Distribution Independent Spatial Data Structures and Algorithms with Applications

by

FATIH E. SEVILGEN

B.S., Bogazici University

M.S., Syracuse University

DISSERTATION

Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer Science
in the Graduate School of Syracuse University

February 2000

Approved _____

Professor Srinivas Aluru

Date _____

© Copyright 1999 Fatih E. Sevilgen

All rights reserved

The main theme of this dissertation is to provide distribution-independent tools such as data-structures and algorithms that are both efficient and easy to use. The computational complexities of the tools presented are examined as rigorously as possible.

The first tool is about the use of space-filling curves. Space-filling curves have immense application areas range from databases to scientific computation. However space-filling curves are defined for a uniform grid of a particular resolution. This results in the running time depending on size of the grid. For arbitrary distributions of data, the grid size could be very large independent of the size of data. We present a simple and efficient algorithm for ordering arbitrary and dynamic multidimensional data in which a particular space-filling curve visits them. The running time of the algorithm is proved to be $O(n \log n)$ for n points independent of the distribution. To support our claims, experimental results for two application areas, databases and domain decomposition, are presented.

The second tool is a dynamic data structure for multidimensional point data that provide both distribution independent size and manipulation algorithms. The DISH tree is based on well known quadtree data structure and eliminates its drawbacks with couple of modifications and extensions. We give the definition of the DISH tree and prove that its size is $O(n)$ for n points. We also present algorithms for searching, insertion, and deletion with $O(\log n)$ running time. To our best knowledge, DISH tree is the first data structure that provides these optimal properties while preserving the basic hierarchical cell structure. The other data structures in the literature either fail

to provide dynamic operations in optimal time or sacrifice from the hierarchical cell structure to get better algorithms even though they are not optimal for some specific cases.

To facilitate the use of the DISH trees, we surveyed some scientific applications that are good candidates to employ the DISH trees, and provide efficient algorithms for the access structures of them. Some of the access structures are k-nearest neighbor, neighbors in a circular region, symmetric k-nearest neighbor, and isotropic neighbor.

Even though the computational power of sequential computers increase exponentially, it is still far inadequate to satisfy the demands of the scientific community. Moreover, the limits microprocessors technology justifies the essentiality of parallel computing. Many scientific algorithms are already been parallelized. To expedite the use of the DISH tree for these algorithms, we present parallel counterpart of our sequential DISH tree algorithms.

We show the relevance of our results by employing the parallel DISH tree in two applications, N-body problem and molecular dynamics. We only regard the applications in computer science perspective and do not consider the detailed computations such as how to compute momentum in SPH method.

approval page

Contents

1 INTRODUCTION **1**

2 PRELIMINARIES **3**

2.1 Hierarchical Data Structures: Hyperoctrees and Compressed Hyperoctrees 3

2.2 Space-Filling Curves 9

2.3 Smallest Cell Technique 13

2.3.1 Model of Parallel Computation 17

2.3.2 Primitive Operations 19

3 SPACE FILLING CURVES FOR ARBITRARY MULTIDIMENSIONAL POINTS **22**

3.1 Distribution Independent Ordering Of Multidimensional Points 24

3.2 Applications 29

3.2.1 Storing Database Records and Range Queries Using Space-filling Curves 29

3.2.2 Experimental Results 38
3.2.3 Parallel Domain Decomposition Methods 46
3.2.4 Parallel Incremental Load Balancing 50

4 DISTRIBUTION INDEPENDENT ADAPTIVE TREES 52

4.0.5 Implementation Issues 55
4.1 Building the DIAT Tree 57
4.1.1 A Divide-and-Conquer Algorithm 57
4.1.2 Bottom-up Construction 63
4.1.3 Construction by Repeated Insertions 67
4.1.4 Construction of BBST 67
4.2 Operations on DIAT Trees 67
4.2.1 Point and Cell Queries 67
4.2.2 Insertions and Deletions 69
4.2.3 Group Queries 71

5 PARALLEL TREE ACCUMULATIONS 77

5.1 Tree Accumularions on PRAM Model 78
5.2 Distributed Memory Tree Accumulations 81
5.2.1 Upward Accumulation 85
5.3 Downward Accumulation 92

6 FAST MULTIPOLE METHOD ALGORITHM USING DIAT TREES 96

6.1 Solving N-body Problem Using DIAT Trees 100
6.1.1 Incremental Updates 110

7 PARALLEL N-BODY ALGORITHM 112

7.1 Parallel Fast Multipole Method 115
7.1.1 Compressed Octree Construction 116
7.1.2 Multipole Expansion Calculation 119
7.1.3 Partial Local Expansion Calculation 119
7.1.4 Local Expansion Calculation 126
7.1.5 Parallel Run-time Analysis 126

8 CONCLUSION AND OPEN PROBLEMS 128

List of Tables

3.1 Parallel partitioning time (in seconds) for uniform and non-uniform graphs of different sizes. 49

5.1 Running time of parallel prefix and random data accesses on the EREW PRAM model and distributed memory computers. 83

List of Figures

2.1 Illustration of hierarchy of cells in two dimensions. 6

2.2 A two-dimensional domain and the corresponding quadtree and compressed quadtree. 7

2.3 Z-curve for grids of size 2×2, 4×4 and 8×8. 11

2.4 Graycode curve for grids of size 2×2, 4×4 and 8×8. 12

2.5 Hilbert curve for grids of size 2×2, 4×4 and 8×8. 13

2.6 A *root cell* of length l and a rectangle R. 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*. 15

3.1 Z-curve for a collection of 12 points using the hierarchical subdivision approach. 26

3.2 A simple algorithm for range queries. 31

3.3 A two-dimensional domain and a query region R. The different numbered boundaries are shown using different styles of lines. The largest dashed lines are 1-boundaries and the smallest dashed lines are 4-boundaries. 33

3.4 A query region and a point P close to its boundary. To determine that P is not in the query region, the search region should be broken into very small cells. 34

3.5 A distribution-independent algorithm for range queries. 37

3.6 Uniformly and non-uniformly distributed data in Z-order. 39

3.7 Number of points in the query region and area of the query region versus percentage of useful points and blocks for uniform and non-uniform distributions. 41

3.8 Number of points in the query region versus percentage of useful points and blocks for various aspect ratios (A = width/length) for the non-uniform distribution. 43

3.9 Size of the query region versus percentage of useful points and blocks for various aspect ratios (A = width/length) for the non-uniform distribution. 44

3.10 Number of points in the query region and size of the query region versus percentage of useful points and blocks for various block sizes (number of points in a block). 45

3.11 Non-uniform computational graphs and their ordering according to the Z-curve. Graph 1 has 6019 nodes, Graph 2 has 10166 nodes and Graph 3 has 15606 nodes. 48

4.1 The hyperoctree and the DIAT tree for 10 points in a two dimensional domain. The positions of the points in the domain are shown on the left. The square regions represents the subdivision of the domain in the construction of the hyperoctree. The resulting hyperoctree is shown at the top right. The corresponding DIAT tree is at the bottom right. The pointers from the leaves to the internal nodes of the DIAT tree are shown but the auxiliary data structure on the leaves is not shown. 54

4.2 Algorithm to merge two DIAT trees rooted at u and v. Note that to merge two nodes, the large cells of both u and v should be subcells of the same root cell and $L(u) \cap L(v) \supseteq S(u) \cup S(v)$. We use subscripts to represent the children of a node. 59

4.3 Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are equal. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The resulting tree is shown on the right. The node M(u_i,v_i) is the result of merging u_i and v_i. 60

4.4 Merging subtrees rooted at u and v when $S(u)$ contains $S(v)$. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right. The node M(u_i,v_i) is the result of merging u_i and v_i. 61

4.5 Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are disjoint. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right. 62

4.6 Algorithm for the bottom-up construction of the DIAT tree. 64

4.7 Illustration of the two cases of bottom-up DIAT tree construction. The figures on the left shows the DIAT trees and the domains before inserting p and the resulting trees are on the right. 65

4.8 Algorithm to insert the cell C in the DIAT tree. 70

4.9 Unified algorithm for the group queries. 72

5.1 Algorithm for parallel upward accumulations. 87

5.2 Execution of the parallel upward accumulation algorithm. The initial tree is depicted in (A). The first two iterations of the algorithm are shown in (B) and (C). The left tree is after step 1 and the right tree is after step 2 of the corresponding iteration. 88

5.3 Algorithm for calculation of downward accumulations. 95

6.1 Algorithm for calculating local expansions of all nodes in the tree rooted at v. 106

7.1 Algorithm for parallel construction of compressed octree. 117

7.2 Algorithm for parallel calculation of partial local expansions. 124

Chapter 1

INTRODUCTION

Placing the work in context and a description of main results in the dissertation.

Talk about distibution-independence. Why it is important.

There has been a significant interest in the scientific community for using space-filling curves for a variety of applications. Several researchers have suggested and used space-filling curves for secondary storage in databases [35, 36, 51, 54, 75]. They are used for reducing the bandwidth of digitally sampled signals [20] and for generating graphics displays [82]. Using space-filling curves, heuristic algorithms have been designed for NP-hard problems such as the traveling salesperson problem [85] and digital halftoning [11]. Pilkington et. al. [84] explore the use of space-filling curves for partitioning workloads to processors of a parallel computer. They have also been used to perform domain decomposition in parallelizing scientific applications. For example, see the parallel N-body algorithm by Warren et. al. [111].

No sophisticated algorithms for these operations are known prior to this work.

In this paper, we present a dynamic variant of compressed hyperoctrees that admits point insertions and deletions. We call this new data structure *Distribution-Independent Adaptive Tree (DIAT)*. We present three optimal deterministic algorithms that each construct the DIAT tree in $O(n \log n)$ time. We also present algorithms for point and cubic region searches, point insertion and point deletion that run in $O(\log n)$ time irrespective of the distribution of the points. To achieve distribution-independence, the DIAT tree is accessed from the leaves rather than the root.

We expect that DIAT trees can be used in place of hyperoctrees by any algorithm that makes use of hyperoctrees. We believe that employing DIAT trees can eliminate the dependence on distribution and result in faster algorithms. As evidence, we provide an algorithm to solve the N-body problem in $O(n)$ time using the DIAT tree. Moreover, our deterministic algorithm for the DIAT tree construction presented in this paper combined with Clarkson's $O(n)$ nearest neighbors algorithm yields an $O(n \log n)$ deterministic algorithm for computing all nearest neighbors. Further, insertions and deletions do not require rebuilding of the tree before nearest neighbors can be found.

Chapter 2

PRELIMINARIES

2.1 Hierarchical Data Structures: Hyperoctrees and Compressed Hyperoctrees

Hierarchical tree data structures are a class of data structures to represent multidimensional data based on the recursive subdivision of the region containing all the data. To construct such a data structure, start with this region and the region is partitioned into subregions using a subdivision scheme. This hierarchical subdivision process is represented by the hierarchical data structure, where each node corresponds to a region and its children correspond to the non-empty subregions obtained by subdividing it.

Hierarchical data structures for multidimensional data can be broadly classified into two categories according to the subdivision scheme: data driven and domain driven. Domain driven data structures use a fixed subdivision scheme regardless of

the actual properties of the data. A popular example is the *hyperoctree* data structure [99]. Data driven data structures use information about the input data to decide the subdivision process. An example is the multidimensional binary search trees, abbreviated *k-d trees*, introduced by Bentley [17].

Data driven data structures are usually distribution-independent and admit efficient, distribution-independent construction and search algorithms. However, they often do poorly on update operations such as deletion and queries requiring spatial information. On the other hand, domain driven data structures are not distribution-independent and in principle, can be unbounded for arbitrary distributions. Consequently, even basic algorithms such as construction and searches are not distribution-independent. Nevertheless, they are very popular due to their suitability for many applications and they often perform well for the range of distributions encountered in practice [29]. Popular application areas include databases, scientific computing and computer graphics.

This section contains an overview of hyperoctrees for any fixed dimension d. These are popularly known as quadtrees in two dimensions and octrees in three dimensions. The focus is on representation of point data.

The following notation in describing hyperoctrees: Call a hypercubic region containing all the points the *root cell*. Define a hierarchy of cells by the following: The root cell is in the hierarchy. If a cell is in the hierarchy, then the 2^d equal-sized hypercubic subregions obtained by bisecting along each coordinate of the cell are also called cells and belong to the hierarchy. Two cells are adjacent if they intersect only

at the boundaries. We adopt the convention that two cells are disjoint if they do not intersect at all or if they are merely adjacent.

Given two cells in the hierarchy, possibly of different sizes, either one is completely contained in the other or they are disjoint. We use the term *subcell* to describe a cell that is completely contained in another. A cell containing the subcell is called a *supercell*. The 2^d subcells obtained by bisecting a cell along each dimension are also called the *immediate subcells* with respect to the bisected cell. Also, a cell is the *immediate supercell* of any of its immediate subcells. We can treat a cell as a set of all points in space contained in the cell. Thus, we use $C \subseteq D$ to indicate that C is a subcell of D and $C \subset D$ to indicate that C is a subcell of D and C is strictly smaller than D. Define the *size of a cell* C, denoted $size(C)$, to be the number of points in cell C. Define the *length of a cell* C, denoted $length(C)$, to be the span of C along any dimension.

