if no such node exists. The field *validP* indicates if the p field is valid or empty.

3. The field *duplQ* is true at a node if its q field is used as the p field of one of its ancestors.

4. The tree is a balanced binary tree (the height of the tree should be $O(\log N)$).

On a tree satisfying the above properties, the algorithms by McCreight can be used to perform the described operations in $O(\log N)$ time. To perform queries on rectangles of the form $[x'_0, x'_1] \times [y'_0, +\infty)$, the p fields should be used to store points with the largest y coordinates.

Creating a Priority Search Tree

A PST can be constructed by repeatedly inserting each point using the operation *InsertPoint(x, y)* starting with an empty tree. This would require $O(N \log N)$ time. However, if the points are already available sorted by their x-coordinate, the PST can be built in $O(N)$ time as described below.

First, use the sorted order on x to construct a Balanced Binary Search Tree (BBST) using the q fields. For any node v, $l(v)$ and $r(v)$ refer to the left and right child of the node v, respectively. The q field at any node in the BBST represents one of the input points. Let v be any node in the tree. The binary search tree property dictates that for any node v' in the left subtree of v, $v'.q.x \leq v.q.x$ and for any node

v' in the right subtree of v, $v'.q.x \geq v.q.x$. The BBST can be built in time linear in the number of points and it satisfies criteria 1 and 4 of the properties of a PST.

It only remains to fill in the appropriate p, *validP* and *duplQ* fields at every node to convert this BBST into a PST. At every node of the BBST, initialize the p field to be the same as the q field and set *validP* to *true* and *duplQ* to *false*. Procedure *MakePST* converts the subtree rooted at v into a PST. When *MakePST* is called, it is assumed that the subtrees rooted at $l(v)$ and $r(v)$ are already PSTs and that the subtree rooted at v is a BBST based on the x-coordinates of the q fields as described above.

```
MakePST(v)

1. Pick the point $p'$ with the smallest $y$-coordinate among $v.p$, $l(v).p$ and $r(v).p$.
   Since the $p$ field is valid only if validP is true, the $p$ fields of only such nodes
   are considered.

2. If validP is false at all three nodes, return.

3. If $p' = v.p$, return.

4. If $p' = l(v).p$, $v' \leftarrow l(v)$ else $v' \leftarrow r(v)$

5. (a) $v.p \leftarrow v'.p$
   (b) $v.validP \leftarrow true$
   (c) $v'.validP \leftarrow false$
   (d) If $v'.q.y < v'.p.y$, $v'.duplQ \leftarrow true$
   (e) MakePST(v')
```

When *MakePST(v)* is called, the p field of v may be borrowed from the p fields of one of the children of v having the smaller y coordinate. If v' is the child of v from which the p field is borrowed, the subtree rooted at v' is no longer a PST, but its left and right subtrees still are. We can recursively adjust the tree rooted at v' to be a PST. A call to *MakePST(v)* may involve traversing a path all the way down to the leaf in the subtree rooted at v. Thus, the running time of *MakePST* is $O(h)$ where h is the height of the subtree rooted at v.

MakePST adjusts the tree to be a PST when the left and right subtrees of the root are PSTs. To build a PST, we can build PSTs for the left and right subtrees of the *root* node and use *MakePST* to adjust the entire tree to be a PST.

```
BuildPST(v)
    1. BuildPST(l(v))
    2. BuildPST(r(v))
    3. MakePST(v)
```

BuildPST is initially called with the *root.* The work required can be described by the following recurrence:

$$T(N) = T\left(\lceil\frac{N}{2}\rceil\right) + T\left(\lfloor\frac{N}{2}\rfloor\right) + \lceil\log N\rceil$$

The solution to the above recurrence is $O(N)$. Thus, the PST can be built in $O(N)$ time.

Setting Up the Data Structure for Range Query

Sort the points (x_i, y_i) by increasing y and use this to build a balance binary search tree on y. This requires $O(N \log N)$ time. At each node v of the search tree, store PST(v), where PST(v) is the priority search tree for all the descendants of node v. In computing PST(v) we can assume that PST($l(v)$) and PST($r(v)$) are already computed. Note that PST(v) contains the union of the points in PST($l(v)$) and PST($r(v)$). By traversing PST($l(v)$), we can get the points in it in sorted order according to x-coordinate. A similar sorted sequence can be obtained by traversing PST($r(v)$). The two sequence can be merged in linear time to get a sorted order on x of the points forming PST(v). Using this, we can build PST(v) in time linear in the number of points in PST(v).

Two types of PST's are created depending on if the node at which the PST is created is the left or right child of its parent. If v is the left child of its parent, PST(v) is created such that the PST can be queried on rectangles with top edge at $+\infty$. If v is the right child of its parent, PST(v) is created such that the PST can be queried on rectangles with bottom edge at $-\infty$. No PST need be built at the root of the tree. The reason for this becomes clear later.

We can build the PST(v) in $O(m)$ time where m is the number of descendants of v. There can be at most 2^i nodes at level i of the tree each of which may contain $2^{\lfloor \log N \rfloor - i + 1}$ nodes. Therefore, the time required to create all the PST's is

$$\sum_{i=1}^{\lfloor \log N \rfloor} 2^i (2^{\lfloor \log N \rfloor - i + 1})$$

$$= \sum_{i=1}^{\lfloor \log N \rfloor} 2^{\lfloor \log N \rfloor + 1}$$

$$\leq \sum_{i=1}^{\lfloor \log N \rfloor} 2N$$
$$= O(N \log N)$$

The data structure can be set up in $O(N \log N)$ time.

Querying

Consider finding the point with the smallest x-coordinate in the query rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$. To answer this, search down the binary search tree to reach the first node v such that the left subtree of v contains y'_0 and the right subtree of v contains y'_1 (see Figure 4.6). We say that the rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$ straddles the children of v. Let y'_2 be the y-coordinate of the point at v. For any point (x, y) in the left subtree of v, $y \leq y'_2$. For any point (x, y) in the right subtree of v, $y \geq y'_2$. Perform the query *MinXInRectangle* on $[x'_0, x'_1] \times [y'_0, \infty)$ on PST($l(v)$). Since every point in PST($l(v)$) has a y-coordinate no greater than y'_2, this is equivalent to the query *MinXInRectangle* on the bounded rectangle $[x'_0, x'_1] \times [y'_0, y'_2]$. Similarly, perform the query *MinXInRectangle* on $[x'_0, x'_1] \times (-\infty, y'_1]$ in PST($r(v)$) which is equivalent to the query on the bounded rectangle $[x'_0, x'_1] \times [y'_2, y'_1]$. Thus, the query on rectangle $[x'_0, x'_1] \times [y'_0, y'_1]$ is split into two queries on rectangles $[x'_0, x'_1] \times [y'_0, y'_2]$ and $[x'_0, x'_1] \times [y'_2, y'_1]$. The query resulting in a smaller x-value is the answer to the original query.

Identifying the node v such that the query rectangle straddles the children of v takes $O(\log N)$ time. The two queries on PST($l(v)$) and PST($r(v)$) take at most $O(\log N)$ time each. Thus, the query time is $O(\log N)$.