A two dimensional example of such a hierarchy is given in figure 2.1. Each subsequent layer represents a subsequent subdivision process. The cell in the first layer is the root cell. While the cells D and E are disjoint, C and H are not. Both the cells H and D are subcells of C and D is an immediate subcell of C. All the cells in a layer have the same length.

The hierarchy of cells are represented by an hyperoctree (quadtree in two-dimensions or actree in three-dimensions). Each node in the hyperoctree corresponds to a cell which encloses at least one particle. The empty cells are not represented. The root node represents the root cell and the immediate subcells of the cell corresponding to

Figure 2.1: Illustration of hierarchy of cells in two dimensions.

a node are represented by its children. The quadtree for the hiearachy in Figure 2.1 is given in Figure 2.2.

For uniform distribution of points, this subdivision process results in a height balanced tree having size linear in the number of points. For arbitrary distributions, the size of the hyperoctree is, in principle, unbounded. This is because a cell may contain all its points in a very small volume so that, recursively subdividing it may result in just one cell containing the points for an arbitrarily large number of steps. In terms of the octree, there can be an arbitrarily long path without any branching. Note that the regions represented by nodes along such a path have different sizes (increasingly smaller sizes away from the root) but they all contain the same points. For many applications, all these nodes essentially contain the same information as the information depends only on the particles the cell contains. Compressed octrees

Figure 2.2: A two-dimensional domain and the corresponding quadtree and compressed quadtree.

are obtained by compressing each such path into a single node. Therefore, each node in a compressed octree is either a leaf or has at least two children (see Figure 2.2 for a two-dimensional illustration).

Such *compressed hyperoctrees* have been studied by Clarkson [25] in the context of the all nearest neighbors problem. Clarkson uses the name *cell tree* and presents a randomized algorithm to construct such a tree for n points in $O(n \log n)$ expected time. He also presents an $O(n)$ deterministic algorithm to solve the all nearest neighbors problem using the cell tree. An $O(n \log n)$ deterministic algorithm for the all nearest neighbors problem is presented by Vaidya in [112]. Though Vaidya does not explicitly mention it, the computational tree implied by his algorithm is equivalent to a compressed hyperoctree. This analogy can be used to derive an $O(n \log n)$ deterministic algorithm for compressed hyperoctree construction. However, the algorithms presented in this paper are much simpler.

Another attempt to rectify the distribution-dependent nature of the hyperoctrees is BD trees by Ohsawa et. al. [27, 72]. By sacrificing the regular hierarchical cubic structure of the regions represented by hyperoctree nodes, they aim to balance the height of the tree. This work can also be considered as an attempt to generalize the one dimensional balanced binary search trees as they use rotation operations similar to the AVL trees to re-balance BD trees. However, maintaining the balanced tree structure is not always possible with arbitrary insertions and deletions. Algorithms to dynamize the hyperoctrees are discussed by Overmars in [79]. However, these are just straightforward algorithms having unbounded running time in the worst case.

The compressed hyperoctrees are distribution-independent as the size of the data structure is in $O(n)$. However, straightforward algorithms for searching, insertion and deletion in compressed hyperoctrees are not distribution-independent and require $O(n)$ time in the worst case. This is because the height of the tree depends on the distribution of the points and is $O(n)$ for some distributions.

2.2 Space-Filling Curves

Several applications benefit from the mapping of multidimensional points to a one-dimensional ordering. The objective in performing such a mapping is that points that are proximate in the multidimensional space should be proximate in the one-dimensional ordering, i.e., the mapping should preserve the "proximity information" present in the multidimensional space. Of course, it is impossible to design a mapping function that strictly preserves the proximity information. For example, a point may have several nearest neighbors in multidimensional space but can have only two nearest neighbors in a one-dimensional ordering. Hence, the goal is to design mappings that preserve the proximity information to the extent possible.

Space-filling curves are mappings from a multidimensional grid to a linear ordering. The path implied in the multidimensional space by the linear ordering, i.e., the sequence in which the multidimensional grid cells are visited according to the linear ordering, forms a non-intersecting curve. By continually increasing the resolution of the grid, the curve can come arbitrarily close to any point in space. Hence the name,

space-filling curve. Some of the popular curves used are the Z-curve [70], the gray-code curve [36] and the Hilbert curve [48, 83]. The proximity information preserving properties of space-filling curves are studied experimentally by Abel et. al. [2] and Jagadish [54]. Some analytical results are given by Moon et. al. [69]. Further general discussion about space-filling curves can be found in [94].

In this section, we describe the notion of space-filling curves and describe some of the popularly used curves. We first restrict our attention to square two-dimensional grids of size $2^k \times 2^k$. Space-filling curves can be described in a variety of ways. They can be described recursively. The curve for a $2^k \times 2^k$ grid is composed of four $2^{k-1} \times 2^{k-1}$ grid curves. Alternatively, one can describe a curve for a $2^k \times 2^k$ grid as constructed by taking the curve for a $2^{k-1} \times 2^{k-1}$ grid and replacing each grid cell of the curve by an appropriately configured 2×2 curve. The curves can also be specified by bit interleaving - the position of a grid point along the curve can be described by interleaving the bits of the coordinates of the point with the interleaving function being a characteristic of the curve. The curves have also been described using EOL type grammars [12]. For our purposes, an intuitive recursive description suffices.

Z-Curve. The Z-curves [70] for 2×2, 4×4 and 8×8 grids are shown in Figure 2.3. The curve is composed of segments which look like the letter 'Z' rotated counterclockwise by 90^o, hence the name Z-curve. The curve has a beginning called the *head* and the end is called the *tail*. The head of the curve is in the lower left corner and the tail is in the upper right corner. The curve for a $2^k \times 2^k$ grid is composed of four $2^{k-1} \times 2^{k-1}$ grid curves one in each quadrant of the $2^k \times 2^k$ grid and the tail of one

Figure 2.3: Z-curve for grids of size 2×2, 4×4 and 8×8.

curve is connected to the head of the next as shown in the figure. The order in which the curves are connected is the same as the order of traversal of the 2×2 curve, and this holds true for all space-filling curves.

Alternatively, given a point in two dimensions and assuming that each coordinate is represented using k bits, the position of the point on the Z-curve can be obtained by interleaving the bits representing the x and y coordinates, starting from the x coordinate. For example, $(3, 5) = (011, 101)$ translates to $011011 = 27$. All the space-filling curves can also be described in such a fashion using a suitable bit-interleaving function.

Graycode Curve. The graycode curve [35] uses the same bit interleaving function as the Z-curve. While the Z-curve visits points in increasing numerical order of the results of applying the interleaving function to the points, the graycode curve visits them in the graycode order (see Figure 2.4). The curve can be described recursively as follows: Place four curves of the previous resolution in the four quadrants. Flip the

Figure 2.4: Graycode curve for grids of size 2×2, 4×4 and 8×8.

upper quadrant curves once around one of the coordinate axes and then around the other axis. The curves in the lower quadrants are left as they are. The four curves are then connected in the order of traversal of the 2×2 curve. The head of the curve is in the lower left hand corner.

Hilbert Curve. The Hilbert curve [48, 83] is a smooth curve that avoids the sudden jumps present in the Z and the Graycode curves (Figure 2.5). The curve is composed of four curves of the previous resolution placed in the four quadrants. The curve in the lower left quadrant is rotated clockwise by 90^o. The curve in the lower right quadrant is rotated counterclockwise by 90^o. The curves are connected in the order prescribed by the 2×2 curve.

The curves can be easily extended for cases when the dimension spans are uneven and for higher dimensional grids. Consider a $2^k \times 2^m$ grid ($k > m$). One can use 2^{k-m} curves each for a $2^m \times 2^m$ grid and place them next to each other. If needed, the curves can be tilted upside down as appropriate, to avoid sudden jumps. Alternatively, one

Figure 2.5: Hilbert curve for grids of size 2×2, 4×4 and 8×8.

can define a $2^{k-m+1} \times 2$ template and use it to construct the curve for a $2^k \times 2^m$ grid. The two strategies can also be combined: use a $2^{l+1} \times 2$ ($0 \leq l \leq k-m$) template to construct a curve for a $2^{m+l} \times 2^m$ grid and use 2^{k-m-l} copies of it placed next to each other. A higher dimensional curve can be obtained by using a template of the same dimension (such as a $2 \times 2 \times 2$ grid for three dimensions). A detailed description of space-filling curves for uneven dimension spans and higher dimensions can be found in [54].

2.3 Smallest Cell Technique

In this section, we introduce smallest cell technique. Smallest cell thechnique gives a constant time solution to the problem defined as: For given two points, find the smallest subcell of the root cell that contains the points. It is proposed by Clarkson [25] in the context of nearest neighbor problem. Smallest cell technique is used to find the least common achestor of two nodes in hyperoctrees and sorting the points

according to a spavce-filling curve in the subsequent chapters.

Consider a rectangle R that has the given two points at the opposite ends of a diagonal - i.e., if the given points are (x_1, y_1) and (x_2, y_2), consider the rectangle R formed by (x_1, y_1), (x_1, y_2), (x_2, y_1) and (x_2, y_2). The smallest cell enclosing the two points is also the smallest cell that encloses the rectangle R, which we shall determine. First, some terminology (see Figure 2.6):

Definition 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 1 *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 2.6.

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

The smallest subcell of the root cell enclosing R is of size $\frac{l}{2^{k-1}}$, where k is the smallest integer such that a *k-boundary* passes through R (Figure 2.6). To determine this, we can examine boundaries parallel to each coordinate axis in turn.

Figure 2.6: A *root cell* of length l and a rectangle R. 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*.

Consider boundaries parallel to the y-axis. Let s be the smallest integer such that an s-*boundary* parallel to the y-axis passes through R, i.e., s is the smallest integer such that $x_{min} < i\frac{l}{2^s} < x_{max}$ for some i.

Lemma 1 *Exactly one s-boundary passes through R.*

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

$i+1$. Let $i'' = \frac{i'}{2}$. One of the *s-boundaries* is specified by $i'\frac{l}{2^s} = i''\frac{l}{2^{s-1}}$. Therefore, this is also a $(s-1)$-*boundary* and a $(s-1)$-*boundary* passes through R, contradicting the minimality of s. ■

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

Lemma 2 *There is at least* 1 *and at most* 2 *t-boundaries passing through R.*

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

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

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

Lemma 3 *The s-boundary passing through R is either* h_1 *or* h_2.

Proof: Suppose not. Since s is the smallest integer such that an *s-boundary* passes through R, $s \leq t$ and by Fact 2, the *s-boundary* passing through R is also a *t-boundary*. By Lemma 5, h_1 and h_2 are the only *t-boundaries* passing through R. Therefore, the *s-boundary* passing through R must coincide with h_1 or h_2. ■

It is now easy to find s since the choice for the *s-boundary* passing through R 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 $t-s$ is equal to the highest power of 2 that divides a'. One way to find this is to set $t-s = \log_2(1+\{a' \oplus (a'-1)\})-1$. Since all the above operations take constant time, the smallest cell contained in the root cell enclosing the box R can be determined in constant time.

2.3.1 Model of Parallel Computation

In this section, we present the parallel model of computation we assume while designing the algorithms in the subsequent chapters.

Multiprocessor platforms have converged to an architecture consisting of a moderate number (ten to a few thousand) of fast and powerful processors connected by a high-speed communication network. Processors interact either through explicit message passing (IBM SP, Workstation clusters) or through shared-address space (SGI Origin, DASH). In either case, the memory is physically distributed with processors having fast access to only certain portions of the memory. In message-passing systems, the memory is divided equally among the processors. Each processor has access to its own local memory and accessing remote memory is done only though explicit message passing with the processor that owns it. Though the interconnection network topology determines the distance a message has to travel, this effect is marginalized by the typically large start-up costs, the moderate size of the machine and hardware routing techniques such as cut-through routing. Though contention is a problem in

the presence of multiple messages and this depends on the bisection bandwidth of the network topology, the effect of contention is eased due to the presence of virtual channels and often the link bandwidth is higher than the node interface bandwidth. Thus, we model communication cost in the presence of multiple messages as follows: In a communication step,

1. Each processor is allowed to send at most one message and receive at most one message.

2. The cost of the communication step is modeled by $\tau + \mu m$, where τ is the start-up time to initiate a communication, $\frac{1}{\mu}$ is the data rate supported by the network, and m is the length of the message.

This corresponds to the assumption that communication corresponding to any permutation can be realized simultaneously. This is the so called *permutation network* model.

In a shared address space architecture, the entire memory is directly accessible to all the processors. Because accessing memory over the interconnection network is expensive and may cause long delays due to contention, shared memory systems employ large caches and access memory in units of large blocks which are stored in local caches. However, multiple writes to the same block of data by different processors raises the *cache coherency* problem, which is handled by the hardware. This has the effect of serializing access to a block in presence of writes. Thus, performance degrades severely in presence of conflicting writes to the same block of data. In

essence, the local caches act as distributed memory. Thus, the classic shared memory assumption that each processor can access a memory location in constant time does not hold good. In fact, a run-time analysis based on a distributed memory architecture often closely reflects the performance of shared address space architectures as well. For this reason, we will limit our run-time analysis to the distributed-memory model.