To create the modified tree in two dimensions, we need to set up the data struc-

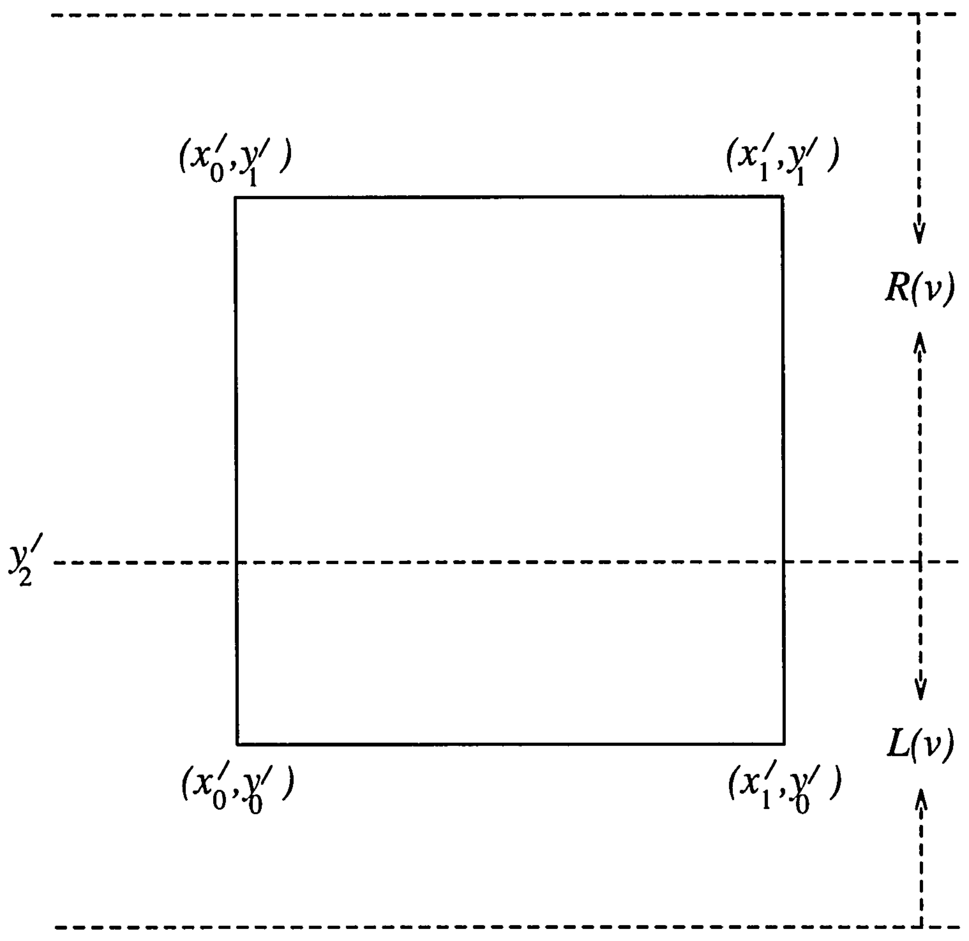

Figure 4.6: A range query in two dimensions using priority search trees

ture for two-dimensional range queries and perform $O(N)$ such queries. The total time required for this is clearly $O(N \log N)$. The rest of the computations can be done in constant time per node of the modified tree. Therefore, the modified tree can be created in $O(N \log N)$ time in two dimensions. Since the multipole method runs on the modified tree in $O(N)$ time, we have an N-body algorithm that runs in $O(N \log N)$ time in two dimensions.

CHAPTER 5. THREE-DIMENSIONAL ALGORITHMS

We now discuss three-dimensional algorithms for the N-body problem. The algorithm presented in the previous chapter for constructing the modified tree naturally extends to three and higher dimensions provided we can design algorithms for range queries.

Notation

In three dimensions, the cells in the modified tree represent cubical regions of space. For any dimension d, the cells are cubes in d-dimensions. A cell is completely determined by the length of an edge of the cell and the position of one of the corners of the cell. Without loss of generality, choose the corner with the smallest value for each coordinate. Let D be the length of the *root cell.*

Let l be any cell. We can once again describe the subcells of this cell by choosing the corner of the cell to be the origin. The cell contains 2^{kd} cells of length $\frac{l}{2^k}$. The cells are positioned at $(i\frac{l}{2^k}, j\frac{l}{2^k}, k\frac{l}{2^k})$ $(0 \leq i, j, k < 2^k)$ in three dimensions.

Definition 5.1 *A plane is called a k-boundary if it contains a surface of a cell of length* $\frac{l}{2^k}$.

In three dimensions, any *boundary* is parallel to one of the XY, YZ or ZX planes. A *boundary* can be specified by the plane to which it is parallel and the distance of the *boundary* from the plane. A *k-boundary* is at a distance of $i\frac{l}{2^k}$ $(0 \leq i \leq 2^k)$ from the plane parallel to it.