It should be stressed that the algorithms presented in this paper are applicable to other models of computation as well, and often equally efficiently. This is because we use a few simple communication primitives such as sorting, parallel prefix and broadcast.

2.3.2 Primitive Operations

In the following, we describe the primitive parallel operations used in our algorithms. As these operations are well known, we briefly describe them and state the run-time, omitting the run-time analysis. For a detailed description and run-time analysis, see [59]. In what follows, p denotes the number of processors.

- **Broadcast:** In a Broadcast operation, one processor has a message of size m to be sent to all other processors. This operation takes $O((\tau + \mu m) \log p)$ time.

- **Parallel Prefix:** Consider n data items $x_1, x_2, \ldots, x_n$ and a binary associative operator $\otimes$ that operates on these data items and produces a result of the same type. The prefix sums problem is to compute $s_1, s_2, \ldots, s_n$, where

$$s_i = x_1 \otimes x_2 \otimes x_3 \otimes \ldots \otimes x_i$$

The s_i's are often called partial sums. Parallel prefix is the problem of computing the prefix sums in parallel. This problem can be solved in $O(\frac{n}{p}+(\tau+\mu)\log p)$ time.

- **Segmented Parallel Prefix:** Segmented prefix computation is a sequence of prefix computations using an associative operator $\otimes$. Consider n data items $x_{1,1}$, $x_{1,2}$, ..., x_{1,n_1}, $x_{2,1}$, $x_{2,2}$, ..., x_{2,n_2}, ..., $x_{m,1}$, $x_{m,2}$, ..., x_{m,n_m}, where $\Sigma_{i=1}^{m} n_i = n$. We want to compute $s_{i,j}$, where

$$s_{i,j} = x_{i,1} \otimes x_{i,2} \otimes x_{i,3} \otimes \ldots \otimes x_{i,j}$$

for $1 \leq i \leq m, 1 \leq j \leq n_i$. This operation can be done using one parallel prefix operation. Therefore, segmented parallel prefix operation takes $O(\frac{n}{p} + (\tau + \mu)\log p)$ time.

- **All-to-All Communication:** In this operation each processor sends a distinct message of size m to every processor. This operation takes $O((\tau + \mu m)p)$ time.

- **Transportation Primitive:** The transportation primitive performs many-to-many personalized communication with possibly high variance in message sizes. If the total length of the messages being sent out or received at any processor is bounded by t, the transportation primitive performs the communication using two all-to-all communications with a uniform message size of $\frac{t}{p}$ [88].

- **Sorting:** Using sample sort [59] in conjunction with bitonic sort for sorting splitters identified during sample sort, sorting n elements takes $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p\log^2

time for $n > p^2 \log p$.

Chapter 3

SPACE FILLING CURVES FOR ARBITRARY MULTIDIMENSIONAL POINTS

Space-filling curves are defined for points that lie on a uniform grid of a particular resolution. This makes it expensive to use them for mapping multidimensional points that are non-uniformly distributed. Further, the running time of algorithms using space-filling curves depends on the size of the grid, which is undesirable. For example, in adaptive grid-based methods used in many scientific applications, a small region of the domain needs fine refinement and this region usually moves with time. Using space-filling curves to keep track of such irregular distributions results in algorithms that depend upon the distribution of the grid cells.

In this chapter, we present a simple and efficient technique for ordering arbitrary and dynamic multidimensional points using space-filling curves. Our goal is to design

algorithms that are distribution-independent - i.e., the run times should only be a function of the number of points and their dimensionality, but independent of their distribution. We present an algorithm that orders n arbitrary d-dimensional points in optimal $O(dn \log n)$ time irrespective of the distribution of the points. Our algorithm is based on a comparison routine we devised, that compares two points and determines in $O(d)$ time which one is first visited by the space-filling curve. This not only makes the application of space-filling curves to arbitrary points efficient but also makes it possible to apply these strategies in the dynamic case. In particular, our technique can be used in conjunction with the well-understood algorithms and dynamic data structures for searching and sorting in one dimension. The techniques presented in this paper can potentially be used to derive distribution-independent algorithms for any application using space-filling curves.

To support our claims, we present applications of our technique in two different areas: databases and parallel computing. Storage and retrieval of records involving multiple keys is a challenging problem in databases. One of the methods, gaining popularity in the database community, is to use space-filling curves for this purpose. We present an efficient and distribution-independent storage structure using space-filling curves and show how to store n database records having d keys in $O(dn \log n)$ time using space-filling curves. Further, we present an algorithm for answering range queries using this storage structure. To the best of our knowledge, this is the first distribution-independent range query algorithm using space-filling curves.

3.1 Distribution Independent Ordering Of Multi-dimensional Points

Consider the problem of ordering n points in the order in which a particular space-filling curve visits them. We use square grids in two dimensions for ease of presentation. The same techniques can be easily extended to higher dimensions.

The traditional method for fitting a space-filling curve through a given set of points relies on the assumption that the coordinates of the points are representable using k-bit integers (for some fixed k). An index is calculated by interleaving the bits of these integers with the interleaving function of the space-filling curve. Sorting these indices gives the order of the points according to the space-filling curve. Although this method is very simple and easy to implement, it does not solve the problem for arbitrary points. If the ratio of the largest distance to the smallest distance of two points along a coordinate is greater than 2^k, the coordinate values of the points cannot be represented by k-bit integers. This is equivalent to saying that the points are not separable by a $2^k \times 2^k$ grid. Furthermore, the run-time depends upon the choice of k.

For arbitrary points, we adopt the following approach: We start with the root cell. We can continuously subdivide the root cell into grids of resolution $2^k \times 2^k$ (starting with $k = 0$ and increasing k) until no subcell on the grid contains more than one point. For the sake of concreteness, we can assume that the left edge and the bottom edge of a cell are included in the cell while the other two edges are not. Note that

once the space-filling curve enters a cell, it will leave the cell only after traversing all the points in the cell (if we imagine a space-filling curve for infinite resolution) and will never reenter the cell again. As each subcell contains at most one point, the order in which the curve visits the subcells also determines the order in which the curve visits the points. The problem with this approach is that the size of the grid required is dependent upon the distribution of the points and the resulting method does not have a bounded running time.

An easy improvement can be made to decrease the run-time of the above algorithm. Start with the root cell and subdivide the root cell into four immediate subcells, unless the root cell contains only one point. If a subcell contains more than one point, recursively subdivide it using the same method. This hierarchical subdivision is identical to the adaptive region quadtree [99]. An example of this procedure is shown in Figure 3.1. This modification may significantly improve the run-time in practice but it does not change the distribution-dependent nature of the algorithm.

Our goal is to design an algorithm that is efficient in practice and has a run-time that is a function of the number of points alone. Note that the order in which the space-filling curve visits two points is independent of the position and the presence of other points. Suppose that we are given two points with the goal of determining which of the two is first visited by the space-filling curve. We first determine the smallest subcell of the root cell that contains these two points. If we subdivide this subcell further into four immediate subcells, the two points must lie in different immediate subcells. Therefore, the order in which the space-filling curve visits the two points

Figure 3.1: Z-curve for a collection of 12 points using the hierarchical subdivision approach.

can be determined if we know the order in which the four immediate subcells of the smallest subcell containing the two points is visited by the curve. For the Z-curve, this order is always subcell III, subcell II, subcell IV and subcell I, irrespective of the position and the size of the smallest subcell containing the two points.

The procedure we adopt to order n points according to a space-filling curve is as follows: We devise a comparison routine that takes two points as input and determines which of the two is first visited by the space-filling curve. This is done by first determining the smallest subcell of the root cell that encloses the two points (a constant time operation using the above procedure) followed by determining the order in which the curve visits the four immediate subcells of this subcell. For the Z-curve

this order is trivial and independent of the size or location of the subcell enclosing the points. For other curves, this may not be so straightforward or even possible. Once we design such a comparison routine, we can use any sorting algorithm to order the points. In the following, we describe such a technique for the Z-curve and graycode curve.

Z-curve. The Z-curve visits the immediate subcells of a cell in the order subcell III, subcell II, subcell IV and subcell I (see Figure 2.3). Once we have the smallest subcell and the position of two points in that subcell, we can immediately find the order of these points on the resulting Z-curve. For example, if one of the points is in subcell I and the other is in subcell III, the point in subcell III is visited before the point in subcell I.

Graycode curve. The order in which the graycode curve visits the immediate subcells depends on the location of the cell in the grid it belongs to. Fortunately, there are only two different orders possible: subcell III, subcell II, subcell I, and subcell IV or subcell I, subcell IV, subcell III, and subcell II. We refer to the former as $\cap$-order and the latter as $\cup$-order. Each row of cells in a grid has a unique order. If we number the rows in a grid bottom-up, the $\cap$-order can only be found at the odd numbered rows and even numbered rows have $\cup$-order. To compare two points in graycode order, we check the location of the smallest cell. If it is on one of the odd numbered rows, the comparison is performed using the $\cap$-order and $\cup$-order is employed otherwise.

A similar comparison routine for the Hilbert curve seems to be more complicated

and may be impossible in constant time. As in the graycode curve, the order of the immediate subcells of a cell in the Hilbert curve depends on the location of the cell. However, this dependence is not as simple. While in the graycode curve the order depends on only the last bit of the location of the cell along the y-axis, all the bits of the location of the cell along both axes is necessary to find the order for the Hilbert curve. This is because there is only one kind of rotation in the generation of graycode curve and the effect cancels out for every two rotations. On the other hand, there are two types of rotations in the Hilbert curve and the cancellations are not so trivial.

The comparison routines described above can be easily extended to d-dimensional space. In this case, the cells are d-dimensional hypercubes and the k-boundaries are $(d-1)$-dimensional hyperplanes perpendicular to an axis. For the smallest cell containing two points p_1 and p_2, we need to check each axis in turn and find the smallest integer s such that a s-boundary intersects the hyperrectangle R defined by p_1 and p_2. This operation takes $O(d)$ time. Immediate subcells of the smallest cell containing the points and the order of the immediate subcells can be determined in $O(d)$ time. Thus, comparison of two points is accomplished in $O(d)$ time. By using any efficient sorting algorithm, the time required to order arbitrary n points in the order in which the Z-curve or the graycode curve visits them is $O(dn \log n)$.

3.2 Applications

3.2.1 Storing Database Records and Range Queries Using Space-filling Curves

Efficient storage and retrieval algorithms are essential for ensuring the performance of database applications. Efficiency of querying a database depends on the storage organization and the data structure (i.e., storage structure) used. Algorithms for storing and retrieving records using a single key have been extensively studied by many researchers and a large number of good storage structures developed [26]. However for records involving multiple keys, the solutions devised are not as satisfactory. Some of the techniques developed are a natural extension of the single-key based storage structures and do not necessarily perform well for queries involving multiple keys. Some storage structures make querying efficient when using some subset of the keys, at the expense of slowing down queries when other keys are involved.

The problem of storing database records with multiple keys is equivalent to the problem of organizing points in a multidimensional space. A file (or a relation) in a database is a collection of records with each record containing d key fields (or key attributes) and other fields (attributes) that are never used to query the database. We can represent each such record by a point in d-dimensional space by assigning a dimension to each key field and specifying a mapping from key values to coordinates. The records have to be stored on a secondary storage medium such as a disk. This

requires mapping the d-dimensional points to the one-dimensional range of block addresses on the disk. The cost of a database query is dominated by the disk access time. It is more economical to retrieve consecutive blocks on a disk instead of scattered blocks (reduces seek time overhead). Therefore, to reduce the number of block accesses and to keep the block accesses consecutive, the mapping should also preserve the proximity information in d-dimensional space.

Storing of database records using space-filling curves is first suggested by Orenstein et. al. [75]. The basic idea is to map the multidimensional data to one dimension using a space-filling curve and to employ any one-dimensional storage structure (e.g. B-tree and its variants [26]) afterwards. Since efficient storage structures are available in one dimension, it is supposed that the resulting multidimensional storage structures are also efficient. However, interleaving the bits of the keys has been used for mappings based on space-filling curves and this method relies on the granularity of the grid using which the curve is defined. For keys that result in non-uniform distributions, these storage structures depend not only on the size of the data but also on the grid size in finest granularity. Thus, the running times are dependent on the distribution and are in fact unbounded for arbitrary distributions.

We use the aforementioned technique for storing database records with multiple keys using space-filling curves. We first map each key field to a coordinate and view the records as points in k-dimensional space. We then order these points in the order in which a particular space-filling curve visits them by using the proposed comparison routines and store the records according to this order. The method allows us to use

```
Range_Search ( query_region, search_cell )

for each immediate subcell of the search_cell
    search_subcell = the immediate subcell
    query_subregion = query_region ∩ search_subcell
    if query_subregion ≠ Ø
        if query_subregion = search_subcell
            report all the points in search_subcell
        else
            Range_Search ( query_subregion, search_subcell )
```

Figure 3.2: A simple algorithm for range queries.

any one-dimensional storage structure efficiently for storing multidimensional data without using bit-interleaving algorithms. Since the comparison routine takes $O(d)$ time, the storage structure can be implemented in optimal $O(dn \log n)$ time. Further, the size of the resulting storage structure is distribution-independent.

Range queries are the most commonly encountered queries in database applications. In a range query, ranges are specified for all or a subset of the keys with the objective of retrieving all the records that satisfy the range criteria. In our mapping of records to multidimensional space, the ranges correspond to a (possibly open-ended) hyperrectangle. The objective is to retrieve all the points that lie in this hyperrectangular region (i.e., query region).

Orenstein et. al. [75] present some heuristics for reducing the run-time for range queries. While these heuristics may be useful in practice, they do little to remove the distribution-dependent nature of the algorithm. More importantly, it cannot even be

guaranteed that the application of a heuristic does not, at least, increase the run-time. To the best of our knowledge, all the algorithms using space-filling curves that are reported in the literature are distribution-dependent.

A simple algorithm to evaluate a range query based on a storage structure employing space-filling curves is to check the cells and their immediate subcells recursively for the query region starting with the whole domain. For each immediate subcell of a cell, a query subregion is calculated by intersecting the query region with the subcell. There are three possibilities: the query subregion could be empty, the whole subcell or a part of the subcell. If the query subregion is the whole immediate subcell, we report all the points in the subcell. Note that a space-filling curve enters a subcell once and exits it only after visiting all the points in it. Therefore, all the points within a subcell are stored consecutively, making it easy to access them. If the immediate subcell does not intersect with the query subregion, nothing further needs to be done. Otherwise, we generate separate range queries with the immediate subcells as search regions and associated query subregions. Pseudo code for this algorithm is given in Figure 3.2.

To report the data points in a cell, we search the entrance point of the space-filling curve into the cell in the database. This point depends on the order in which the immediate subcells of a cell are visited by the space-filling curve. For example, the first immediate subcell of a cell visited by the Z-curve is the lower left subcell of the cell. Therefore, the entrance point of the Z-curve in a cell is its lower left corner. Once we find the first point in the cell, we report successive points until the curve

Figure 3.3: A two-dimensional domain and a query region R. The different numbered boundaries are shown using different styles of lines. The largest dashed lines are 1-boundaries and the smallest dashed lines are 4-boundaries.

crosses the boundary of the cell.

The range query algorithm described above has two major drawbacks:

- The query region is often a very small part of the whole domain. Consider the case where the smallest boundary passing through the query region is a j-boundary, for some j. Since each recursive call divides a cell along the next boundary, the recursion should continue from 1 to j to get the first hit on the query region. The effort spent until this point is wasted. In Figure 3.3, the smallest boundary passing through the query region R is a 4-boundary. Hence, the calls to divide the search region along 1, 2 and 3-boundaries are all only to

Figure 3.4: A query region and a point P close to its boundary. To determine that P is not in the query region, the search region should be broken into very small cells.

find the 4-boundaries that divide the query region. We gave a small example for illustration here, but the size gap between the query region and the search region could be arbitrarily large.

This case does not happen only at the initiation of a range query. We divide the query region among the immediate subcells of a cell in each recursive call. Even if the query region is spread over more than one immediate subcell of a cell, this may not be the case for the immediate subcells of the immediate subcells.

- If there are points very close to the boundaries of the query region, it may be very expensive to check if these points are in the query region, using this algorithm. We divide the cell containing this point at the boundary into finer and finer

granularities to find subcells that do not intersect but are completely inside or outside of the query region. In Figure 3.4, we have to divide the search region until we separate the query region and the point with the boundaries shown with dotted lines.

While the first problem is the result of the position and size of the query region, the second problem depends on the distribution of the data surrounding the query region. However, the effect is the same in both cases – unbounded running time.

We now present an algorithm that eliminates large portions of the search region and at the same time guarantees bounded running time. The following terminology is to be used in understanding the algorithm: A *search cell* is a cell that contains the query region. During the execution of the algorithm, a search cell may be split into immediate subcells called *search subcells* and the query region is also split into *query subregions* accordingly. Therefore, each search subcell has a corresponding query subregion that it contains. For each search cell we keep track of a *search segment*, which is a continuous segment of the space-filling curve that contains points within this search cell. The search segment may not contain all the points in the search cell, but the algorithm ensures that it contains all the unreported points in the query region associated with the search cell. *Search subsegments* are defined similarly.

We solve the first problem by directly finding the smallest subcell of a search cell that contains the query region of interest ($O(d)$ time for d-dimensional data), each time before dividing a search cell into search subcells. Since we divide the smallest

cell that contains the query region, at least two search subcells should contain some part of the query region. This guarantees that each call divides the query region as well.

The number of search cells generated in this process need not be bounded and result in a distribution-dependent algorithm. This problem cannot be eliminated by focusing on search/query regions alone. To solve this problem, we ensure that at least one point that is in the search segment but not in the query region is eliminated (if such a point exists) before a search cell is subdivided into search subcells. We do this by examining the search segment sequentially and reporting all the points that are in the query region until we find the first point that is not in the query region. We then find the largest cell that contains this point but does not intersect the query region. All the points in this cell can be skipped without any loss of information. The largest cell can be found as follows: Draw the straight line that gives the shortest distance from the point to the query region. Calculate the smallest numbered boundary that crosses this line (i.e separates the point and the query region) along each axis. Of course, if the line is perpendicular to an axis, none of the boundaries along this axis could separate the point and the query region. The cell can be inferred from the smallest of these boundary numbers and the point.

The pseudo code of the algorithm using the techniques explained above is given in Figure 3.5. We call the **Range_Search** routine with three parameters: query region, search cell and search segment. We break the search cell into search subcells and split the query region into query subregions, one for each search subcell as in the simple

```
Range_Search ( query_region, search_cell, search_segment )

for each immediate subcell of the search_cell
    search_subcell = the immediate subcell
    query_subregion = query_region ∩ search_subcell
    if query_subregion ≠ Ø
        search_subcell = smallest subcell of search_subcell containing query_subregion
        search_subsegment = segment of search_segment containing the search_subcell
        while the first point in the search_subsegment is in the query_subregion
            report the point
        if search_subsegment ≠ Ø
            cell = the largest cell that contains the first point in the search_subsegment
                   but does not intersect query_subregion
            drop the segment for the cell from the search_subsegment
            adjust search_subcell and query_subregion to conform with the search_subsegment
            Range_Search ( query_subregion, search_subcell, search_subsegment )
```

Figure 3.5: A distribution-independent algorithm for range queries.

algorithm. However, we do not make recursive calls right away but try to eliminate as much of each search subcell and the associated search subsegment as possible. First, we reduce each search subcell by finding the smallest subcell of it that still contains its query subregion. The search subsegment for each search subcell is the part of the search segment that lies in the search subcell. We then traverse the points in the search subsegment and report them until the first point not in the query subregion is found. We reduce the search subsegment by eliminating the largest cell that is outside of the query subregion, contained in the search subcell, and containing this point. **Range_Search** is now called recursively with this reduced search subcell and

its associated query subregion and the reduced search subsegment.

The worst case running time of this algorithm is $O(n)$. Previous algorithms reported in the literature that use space-filling curves have run-times that are not bounded as a function of the number of points. Our algorithm guarantees a bounded running time and also performs better than existing algorithms because it simultaneously reduces the search segment to focus on and eliminates parts of search cells that do not intersect the query region. Of course, a naive algorithm that simply scans through the points and decides which are in a query region also runs in $O(n)$ time. While such an algorithm would always require time proportional to the total number of points irrespective of the query, the **Range-Search** algorithm we presented is expected to perform well in practice.

3.2.2 Experimental Results

To verify the distribution-independent nature of the aforementioned algorithm and to show the algorithm performs well in practice, we performed several experiments in two dimensions using the Z-curve. We tested our algorithm with both uniformly and non-uniformly distributed points. The domain of data points is the unit square in all experiments. We used multiplicative congruential random number generator for uniform distribution and generate points that are uniformly distributed in both dimensions. As a non-uniformly distributed sample, we chose normal distributions with mean values of 0.7 and 0.3 and standard deviations of 0.15 and 0.2 along x and y-axis, respectively. For both distributions, we generated $10,000$ data points in a unit

Uniform distribution Non-uniform distribution

Figure 3.6: Uniformly and non-uniformly distributed data in Z-order.

square. The results of fitting Z-curves for our uniform and non-uniform samples are shown in Figure 3.6.

The query regions were also generated randomly. We generated 100 random query regions and positioned each of them at 100 randomly chosen locations. The lengths of the query regions along the x-axis were uniformly distributed between 0 and 0.5. The lengths along the y-axis were determined using an aspect ratio (length along the x-axis / length along the y-axis). For each randomly chosen query region, we tried 21 different aspect ratios: $1/10, 1/9, ..., 1, 2, ..., 10$. This ensures that many different sizes and shapes of query regions were generated. The locations of query regions were uniformly distributed over all possible locations in the unit square.

The performance of a query operation is mainly determined by the number of data blocks accessed in comparison to the number of data blocks actually needed to answer the query. To normalize the number of blocks accessed and place the results on a common scale, we present performance results using percentage of useful blocks among all the blocks accessed. We also use the percentage of points in the query region among all the points accessed as a performance measure.

Number of data blocks accessed to answer a range query may depend on the following five factors:

- number of points in the query region
- area of the query region
- distribution of points
- shape of the query region
- number of points in a block (i.e., block size).

We chose the first two factors as variants in our experiments and show the performance of the algorithm as the number of points in query regions or area of query regions change. For each experiment, we collected the information about performance measures, the fraction of useful points and blocks accessed. To eliminate minor fluctuations we averaged the performance results in small ranges (e.g. interval of 10 points).

In Figure 3.7, we plot the results for uniform and non-uniform distributions. The aspect ratio is 1 (i.e., square query regions) and block size is 10 for this experiment.

Uniform distribution Non-uniform distribution

Figure 3.7: Number of points in the query region and area of the query region versus percentage of useful points and blocks for uniform and non-uniform distributions.

We observe a rapid increase in the performance with the increase in the number of points and the area for either distribution. There are small fluctuations but these can be attributed to the random size and positioning of the query region. For the non-uniform distribution, even when the number of points is less than 10, the percentage of blocks that are actually needed is more than 50% and the percentage of points in the query region is more than 40%. The percentages increase very rapidly: percentage of useful blocks crosses 90% before 1000 points. These figures show that the algorithm performs very well even for non-uniform distributions.

When we compare the graphs in Figure 3.7 pairwise, the performance of the algorithm for non-uniform distribution is unexpectedly a little better than the one for uniform distribution. This is mainly because of the fact that the concentration of points is more at some places for non-uniform distribution and the queries containing these places dominates in the calculation of the average for the intervals. For example, the same query region may include much more points in non-uniform distribution.

We illustrate the effect of different aspect ratios on the performance measures in Figure 3.8 and 3.9. We show the results for aspect ratios of 1, 1/3, 1/6, and 1/10 and non-uniform distribution (the results for uniform distribution are not shown but very similar). As expected, there is a small decrease in the performance while the aspect ratio gets smaller. However, the decrease is not very crucial: even for an aspect ratio of 1/10, more than 65% of the blocks accessed are useful to answer queries with 50 points. The decrease in the performance is about 10% from an aspect ratio of 1 to an aspect ratio of 1/10. We got similar results for aspect ratios of $1, 2, ..., 10$, which

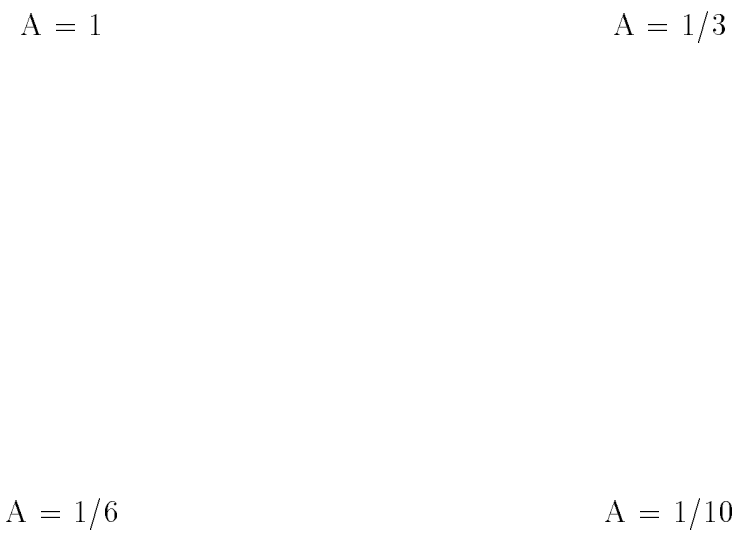

Figure 3.8: Number of points in the query region versus percentage of useful points and blocks for various aspect ratios (A = width/length) for the non-uniform distribution.