Fact 5.2 *Any k-boundary is also a j-boundary for every $j > k$.*

There are 2^k+1 planes parallel to each of XY, YZ or ZX planes and spaced $\frac{l}{2^k}$ apart that are *k-boundaries*. The intersections of the *k-boundaries* determine the cells of size $\frac{l}{2^k}$. Note that the description of the subcells and the boundaries is relative to the cell.

In a d-dimensional problem, the boundaries are hyperplanes of dimension $(d-1)$ and can be described by their distance from the $(d-1)$-dimensional plane parallel to these boundaries and passing through the origin. All the properties described in Chapter 4 are valid for any dimension d.

Creating the Modified Tree in Three Dimensions

The three-dimensional algorithm is quite similar to the two-dimensional version of constructing the tree except that each cell is now split into 8 subcells. In general, the algorithm to build the modified tree for any dimension d can be described as:

BuildTree(c)

1. Find the smallest cell c' contained in c that still contains all the particles contained in cell c.

2. If c contains no particles, return '*empty tree*'.

3. If c contains exactly one particle, return the one node tree c.

4. Split the cell c' into 2^d subcells.

5. For each subcell sc of c', *BuildTree(sc)*.

6. Return the tree obtained by joining all the trees obtained in the previous step, with c' as the root of the tree.

Once again, step 1 is accomplished by finding the smallest d-dimensional rectangular box containing all the particles in c. The box b is a rectangular parallelepiped in three dimensions, given by $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$, where x_{min} is the smallest x-coordinate of all the particles in the cell c etc.. The smallest subcell of c enclosing b is of size $\frac{l}{2^{k-1}}$, where k is the smallest integer such that a *k-boundary* passes through b. we can once again determine this by examining boundaries parallel to each coordinate planes in turn. The computation is identical to the two dimensional case except that to find x_{min}, y_{min} and z_{min}, we need to perform range queries in three dimensions.

In general, the running time of *BuildTree* in d dimensions is $O(Nd)$ plus the time required to set up a data structure for d-dimensional range queries and the time required for $O(N)$ such queries.

Range Queries in Three Dimensions

In this section, we discuss the problem of setting up a data structure to answer range queries in three dimensions. The problem we are interested in is formally described as: Given N points (x_i, y_i, z_i) $(1 \leq i \leq N)$, set up a data structure to perform queries of the form 'find the point with the minimum (maximum) x-coordinate in the rectangular parallelepiped given by $[x'_0, x'_1] \times [y'_0, y'_1] \times [z'_0, z'_1]$' efficiently. The solution is once again based on Priority Search Trees [28]. The data structure designed has $O(N \log^2 N)$ size and requires $O(N \log^2 N)$ time to compute. The queries are answered in $O(\log^2 N)$ time per query.

Setting Up the Data Structure for Range Query

In the previous chapter, we have described an algorithm to perform range queries for any arbitrary dimension d. The query time of the algorithm presented is $O(\log^d N)$. The algorithm recursively reduces a d-dimensional query to $O(\log N)$ queries in $(d-1)$ dimensions, until all queries are reduced to one-dimensional queries. The one-dimensional queries are then solved using a simple binary search. The query time of this algorithm in two dimensions is $O(\log^2 N)$. We then presented a solution using priority search trees to answer two-dimensional queries in $O(\log N)$ time. We can combine these two algorithms to answer d-dimensional queries in $O(\log^{d-1} N)$ time.

A d-dimensional query is answered once again by reducing it to $O(\log N)$ queries in $(d-1)$ dimensions. The reduction is recursively applied until all queries are reduced to two dimensions. The data structure using the priority search trees is now used to directly answer the queries in two dimensions. This reduces the running time by a factor of $O(\log N)$.

The data structure for three dimensions is set up as follows: First, sort the points by increasing z and store the points in a balanced binary search tree T. At each node of v store a structure $D(v)$, which is the two dimensional structure using priority search trees described in the previous chapter built on the x and y coordinates of the points in v's subtree. Creating $D(v)$ takes $O(m \log m)$ time where m is the number of nodes in the subtree rooted at v. Let $v_1, v_2, ..., v_k$ be the nodes in T at level i. Let $S(v_i)$ be the size of the subtree rooted at v_i. We have

$$\sum_{i=1}^{k} S(v_i) \leq N$$

The work required in creating all the $D(v_i)$'s $(1 \leq i \leq k)$ is

$$\begin{aligned} & \textstyle\sum_{i=1}^{k} S(v_i) \log S(v_i) \\ \leq\ & \textstyle\sum_{i=1}^{k} S(v_i) \log N \\ =\ & \log N \textstyle\sum_{i=1}^{k} S(v_i) \\ \leq\ & N \log N \end{aligned}$$

Hence, the two dimensional structures using priority search trees can be constructed for all the nodes in a given level of the tree T in $O(N \log N)$ time. Since there are at most $\lceil \log N \rceil$ levels in the tree, the time required for creating the entire data structure is $O(N \log^2 N)$. The space required is also $O(N \log^2 N)$.

Querying

Given a query $q = [x'_0, x'_1] \times [y'_0, y'_1] \times [z'_0, z'_1]$, first determine a set V of $O(\log N)$ nodes with the property that for each $v \in V$, the interval $[z'_0, z'_1]$ spans v's subtree but not the subtree of v's parent. This step can be accomplished in logarithmic time using algorithm *FindNodes* (Lemma 4.7). At each node $v \in V$, the z direction now becomes redundant and the problem can be solved by querying $D(v)$ with $[x'_0, x'_1] \times [y'_0, y'_1]$ and taking the minimum y returned from these queries. Time required is clearly $O(\log^2 N)$.

To create the modified tree in three dimensions, we need to set up the data structure for three dimensional range queries and perform $O(N)$ such queries. The total time required for this is clearly $O(N \log^2 N)$. The rest of the computations can be done in constant time per node of the modified tree. Therefore, the modified tree can be created in $O(N \log^2 N)$ time in two dimensions. Since the multipole method runs on the modified tree in $O(N)$ time, we have an N-body algorithm that runs in $O(N \log^2 N)$ time in three dimensions.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Conclusions

The study of physical systems by particle simulation is becoming increasingly important in scientific computing. The traditional solutions based on grid methods are appropriate only for uniform distributions. A new class of algorithms known as hierarchical N-body methods have emerged to address the problem of efficiently performing particle simulations for non-uniform distributions. Unfortunately, most of these algorithms are analyzed for uniform distributions and the results are expected to be valid for non-uniform distributions.

In this thesis, we have presented a rigorous worst-case analysis of some of the popular hierarchical N-body algorithms along with proofs that the running times of these algorithms are not valid for arbitrary distributions. We have presented a distribution-independent hierarchical clustering scheme and have presented monopole and multipole methods based on this scheme. The multipole method based on our scheme runs in $O(N \log N)$ time in two dimensions and $O(N \log^2 N)$ time in three dimensions irrespective of the distribution. A key feature of our algorithm is that its running time is purely a function of the number of particles. In contrast, the existing algorithms have running times that depend on the positions of the particles and the precision of the machine.

Future Work

Optimal Clustering Schemes

All the hierarchical N-body algorithms depend on clustering of the particles into a hierarchical structure and approximating the particle-cluster and/or cluster-cluster interactions instead of computing interactions between the individual particles. The amount of work required in the force calculation stage clearly depends upon the way the particles are clustered. Except for the initial paper on hierarchical N-body methods by Appel [5], all the other algorithms rely on a fixed cubical subdivision of the space into cells, irrespective of the location of the particles. Appel allows for arbitrarily shaped clusters modified across iterations using heuristics. It is interesting to study the problem of generating optimal clustering mechanisms, where optimal is defined as the clustering scheme resulting in the least amount of work in the force calculation stage.

It should be noted that performing force calculations without any clustering requires $O(N^2)$ time. If clustering were performed to reduce the work in force calculation, the clustering algorithm is useful only if it has approximately the same complexity as the force calculation using the clustering. Due to this, generating an optimal clustering scheme may not be useful. However, it may be possible to update the optimal clustering from one iteration to another efficiently. Even otherwise, an optimal clustering scheme will be useful in radiosity applications, where the particles represent polygonal patches in the scene and hence do not move. Thus, the 'bodies' do not move between iterations and the same clustering can be used throughout.