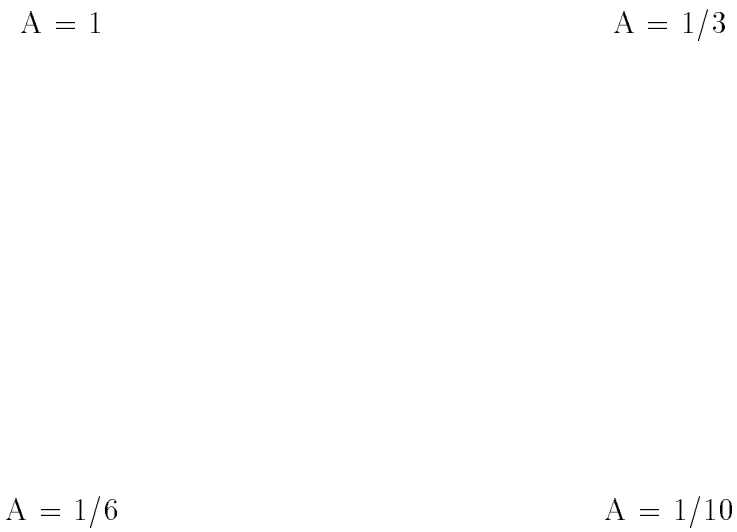

Figure 3.9: Size of the query region versus percentage of useful points and blocks for various aspect ratios (A = width/length) for the non-uniform distribution.

Figure 3.10: Number of points in the query region and size of the query region versus percentage of useful points and blocks for various block sizes (number of points in a block).

are not shown here.

The effect of block size on the performance of the algorithm is also checked for both distributions. The results for block sizes of 5, 10, and 20 for non-uniformly distributed points are shown in Figure 3.10. It is almost impossible to distinguish the graphs for different block sizes. This shows that the block size has very little effect on the performance of the range query after some reasonable block size such as 5. Similar results are also observed for the uniform distribution.

3.2.3 Parallel Domain Decomposition Methods

Parallelization of applications requires distribution of the workload to processors. In many scientific applications, this is accomplished by decomposing the computational domain. Because of their low running time, decomposition techniques using space-filling curves are a good alternative [77, 84, 111]. In this section, we present a distribution-independent domain decomposition algorithm using space-filling curves along with experimental results on the Intel Paragon.

Space-filling curve based domain decomposition methods are useful when there is interaction between proximate entities of the domain. In such cases, communication is reduced by keeping proximate entities together in the same processor. For example, in simulations involving particles such as N-body methods and molecular dynamics, frequent interactions occur between neighboring particles and interactions between faraway particles are either ignored or occur infrequently. In such an application, the particles can be sorted according to a space-filling curve and assigned to processors. In grid-based methods, the grid-cells can be ordered according to a space-filling curve and the grid-cells can be assigned to processors based on this order.

A particularly useful instance of domain decomposition is the graph partitioning problem. In this, entities are represented by vertices and the interaction between entities is explicitly shown by edges. Many scientific and engineering applications involving iterative methods can be represented by computational graphs [77]. Typically, the nodes of the graph correspond to two- or three-dimensional coordinates

and the edges are limited to vertices that are physically proximate. Examples of such applications include finite element methods, PDE solvers and molecular dynamics simulations. To parallelize the application on a collection of p processors, the computational graph should be partitioned to the p processors.

There is currently significant interest in the research community for using space-filling curves for graph partitioning. Ou [78] and Pilkington et al. [84] have shown that space-filling curve based partitioning (SBP) techniques provide a partition of quality comparable to Recursive Coordinate Bisection. Further, it beats Recursive Coordinate Bisection in running time for almost all experiments. Although experimental results show that SBP performs well, the running time depends on the distribution of the data.

To effect parallel domain decomposition on p processors, we adopt the following strategy: We first sort the data along the space-filling curve using a standard sorting algorithm. The resulting sorted data is cut into p equal parts and distributed to the p processors. Any parallel sorting algorithm can be used in conjunction with the comparison routines designed in Section 3. Parallel sorting also has the desirable side effect of distributing the sorted data. Hence, a separate distribution phase is not necessary.

It is also very important to be able to determine the processor a given data point is allocated to. The comparison routines can be used to solve this problem as well. We can use an array of length p to store the coordinates of the first point stored on each processor, i.e., processor boundaries. By a simple binary search on this array

Graph 1 Graph 2 Graph 3

Figure 3.11: Non-uniform computational graphs and their ordering according to the Z-curve. Graph 1 has 6019 nodes, Graph 2 has 10166 nodes and Graph 3 has 15606 nodes.

using the prescribed comparison routine, the processor containing a given data point can be determined in $O(d \log p)$ time for d-dimensional data.

We implemented the space-filling curve based partitioning algorithm described above on an Intel Paragon. The algorithm was implemented in C with calls to the MPI message-passing library functions. We employed parallel sample sort algorithm because of its practical efficiency and scalability [59] and chose the Z-curve. We tested

Number of Nodes	Graph size (# of vertices)					
	6019		10166		15606	
	non-uniform	uniform	non-uniform	uniform	non-uniform	uniform
2	19.32	29.28	75.25	102.60	87.44	176.71
4	6.33	6.53	11.27	10.12	19.09	15.66
8	2.44	2.19	4.61	3.92	7.79	6.54
16	1.20	1.27	2.29	2.30	3.50	3.47
32	1.91	1.74	2.08	2.43	2.78	3.43

Table 3.1: Parallel partitioning time (in seconds) for uniform and non-uniform graphs of different sizes.

our implementation with three different graph sizes and two types of distributions for each size: uniform and non-uniform. The uniform distributions were generated using a uniform random number generator. The graphs chosen for the non-uniform distributions were taken from Ou's dissertation [77] and shown in Figure 3.11. The run times of parallel domain decomposition for different numbers of processors are shown in Table 3.1. The times are in seconds and are averaged over 10 runs.

The main purpose of this experiment is to check the running times for same graph sizes but different distributions. Apart from minor fluctuations attributable to the random nature of sample sort, the running times do not differ from uniform distribution to non-uniform distribution. This demonstrates that our algorithm is independent of the distribution of the data. Note that there is a super-linear speedup for small processor sizes. This anomaly is because sample sort algorithm does not perform well for very small number of processors. The sample selected does not accurately reflect the characteristics of the data.

3.2.4 Parallel Incremental Load Balancing

Many scientific simulations requires the computational graph to change with time. Although these changes are very small compared to the whole graph, they may cause imbalance in the workload. Since load imbalance is one of the reasons of degrading performance of applications, the graph should be redistributed to balance the workload when there are changes. It is very expensive to redistribute the graph from scratch each time when there is a change. There are two popular approaches to solve this problem: In the first approach, load distribution is performed from scratch after the changes exceed a threshold value. In the second approach, the previous domain decomposition information is used to improve the performance of the load balancing algorithm, i.e., incremental load balancing is performed. In what follows, we discuss the use of the proposed comparison techniques in incremental load balancing.

Space-filling curve based partitioning has two phases: sorting the multidimensional data and partitioning the resulting order to processors. After the changes in the graph, the data may not remain sorted in the order induced by the space-filling curve. If the changes are local to each processor, i.e., the changes are within the boundaries of the processor, readjusting the graph locally in each processor is sufficient. However, there may be vertices that go beyond processor boundaries. The idea is to determine the processors responsible for such vertices and send them to the appropriate processors so that local readjustments are enough. For this purpose, we can use a technique similar to the one described in the previous subsection. We sort all the vertices that need

to be sent to other processors and find the positions of the processor boundaries on this sorted array using binary search. For the sort and search operations, we employ the proposed comparison routines. After the processors responsible for the vertices are determined, an all-to-all communication is sufficient to exchange them. At this point, all the vertices within the processors are local. We merge the vertices locally within the processors to achieve the global order. Once we get the multidimensional graph sorted again, we can partition it again to balance the workload. This could be accomplished by computing a parallel prefix on the number of points in each processor and shifting the excess points to neighboring processors. Since space-filling curves preserve locality, the changes are mostly local. The number of vertices that need to be exchanged between processors is expected to be small. Thus, the cost of incremental load balancing is expected to be small.

Chapter 4

DISTRIBUTION INDEPENDENT ADAPTIVE TREES

The DIAT tree is essentially a compressed hyperoctree with some additional information attached to it to enable fast execution of queries, insertions and deletions.

At each node v in the DIAT tree we store two cells, *large cell of* v and *small cell of* v denoted by $L(v)$ and $S(v)$, respectively. The large cell is the largest cell that the node is responsible for and the small cell is the smallest subcell of the large cell that contains all the points within the large cell. The large cell of the root node is a cell large enough to contain all the points. Consider any internal node v in the DIAT tree. To obtain its children, we subdivide $S(v)$ into 2^d equal-sized immediate subcells. Note that the subdivision of $S(v)$ results in at least two non-empty immediate subcells. For each non-empty immediate subcell C resulting from the subdivision, create a child u such that $L(u) = C$. Therefore, $L(u)$ at a node u is an immediate subcell of $S(v)$

at its parent v. A node is a leaf if it contains a single point and the small cell of a leaf node is the hypothetical cell with zero length containing the point. The internal nodes keep cell information and point information is stored in the leaves.

The children of a node in a DIAT tree are ordered according to a fixed space-filling curve order of the large cells of the children. This ordering results in a unique ordering of the nodes as follows: If the large cells of two nodes are disjoint the order of the cells is same as the order of the nodes. Otherwise, one of the large cells contains the other and the node representing the larger large cell comes after theother node. The order of the leaf nodes also corrsponds to the order of the points repsented by the leaf nodes according to space-filling curve used. This ordering of nodes is very important for designing efficient algorithms.

DIAT trees can be searched in a manner similar to binary search trees. For example, to locate a point, start at the root and traverse to a leaf by repeatedly branching to the child containing the point. Such algorithms are not efficient on DIAT trees because the height of a DIAT tree is $O(n)$ in the worst-case. For speeding accesses and for efficient maintenance in the dynamic case, we equip the DIAT tree an auxiliary data structure on its nodes.

The auxiliary data structure is a balanced binary search tree (abbreviated BBST) built on the nodes of the DIAT tree. The BBST is used to access the nodes of the DIAT tree without navigating from the root to the leaf. Each node in the BBST contains a pointer to the corresponding node in the DIAT tree. To compare two nodes in the BBST, we use the order of the small cells of the nodes according to the

Figure 4.1: The hyperoctree and the DIAT tree for 10 points in a two dimensional domain. The positions of the points in the domain are shown on the left. The square regions represents the subdivision of the domain in the construction of the hyperoctree. The resulting hyperoctree is shown at the top right. The corresponding DIAT tree is at the bottom right. The pointers from the leaves to the internal nodes of the DIAT tree are shown but the auxiliary data structure on the leaves is not shown.

space-filling curve tree. During the process of ordering two nodes, essentially, we are computing the least common ancestor of the nodes and ordering the large cells of two children of the least common ancestor. The relative order of two nodes in a DIAT tree can be determined in $O(1)$ time.

Lemma 4 *DIAT-size The size of a DIAT tree for n points is $O(n)$.*

Proof: Let $S(n)$ represent the number of nodes in the DIAT tree. Note that each internal node has at least two and at most 2^d children. Also, the number of points contained in a node is the sum of the number of points contained in its children. Thus,

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

$$\sum_{i=1}^{k} n_i = n$$

$$S(1) = 1$$

These equations are satisfied by $S(n) \leq 2n - 1$, and the number of nodes is bounded by $O(n)$. Since, there is only one pointer from each leaf node, the size of the DIAT tree is $O(n)$. ■

4.0.5 Implementation Issues

Unlike octrees, DIAT trees can be implemented using arrays because the number of nodes in a DIAT tree is bounded by $2n - 1$ for n particles. Each element in the array

represents a node in the DIAT tree and the BBST. For the internal nodes of the DIAT tree, we store only the small cell information, i.e., the length and the the coordinates of a corner of the cell. The large cell information can be obtained by subdividing the small cell of the parent node.

For static particles, the BBST can also be implicitly represented by using its inorder traversal, which corresponds to the sorted order. In a dynamic environment, implementation on an array should allow insertions and deletions. To facilitate this on an array, a list of free nodes that are not being used by the DIAT tree is maintained. For insertions, we draw an element from the list and in the case of deletion, we return the deleted node to the list.

The implementation of BBST requires an algorithm to compare two cells. We can use the definition of the order of two cells above but an algorithm to find the smallest cell containing both the cells is needed. To do this, we associate a key with each cell based on its length and position and compare cells using these keys. The key for a cell is the concatenation of strings of the form xyz used for the immediate subcell at the subdivisions from the root to the cell. We add a leading 1 to this key to get a unique key for each cell. For example the cell D in Figure 2.1 has the key 10100 and the cell H has the key 1000100. Notice that, after removing the leading 1, the key for D is a suffix of the key for H. This is true whenever there is subcell-supercell relation between the nodes. Thus, we find the largest common suffix of the keys of the two cells and add the leading 1 at the beginning of the suffix to obtain the key for the smallest cell. The order of the two cells can be determined by comparing the

three bits just before this suffix, if one is not a subcell of the other.