Designing Faster Algorithms

The N-body problem has an obvious lower bound of $O(N)$ to compute all pair-wise interactions. In light of the proof that Greengard's and other multipole related methods are not $O(N)$, the fastest distribution independent algorithm has a complexity of $O(N \log N)$ in two dimensions and $O(N \log^2 N)$ in three dimensions. This complexity is mainly due to the hierarchical tree creation. However, it may be possible to update the tree in linear time across iterations. The hope for the possibility of such an algorithm stems from the fact that the particles are guaranteed to move only by a small distance in each iteration. A linear time algorithm for updating the tree results in an algorithm with complexity matching the lower bound.

Reducing the Number of Iterations

Most of the research on N-body methods is concentrated on reducing the $O(N^2)$ complexity per iteration of the naive algorithm. The particle simulation continues by computing the updated position of the particles over a short time interval δt and repeating the force computations. The common approach is to use the same global time increment δt for all pairs of particles in the system. When two particles come very close to each other, extremely small time increments are necessary due to the resulting high acceleration. Given a large system of particles over varying length scales, it is highly probable that such close particles exist, resulting in a large number of iterations. Using variable time increments based on the distance between interacting particles/clusters is a viable alternative. The scheme has been suggested by Appel, but it is yet to be formally analyzed and integrated into the more popular algorithms.

Parallel Algorithms

The N-body problem poses a formidable challenge for parallel computation. The properties of the problem including highly non-uniform and dynamic distribution and the necessity for global data in computing interactions make it difficult to parallelize. Some researchers including Warren and Salmon [37, 38], Singh [32] have worked in this area but optimal algorithms still elude discovery.

BIBLIOGRAPHY

[1] S.J. Aarseth, J. Richard Gott III and E.L. Turner, N-body simulations of galaxy clustering; I. Initial conditions and galaxy collapse times, *Astrophys. J., 228* (1979) 664-683.

[2] S. Aluru, G.M. Prabhu and J. Gustafson, Truly distribution-independent algorithms for the N-body problem, *Proc. Supercomputing '94* (1994), to appear.

[3] C.R. Anderson, An implementation of the fast multipole method without multipoles, *SIAM J. Sci. Stat. Comp., 13* (1992) 923-947.

[4] A.W. Appel, *An investigation of galaxy clustering using an asymptotically fast N-body algorithm*, Undergraduate thesis, Princeton Univ., Princeton, NJ, April 1981.

[5] A.W. Appel, An efficient program for many-body simulation, *SIAM J. Sci. Stat. Comp., 6* (1985) 85-103.

[6] J. Barnes, A modified tree code: Don't laugh; It runs, *J. Comp. Phys., 87* (1990) 161-170.

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

[8] J. Barnes and P. Hut, Error analysis of a tree code, *Astrophys. J. (Suppl.) 70* (1989) 389-417.

[9] S. Bhatt, M. Chen, C.Y. Len and P. Liu, Abstractions for parallel N-body simulations, Tech. Rep. DCS/TR-895, Yale University, New Haven, CT, 1992.

[10] J. Carrier, L. Greengard and V. Rokhlin, A fast adaptive multipole algorithm for particle simulations, *SIAM J. Sci. Stat. Comp., 9* (1988) 669-686.

[11] T. Chan, Hierarchical algorithms and architectures for parallel scientific computing, *Proc. ACM Conf. on Supercomputing* (1990) 125-134.

[12] K. Esselink, The order of Appel's algorithm, *Info. Proc. Letters, 41* (1992) 141-147.

[13] D.P. Fullagar, P.J. Quinn, C.J. Grillmair, J.K. Salmon and M.S. Warren, N-body methods on MIMD supercomputers: Astrophysics on the Intel Touchstone Delta, *Proc. Fifth Australian Supercomputing Conference* (1992) 234-241.

[14] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, MIT Press, Cambridge, MA, 1988.

[15] L. Greengard and W. Gropp. *Parallel Processing for Scientific Computing,* Chap: A Parallel Version of the Fast Multipole Method, SIAM, 1987, 213-222.

[16] L. Greengard and V. Rokhlin, A fast algorithm for particle simulations, *J. Comp. Phys., 73* (1987) 325-348.

[17] P. Hanrahan, D. Salzman and L. Aupperle, A rapid hierarchical radiosity algorithm, *Proc. Computer Graphics '91* (1991) 197-206.

[18] L. Hernquist, Performance characteristics of tree codes, *Astrophys. J. (Suppl.) 64* (1987) 715-734.

[19] L. Hernquist, Vectorization of tree traversals, *J. Comp. Phys., 87*(1990) 137-147.

[20] R.W. Hockney and J.W. Eastwood, *Computer simulation using particles*, McGraw-Hill, New York, 1981.

[21] R. Janardan, *Personal Communication.*

[22] J.G. Jernigen and D.H. Porter, A tree code with logarithmic reduction of force terms, hierarchical regularization of all variables and explicit accuracy controls, *Astrophys. J. (Suppl.), 71* (1989) 871-893.

[23] J. Katzenelson, Computational structure of the N-body problem, *SIAM. J. Sci. Stat. Comp., 10* (1989) 787-915.

[24] J. Makino, Comparison of two different tree algorithms, *J. Comp. Phys., 88* (1990) 393-408.

[25] J. Makino, Vectorization of a treecode, *J. Comp. Phys., 87* (1990) 148-160.

[26] J. Makino and P. Hut, Performance analysis of direct N-body simulations, *Astrophys. J. (Suppl.), 68* (1988) 833-856.

[27] J. Makino and P. Hut, Gravitational N-body algorithms: A comparison between supercomputers and a highly parallel computer, *Computer Physics Reports, 9* (1989) 199-246.

[28] E.M. Mc Creight, Priority Search Trees, *SIAM J. Comp.* (1985) 257-268.

[29] R.H. Miller and K.H. Prendergast, Stellar dynamics in a discrete phase space, *Astrophys. J., 151* (1968) 699-709.

[30] R.H. Miller, K.H. Prendergast and W.J. Quirk, Numerical experiments on spiral structure, *Astrophys. J., 161* (1970) 903-916.

[31] J.K. Salmon, *Parallel hierarchical N-body methods*, Ph.D. thesis, California Institute of Technology, 1991.

[32] J.P. Singh, *Parallel hierarchical N-body methods and their implications for multiprocessors*, Ph.D. thesis, Stanford University, Stanford, CA, 1993.

[33] J.P. Singh, J.L. Hennessy and A. Gupta, Scaling parallel programs for multiprocessors: methodology and examples, *IEEE Computer* (1993) 42-50.

[34] J.P. Singh, C. Holt, T. Totsuka, A. Gupta and J.L. Hennesy, Load balancing and data locality in hierarchical N-body methods, *J. Parallel Distrib. Comput.*, to appear.

[35] B.E. Smits, J.R. Arvo and D.H. Salesin, An importance-driven radiosity algorithm, *Proc. Computer Graphics '92* (1992) 273-282.

[36] M.S. Warren and J.K. Salmon, A parallel treecode for gravitational N-body simulations with up to 20 million particles, *Bull. Amer. Astro. Soc., 23* (1991) 1345-1363.

[37] M.S. Warren and J.K. Salmon, Astrophysical N-body simulations using hierarchical tree data structures, *Proc. Supercomputing '92* (1992) 570-576.

[38] M.S. Warren and J.K. Salmon, A parallel hashed oct-tree N-body algorithm, *Proc. Supercomputing '93* (1993) 1-12.

[39] F. Zhao, *An $O(N)$ algorithm for three-dimensional N-body simulations*, M.S. Thesis, Massachusetts Institute of Technology, Boston, MA, 1987.

[40] F. Zhao and L. Johnsson, The parallel multipole method on the connection machine, *SIAM. J. Sci. Stat. Comp., 12* (1991) 1420-1437.