4.1 Building the DIAT Tree

In this sections, we give three algorithms to build the DIAT tree in $O(n \log n)$ time. While all three algorithms achieve the same time-bound and are optimal, understanding the first two is necessary as the ideas developed in them are needed to make the DIAT tree dynamic. After building the DIAT tree, the pointers from leaves to internal nodes are assigned and the BBST is built for searches and dynamic operations.

Smallest cell technique can be used for a recursive construction of the DIAT tree starting from the root. Given a node v and $L(v)$, let l_i (respectively, h_i) be the smallest (respectively, largest) coordinate along dimension i of any point in $L(v)$. Let $R_v = [l_1, h_1] \times [l_2, h_2] \times \ldots \times [l_d, h_d]$. The smallest subcell containing all points in $L(v)$ must also contain R_v. Thus, $S(v)$ can be computed from $L(v)$. Recursively construct the subtree under v by subdividing $S(v)$ into its immediate subcells. Scanning all points in $L(v)$ to determine R_v takes time proportional to $size(L(v))$, making the worst-case run-time of the algorithm $O(n^2)$.

4.1.1 A Divide-and-Conquer Algorithm

Let T_1 and T_2 be two DIAT trees representing two distinct sets S_1 and S_2 of points. Let r_1 (resp., r_2) be the root node of T_1 (resp., T_2). Suppose that $L(r_1) = L(r_2)$, i.e., both T_1 and T_2 are constructed starting from a cell large enough to contain $S_1 \cup S_2$.

A DIAT tree T for $S_1 \cup S_2$ can be constructed in $O(|S_1| + |S_2|)$ time by merging T_1 and T_2.

To merge T_1 and T_2, we start at their roots and merge the two trees recursively. Suppose that at some stage during the execution of the algorithm, we are at node u in T_1 and at node v in T_2 with the task of merging the subtrees rooted at u and v. An invariant of the merging algorithm is that $L(u)$ and $L(v)$ cannot be disjoint. Furthermore, it can be asserted that $L(u) \cap L(v) \supseteq S(u) \cup S(v)$. In merging two nodes, only the small cell information is relevant because the small cell contains all the particles of a node and the rest of the large cell $(L(v) - S(v))$ is empty. For convenience, assume that a node may be empty. If a node has less than 2^d children, we may assume empty nodes in place of the absent children. Four distinct cases arise as shown in Figure 4.2. Cases I and III admit symmetric cases with the roles of u and v interchanged and can be handled similarly.

- Case I: If there is no point in the large cell corresponding to one of the nodes (i.e. the node is empty), the result of merging is simply the tree rooted at the other node.

- Case II: If the small cells $S(u)$ and $S(v)$ of u and v represent the same region (See Figure 4.3), the corresponding children of u and v have the same large cells which are the the immediate subcells of $S(u)$ (or equivalently, $S(v)$). In this case, we simply merge these children one by one.

- Case III: If the small cell of one node strictly contains the other, one of the

Algorithm 1 MergeTree (u,v)

CASE I: u is empty tree

Return v

CASE II: $S(u) = S(v)$

Create a new node w

For all associated children u_i and v_i of u and v (i.e., $L(u_i) = L(v_i)$)

$w_i =$ **MergeTree** (u_i,v_i)

$S(w) = S(u)$; $L(w) = L(u) \cap L(v)$

Return w

CASE III: $S(v) \subset S(u)$

Find a child u_i of u that can be associated with v such that $L(u_i) \supseteq S(v)$

$u_i =$ **MergeTree** (u_i,v)

Return u

CASE IV: $S(u)$ and $S(v)$ are disjoint (i.e. $S(u) \cap S(v) = \emptyset$)

Create a new node w

Find the smallest cell b that contains $S(u)$ and $S(v)$

$S(w) = b$; $L(w) = L(u) \cap L(v)$; $w_i = u$; $w_j = v$

Set $L(u)$ and $L(v)$ to corresponding subcells of $S(w)$

Return w

Figure 4.2: Algorithm to merge two DIAT trees rooted at u and v. Note that to merge two nodes, the large cells of both u and v should be subcells of the same root cell and $L(u) \cap L(v) \supseteq S(u) \cup S(v)$. We use subscripts to represent the children of a node.

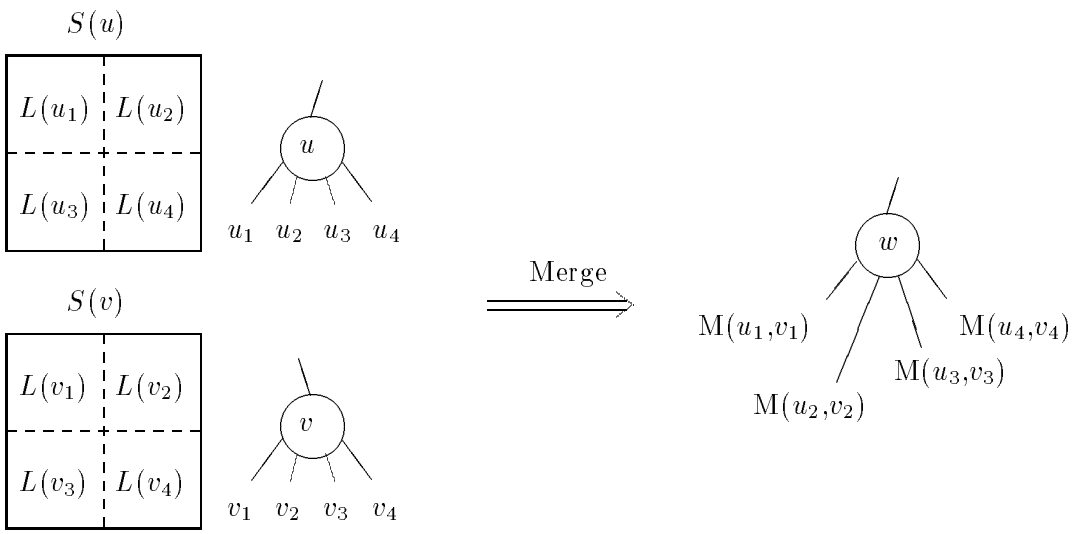

Figure 4.3: Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are equal. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The resulting tree is shown on the right. The node M(u_i,v_i) is the result of merging u_i and v_i.

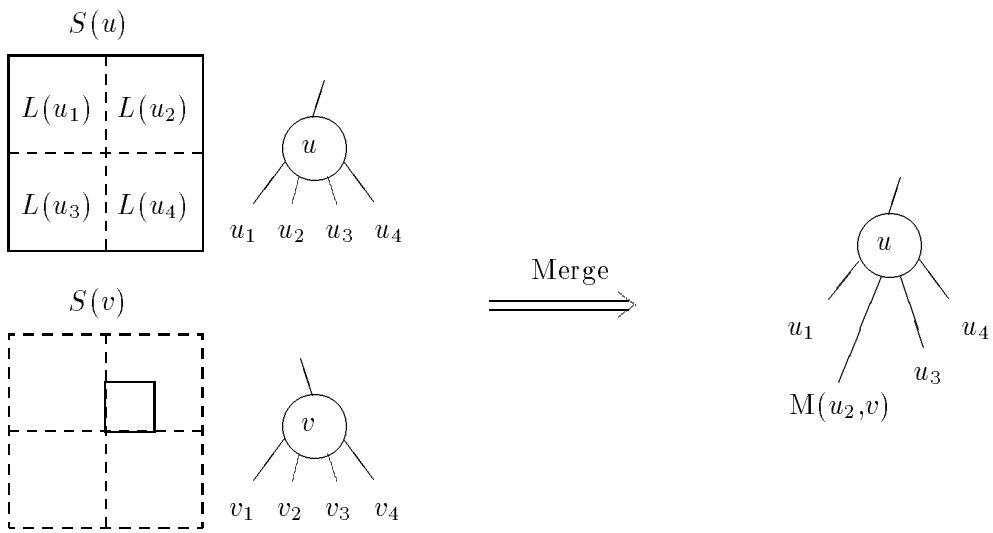

Figure 4.4: Merging subtrees rooted at u and v when $S(u)$ contains $S(v)$. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right. The node $\mathrm{M}(u_i,v_i)$ is the result of merging u_i and v_i.

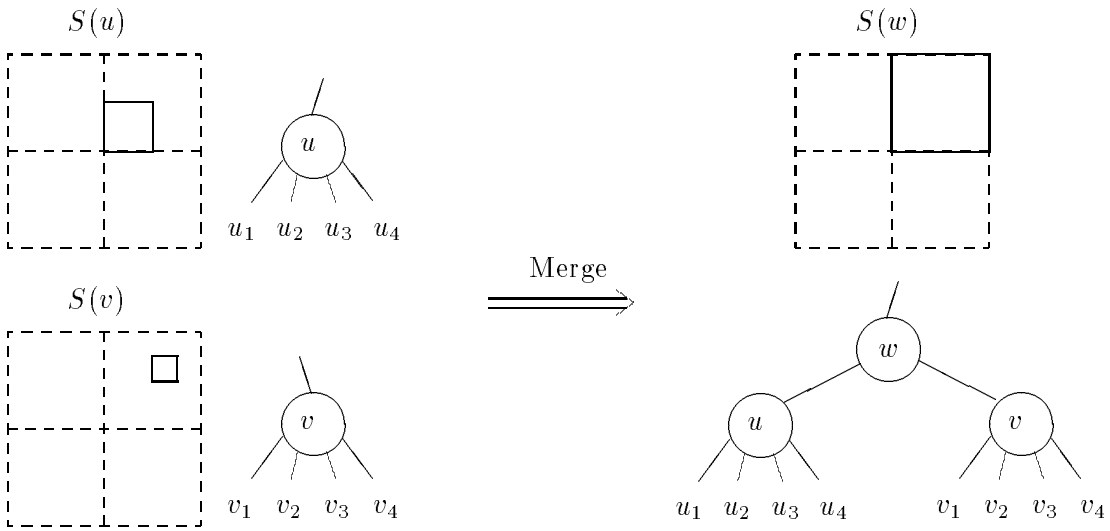

Figure 4.5: Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are disjoint. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right.

immediate subcells of the small cell is equal to or contains the other (See Figure 4.4). Assume that $S(u)$ contains $S(v)$. The particles in v are also in the large cell of one of the children of u. Thus, this child of u is merged with v.

- Case IV: If the small cells at the nodes are disjoint, i.e., $S(u) \cap S(v) = \emptyset$ then the smallest cell containing both $S(u)$ and $S(v)$ has $S(u)$ and $S(v)$ in different immediate subcells (See Figure 4.5). If they both were to lie in the same immediate subcell, this immediate subcell would have been smaller than the smallest cell containing them, a contradiction. In this case, we create a new node in the merged tree with its small cell as the smallest cell that contains

$S(u)$ and $S(v)$ and simply make u and v the two children of this node. Subtrees of u and v are disjoint and need not be merged.

Lemma 5 *merging Two DIAT trees with a common root cell can be merged in time proportional to the sum of their sizes.*

Proof: The merging algorithm presented above performs a preorder traversal of each DIAT tree. The whole tree may not need to be traversed because in merging a node, it may be determined that the whole subtree under the node directly becomes a subtree of the resulting tree. In every step of the merging algorithm, we advance on one of the trees after performing at most a constant amount of work. Thus, the running time is bounded by the sum of the sizes of the trees to be merged. ■

To construct a DIAT tree for n points, scan through the points and find the smallest and largest coordinate along each dimension. Find a cubic region that contains all the points and use this as the root cell of every DIAT tree constructed in the process. Recursively construct DIAT trees for $\lfloor \frac{n}{2} \rfloor$ points and the remaining $\lceil \frac{n}{2} \rceil$ points and merge them in $O(n)$ time. The DIAT tree for a single point is a single node v with the root cell as $L(v)$. The algorithm is similar to mergesort, and takes $O(n \log n)$ time.

4.1.2 Bottom-up Construction

To perform a bottom-up construction, first compute the ordering of the points as they should appear in the DIAT tree and compute the tree bottom-up using this order.

Algorithm 2 Bottom-up Construct (points)

Find the root cell

Sort the points according to the DIAT tree

Create one node DIAT tree rooted at v from the first point p in the sorted list

While there are more points to insert

Create a new node w for the next point p

While p is not in $L(v)$

$v = parent(v)$

If $p \notin S(v)$

Calculate the smallest subcell C of $L(v)$ containing p and $S(v)$

Create a new node u in the place of v

$L(u) = L(v)$; $S(u) = C$

Set $L(v)$ and $L(w)$ to the corresponding immediate subcells of C

Make v and w children of u

Else if $p \in S(v)$

Set $L(w)$ to the immediate subcell of $S(v)$ containing p

Insert w as a child of v

$v = w$

While v is not the root node

$v = parent(v)$

Return v

Figure 4.6: Algorithm for the bottom-up construction of the DIAT tree.

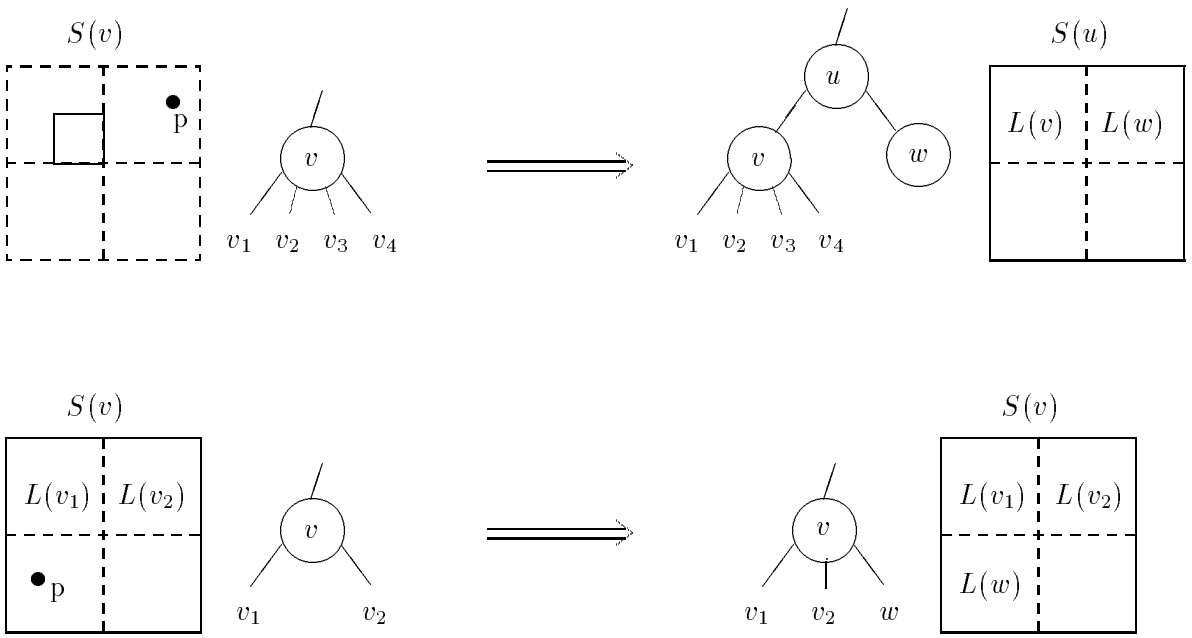

Figure 4.7: Illustration of the two cases of bottom-up DIAT tree construction. The figures on the left shows the DIAT trees and the domains before inserting p and the resulting trees are on the right.

Given n points and a root cell containing them, the points can be sorted in $O(n \log n)$ time in a space-filling curve order. The DIAT tree is then incrementally constructed using this sorted list of points starting from the single node tree for the first point and the root cell. The pseudo code of the algorithm is given in Figure 4.6. During the insertion process, we keep track of the most recently inserted leaf. Let p be the next point to be inserted. Starting from the most recently inserted leaf, traverse the path from the leaf to the root until we find the first node v such that $p \in L(v)$. Two possibilities arise (See Figure 4.7):

- Case I: If the point is not in $S(v)$ the point is in the region $L(v) - S(v)$, which was empty previously. The smallest cell containing p and $S(v)$ is a subcell of $L(v)$ and has p and $S(v)$ in different immediate subcells. Thus, we create a new node u between v and its parent and insert p as a child of u.

- Case II: Otherwise, v is not a leaf node. Consider the children of v presently. None of them has p in its large cell because 1) The node v is the first node on the path to the root which has p in its large cell and 2) the child of v on the path appears after all other children of v but it should appear before the prospective child of v that has p in its large cell. In other words, the DIAT tree presently does not contain a node that corresponds to the immediate subcell of $S(v)$ that contains p, i.e., this subcell does not contain any of the points previously inserted. Therefore, it is enough to insert p as a child for v corresponding to this subcell.

Once the points are sorted, the rest of the algorithm is identical to a post-order walk on the final DIAT tree with $O(1)$ work per node. The time for a single insertion is not bounded by a constant but all n insertions together require only $O(n)$ time. Combined with the initial sorting of the points, the tree can be constructed in $O(n \log n)$ time.

4.1.3 Construction by Repeated Insertions

An algorithm to insert a point in an n-point DIAT tree in $O(\log n)$ time is presented later. Using this algorithm, a DIAT tree can be constructed by starting from a single point tree and inserting the remaining points one by one in any arbitrary order. This results in a third algorithm that also runs in $O(n \log n)$ time.

4.1.4 Construction of BBST

After constructing the DIAT tree, it remains to build the BBST. Given the DIAT tree, the balanced binary search tree can be constructed in $O(n)$ time. First, the sorted order of points according to the DIAT tree is read in $O(n)$ time using a walk on the tree. Then the BBST construction on this sorted data takes additional $O(n)$ time. Alternatively, the sorted order of points according to the DIAT tree can be directly computed in $O(n \log n)$ time and using this sorted data both the DIAT tree and the BBST can be computed in $O(n)$ time each.

Theorem 3 *A DIAT tree for n points can be constructed in $O(n \log n)$ time.*

4.2 Operations on DIAT Trees

4.2.1 Point and Cell Queries

We consider two types of searches on the DIAT trees: point search and cell search. The result of a point search is the the leaf node corresponding to the point if the

point exists. The point search is not the main goal of DIAT trees. However, for cell searches and update operations, we need to use the point searches. If the cell information is not necessary, there are many orderings of points that can be used to perform the point searches efficiently.

Given a cell C, the cell search problem is to locate the node in the DIAT tree representing C. A node v in the tree is said to represent C if $S(v) \subseteq C \subseteq L(v)$, i.e., the points in cell C are exactly the points in the subtree under v. If C does not contain any points, it is returned that the cell does not exist in the DIAT tree.

Point and cell queries are similar since a pooint can be considered to be a zero length cell. A node v represents a cell C if $S(v) \subseteq C \subseteq L(v)$. The node in the DIAT tree representing the given cell is located using the BBST. To use the BBST, first generate the key for the cell to be searched. Then, traverse the path from the root to the node that is being searched. To decide which child to visit next on the path, the comparison routine described earlier is used. If the cell comes before the small cell of the node, continue the search with the left child. If it comes after the large cell of the node, continue with the right child. Otherwise, check whether the cell is represented by the node. If the cell is not represented by the node it is not represented by any node in the DIAT tree. As the height of a BBST is $O(\log n)$, the time taken for a point or cell query is $O(\log n)$.

4.2.2 Insertions and Deletions

As particles can be treated as cells of zero length, we describe insertions and deletions for cells only. These operations are meaningful only if a cell is inserted as a leaf node or if the cell to be deleted is represented by a leaf node in the DIAT tree. Therefore, a cell can not be deleted unless all its subcells are deleted from the DIAT tree.

4.2.2.1 Cell Insertion

To insert a cell, first check whether it is represented in the DIAT tree. If the cell is not represented it should be inserted as a leaf node. Create the a v for the given cell C and insert v in the BBST and the DIAT tree (i.e., set the BBST and DIAT tree pointers). The algorithm is more formally described in Figure 4.8.

To insert v in the DIAT tree, the parent of v in the DIAT tree should be found. To do this, first find the BBST successor of v. The successor is either the parent of v or the leftmost leaf of a child of the parent of v. In either case, the smallest cell containing $S(v)$ and $S(sucessor(v))$ is represented by the parent w of v. If $S(w)$ is a supercell of $S(v)$ (CASE I), v is a missing child of w because otherwise, the successor of v should be the child u of w where $S(u) \supseteq S(v)$. On the other hand, if $S(w)$ is not a supercell of $S(v)$ (CASE II), v can not be inserted directly. It is necessary to create the sibling of v to accommodate the children of the parent, w. Both v and its sibling are made to be the new children of w in the DIAT tree and are inserted in the BBST using a standard insertion algorithm. The overall algorithm requires at most two insertions and three searches in the BBST, thus taking $O(\log n)$ time.

Algorithm 3 Insert-cell (C)

Search the cell C in BBST.

If the cell is represented by a node in BBST, terminate.

Create a node v for the cell C where $S(v) = C$.

Find the successor of v in the BBST

Calculate the smallest cell D containing the small cells of v and the succes sor.

Search the cell D in BBST and find the node w representing D.

CASE I : $S(w) \supset S(v)$.

Insert v as a child of u

CASE II : Otherwise.

Create e new DIAT tree node w'

Copy children of w to w'

$S(w') = S(w)$; $S(w) = D$

$L(w')$ = subcell of $S(w)$ containing $S(w')$

Insert the node w' and v as children of w

Insert the node w' in BBST

$L(v)$ = subcell of $S(w')$ containing $S(v)$

Insert the node v in the BBST.

Figure 4.8: Algorithm to insert the cell C in the DIAT tree.

4.2.2.2 Cell Deletion

As in insertion, the cell should be deleted from the BBST and the DIAT tree. The first step is to check if C can be deleted (i.e., whether it is a leaf node in the DIAT tree). To delete the cell from BBST, the standard deletion algorithm is used. If the removal of the corresponding DIAT tree node leaves its parent with only one child, the parent is deleted as well. Since each internal node has at least two children, the delete operation can not propagate to higher levels.

4.2.3 Group Queries

In group queries, the query should be performed for each particle. While the running time of performing the query on an individual particle may be large, group queries can be answered more efficiently by intelligently combining the work required in answering queries for the particles. For all the queries we consider in this class, a variation of the algorithm presented in Figure 4.9 is used. The functions *Select, Status* and *Process* used in the algorithm are different for different queries.

A node u can be *resolved* with respect to node v if, either $S(u)$ is in the result of the query for all particles in $S(v)$ or $S(u)$ is out of the result of the query for all particles in $S(v)$. Define the *active set* of a node v to be the set of nodes that can not be resolved with respect to v and have small cells smaller and large cells larger than $S(v)$. The algorithm uses a top-down traversal on the DIAT tree. The active set of the root node contains itself. The active set of a node v is calculated by traversing

Algorithm 4 Group-query (v)

P = active set at v's parent

A = active set at $v = \emptyset$

While $P \neq \emptyset$ do

u = *Select (P)*

$P = P - \{u\}$

decision = *Status (v, u)*

If decision = PROCESS

Process (v, u)

If decision = UNKNOWN

If $S(u)$ is smaller than $S(v)$

$A = A \cup \{u\}$

Else $P = P \cup children(u)$

Figure 4.9: Unified algorithm for the group queries.

portions of the subtrees rooted at the nodes in the active set of its parent. When considering the status of a node u with respect to the node v, the function *Status(v,u)* returns one of the following three values:

PROCESS	–	If $S(u)$ is in the result of the query for all particles in $S(v)$
DISCARD	–	If $S(u)$ is not in the result of the query for all particles in $S(v)$
UNKNOWN	–	If neither of the above is true

If the result is either PROCESS or DISCARD, the children of u are not explored. Otherwise, u cannot be resolved with respect to v. If $S(u)$ is smaller than the size of $S(v)$, then u is added to the set of active nodes at v for later consideration by v's children. If $S(u)$ is larger, then u's children are considered with respect to v. It follows that the active set of a leaf node is empty, as the length of its small cell is zero. Therefore, entire subtrees rooted under the active set of the parent of a leaf node are explored and the operation is completed for the particle inhabiting that leaf node.

The order in which nodes are considered is important for deriving optimal algorithms for some operations. The function *Select* is used to accommodate this.

4.2.3.1 Spherical Region Queries

For each particle, the spherical region query finds all particles that lie within a distance of $r > 0$ from it.

A cell D may contain particles in the spherical region corresponding to some particle in cell C only if the smallest distance between D and C is less than r. If the largest distance between D and C is less than r, then all particles in D are in the spherical region of every particle in C. Thus, the function *Status(v,u)* is defined as follows: If the largest distance between $S(u)$ and $S(v)$ is less than r, return PROCESS. If the smallest distance between $S(u)$ and $S(v)$ is greater than r, return DISCARD. Otherwise, return UNKNOWN. Processing u means including all the particles in u in the query result for each particle in v. For this query, we do not need any special selection.

4.2.3.2 Nearest Neighbors

The invariant we would like to maintain for nearest neighbor queries is that one of the nodes in the active set of a node v contains the nearest neighbor of each particle in v. Initially, the root node has itself in its active set so, the invariant is satisfied. The algorithm keeps track of the minimum of largest distances between $S(v)$ and $S(w)$ for all w in P. If the smallest distance between $S(v)$ and $S(u)$ is greater than the minimum, node u can not contribute to nearest neighbors with respect to $S(v)$ and can be discarded safely without affecting the invariant. The function *Status(v,u)* returns DISCARD in this case and UNKNOWN otherwise. The option PROCESS is never used. To be able to prove the running time of $O(n \log n)$, the nodes in P should be considered in the order of the sizes of their small cells, from the largest to the smallest. The function *Select* picks a node in P having the largest small cell,

breaking ties arbitrarily.

Computing nearest neighbors is a well-studied problem [23, 25, 112]. We can relate the computation performed by our algorithm to Vaidya's algorithm [112], even though the algorithms appear to be very different on the surface. Though Vaidya does not consider DIAT trees, the computations performed by his algorithm can be related to traversal on DIAT trees and a proof of the running time of our algorithm can be established by a correspondence with Vaidya's algorithm. Vaidya proves that for each internal node, the size of the active set is constant and the set P never grows to more than a constant size during the execution. Thus, the selection of the largest cell can be performed in constant time for internal nodes and no special handling is necessary when a node is inserted in the set P. On the other hand, the size of the set P can grow as large as $\Theta(n)$ for the leaf nodes. By using a heap for the small cells of the nodes in P, both the function Select and insertions to P can be accomplished in $O(\log n)$ time. Vaidya proves that any node w can be considered by only a constant number of nodes [112]. While there are $O(n)$ nodes in a DIAT tree, the total number of insertions and activations of the function Select is bounded by $O(n)$. As a result, the total running time for nearest neighbor queries is $O(n \log n)$.

4.2.3.3 k-Nearest Neighbors

For computing the k-nearest neighbors of each particle, some modifications to the nearest neighbors algorithm are necessary. For each node w in P, the algorithm keeps track of the largest distance between $S(v)$ and $S(w)$. Let d_k be the k^{th} smallest

of these distances. If the number of nodes in P is less than k, then d_k is set to ∞. The function *Status(v,u)* returns DISCARD if the smallest distance between $S(v)$ and $S(u)$ is greater than d_k. The function *Select* still picks the largest cell in P.

The running time of the algorithm is $O(kn \log n)$. To prove this bound, we consider a slight modification to the DIAT tree. Assume the internal nodes that have less than k points are deleted and each leaf node is made a child of the closest ancestor that has more than k children. In this modified tree there are $O(n/k)$ internal nodes and for each internal node, the size of its active set is still constant. However, an internal node in the modified tree can be considered by $O(k)$ nodes during k-nearest neighbor queries. Assume node v considers node w. If the value of *Status(v,w)* is UNKNOWN, all the children of w should be inserted in the set P of v. Since there are at most $O(k)$ children of w, considering w may result in $O(k)$ insertions. Thus each internal node contributes to $O(k^2)$ insertions and there are a total of $O(kn)$ insertions in P. A heap can be maintained at each leaf node to perform the insertions and selections in $O(\log n)$ time. So, the total running time is $O(kn \log n)$.

Chapter 5

PARALLEL TREE ACCUMULATIONS

In this chapter, the focus is on two types of accumulations on trees of bounded degree, upward accumulations and downward accumulations. The upward accumulation problem is to aggregate the data at all the descendants of each node in the tree. Sequentially, this operation can be done using an upward pass from leaves to the root. Hence, the name upward accumulation. For leaves, the accumulation is trivial. For an internal node, the aggregated data in the children of the node are combined with the data at the node using an accumulation function. In the downward accumulation problem, for each node in the tree, the data in the ancestors are accumulated. Analogously, this operation can be done by traversing the tree downwards from the root to the leaves. For each node, the aggregated data is copied from the parent node and combined with the data in the node using an accumulation function. Both

accumulations take optimal $O(n)$ sequential time, where n is the number of nodes in the tree.

Tree accumulations are part of a variety of applications ranging from graph problems to scientific computing. They have been used for determining minimum covering and maximal independent sets, drawing and labeling trees and evaluating expression trees [31, 41, 46, 65, 66, 91]. Broadcast and gather operations on trees and calculation of the depth of the nodes and sizes of the subtrees can be accomplished using appropriate accumulation functions. To parallelize applications that use accumulations, the accumulation procedures should be parallelized as well. For example, we employ upward and downward accumulations in the parallel Fast Multipole Method used to compute pairwise interactions between multiple data sources in subsequent chapters.

5.1 Tree Accumularions on PRAM Model

A related problem is that of parallel tree contraction. Tree contraction is to repeatedly reduce a tree using independent node removals until a tree consisting of only the root node is obtained. There are two basic methods of parallel tree contraction, introduced by Miller and Reif [65] and Kosaraju and Delcher [58]. Both these methods are optimal $O(\log n)$ time algorithms using $O(n/\log n)$ EREW PRAM processors. In this section, we briefly describe these algorithms and how to use them to calculate tree accumulation [3, 39, 43, 58]. Both algorithms consist of two phases, *contraction* and *expansion*. Gibbons presents a CREW PRAM algorithm for downward accumulations

that runs in $O(\log h)$ time where h is the height of the tree [42].

The first algorithm is based on the tree contraction algorithm by Kosaraju and Delcher [58]. We focus on upward accumulation and explain the differences between upward and downward accumulations later. Consider a complete binary tree T. Define a primitive operation, called *contract*, as follows: The contract operation can be applied to a leaf node v and it involves removing v and its parent from T by connecting v's sibling directly to its grandparent. During the removal process, some information about the sibling is stored in v, to be used in the expansion process. The data in v, its parent and its sibling are combined and stored in the sibling. Assume the leaf nodes of T are numbered from left to right, excluding the leftmost and rightmost leaves. The contract operation is applied concurrently, first to odd numbered leaves that are left children of their parents, and then to odd numbered leaves that are right children of their parents. This guarantees that the parents of the leaves contracted are not adjacent and almost half of the nodes in the tree are removed. Thus, after $O(\log n)$ iterations, all the nodes are removed except the leftmost and rightmost leaves. After processing these two leaves, the accumulation at the root node is obtained. To calculate accumulations at other nodes, an expansion phase is performed. The nodes are expanded in the reverse order of contraction. Before a leaf node is expanded, its sibling has its correct accumulation. The leaf node and its parent are inserted into the tree and the information stored in the leaf node and the accumulation in the sibling are used to compute the final values of these nodes. For downward accumulation, the contract operation is changed; instead of removing a

leaf and its parent, the leaf and its sibling are removed. The accumulation during the contraction phase and the information stored in the leaf nodes are changed as well.

The second algorithm is by Miller and Reif [65]. Once again, we describe upward accumulation only and downward accumulation is treated similarly. Consider a tree with constant degree. Let *rake* be the operation of removing all the leaves from the tree. During the rake operation, data in the leaf nodes are aggregated with the data in their parents. Note that successive applications of the rake operation may not result in an $O(\log n)$ iteration algorithm. To rectify this problem, the *compress* operation is used. Let a *chain* be a maximal sequence of nodes $v_1, v_2, \ldots, v_l$ such that v_{i+1} is the only child of v_i $(1 \leq i < l)$. The maximality of the sequence implies that v_1 has a sibling and v_l is either a leaf or has at least two children. In each iteration, the compress operation removes every other node in such chains by using pointer jumping, i.e., by adjusting the pointer structure for the remaining nodes on a chain to point to their respective grandparents. Thus, the length of the chains is halved after each iteration. During pointer jumping, the data in its parent is aggregated in each node and some information about the parent is stored in the stack of the node. At each iteration in the contraction phase, a rake operation on the leaf nodes in the residual tree and a pointer jumping on all the chains are performed. It is proved in [65] that after $O(\log n)$ iterations, all the chains are removed and the final value for the root node is calculated. Expansion phase is used to calculate upward accumulations for all nodes. During the expansion phase, the information stored in the stacks is retrieved in the reverse order and the accumulations in the parent nodes are restored to the

final value. For downward accumulation, the general structure of the computation is the same as for upward accumulation but the accumulation function during the contraction phase and the information stored in the stacks are different. Note that the maximum size of the stack for a node can be as large as $O(\log h)$ where h is the height of the tree. In [39], Gazit et. al. propose some techniques to reduce the memory requirement for each node to a constant.

Consider a straightforward implementation of these two algorithms on distributed memory computers with p processors. There are two phases and each phase has $O(\log n)$ iterations for both upward and downward accumulations. In each iteration, $O(\frac{n}{p})$ random data accesses and $O(\frac{n}{p})$ calculations are required.

5.2 Distributed Memory Tree Accumulations

The PRAM model ignores the problems associated with data decomposition and the cost of communication, and hence does not capture the essential characteristics of parallel computers. Consider a straightforward implementation of these two algorithms on distributed memory computers with p processors. There are two phases and each phase has $O(\log n)$ iterations for both upward and downward accumulations. In each iteration, $O(\frac{n}{p})$ random data accesses and $O(\frac{n}{p})$ calculations are required.

Tree contractions on distributed memory computers are studied by many researchers [13, 40, 50, 93]. They proposed algorithms for different architectures such as hypercubes and meshes as an extension of one of the two techniques on the PRAM

model. These algorithms use communication techniques that are particular to their target architecture.

The parallel algorithms for upward and downward accumulations presented in this section are both rigorously analyzable and practical, taking into detailed account the distributed nature of memory in parallel computers. They are presented in an architecture-independent fashion, using only well-understood and basic communication operations such as parallel prefix and all-to-all communication. We ensure that a node is deleted only after its upward accumulation is known, yet our algorithm requires only $O(\log n)$ iterations like the two EREW PRAM algorithms. In fact, the number of iterations depends on the shape of the tree and can be anywhere between 1 and $\log n$ for our algorithm, while other algorithms require $\Theta(\log n)$ iterations for any tree. There are several advantages to this approach: The expansion phase for upward accumulation is eliminated. During the contraction phase itself, all the correct accumulations are calculated. As for downward accumulation, our algorithm is also a two-phase algorithm, requiring a contraction followed by an expansion. In the contraction phase, we only explore the structure of the tree and accumulations are calculated in the expansion phase. Only one contraction phase is sufficient for multiple downward accumulations on a tree either for different data or for different accumulation functions, which is not the case for previous algorithms. This reduces the number of random data accesses. While random memory accesses are fundamental to PRAM algorithms, they are very expensive in distributed memory computers.

The following two advantages make our algorithm superior for distributed memory

	n-processor EREW PRAM	p-processor distributed memory parallel computer
Parallel Prefix	$O(\log n)$	$O(\frac{n}{p} + (\tau + \mu)\log p)$
Random data access	$O(1)$	$O(\tau p + \mu\frac{n}{p})$

Table 5.1: Running time of parallel prefix and random data accesses on the EREW PRAM model and distributed memory computers.

computers: Firstly, we use the parallel prefix operation for chains instead of many pointer jumpings which either result in two random data accesses or in larger message sizes if the two are combined into one. We feel that parallel prefix operations are avoided in [65] because they are more expensive than random data access on the PRAM model. On a parallel computer, an EREW PRAM random memory access can be simulated using a transportation primitive, where each processor sends and receives $O(\frac{n}{p})$ sized data. In Table 1, we compare the running times of parallel prefix and random data access (transportation primitive) for the distributed memory and EREW PRAM models. Apparently, random data access is cheaper than parallel prefix for the EREW PRAM model. On the contrary, for distributed memory computers, random data access is considerably more expensive than parallel prefix. Thus, it is beneficial to use parallel prefix instead of random data access. Secondly, our algorithm calculates upward accumulations using only a contraction phase. This also significantly reduces the number of times the expensive transportation primitive is invoked. An important characteristic of our algorithm for downward accumulation

is that we only gather information about the structure of the tree during the contraction phase. If both upward and downward accumulations are necessary for an application, our algorithm requires only one contraction and one expansion. Similarly, for repeated downward accumulation calculations on a tree only one contraction phase followed by expansion phases are sufficient. The previously known algorithms require separate contractions for each downward accumulation, even when the same accumulation function is applied to different data. Although our algorithm is more efficient for distributed memory computers, an implementation of our algorithm on EREW PRAM will be suboptimal - taking $O(\log^2 n)$ time on $O(\frac{n}{\log n})$ processors.

We consider only trees having constant degree and accumulation functions that are associative. We assume that the nodes of the tree are stored in an array T according to its postorder traversal and the array is distributed among processors equally. Each element of the array contains a node, the index in T of its parent and the indices in T of its children.

The following operations can be used to create the input we assume, if it is not readily available: Let T' be the directed graph obtained by replacing each edge in the tree with two directed edges of opposite direction. An *Euler tour* of the tree is a path starting from the root node and traversing all the directed edges in T' exactly once and returning to the root. For each different drawing of the tree on a plane, there is a specific Euler tour which traverses the edges around the outside of T' in a counter-clockwise fashion. Computation of the Euler tour is described in [107] and takes constant time on $O(n)$ EREW PRAM processors. The direct translation to

distributed memory computers with p processors requires $O(\frac{n}{p})$ work and a constant number of transportation primitives. The Euler tour can be used to determine the postorder traversal of the tree [55]. Note that, the Euler tour may visit a node several times but it is visited for the first time from its parent and for the last time after visiting all its descendants. So, if we remove all the repetitions except the last occurrence and number the remaining nodes, we get the postorder traversal of the tree. This can be accomplished by using a parallel prefix on the edges of the path after assigning 0 to all the edges pointing downward and 1 to all edges pointing upward. The postorder number of a node is according to the prefix sum of the directed edge from the node to its parent. The methods described in [102, 103] for the list ranking problem can be used for an efficient implementation of such a parallel prefix. Now, we can move all the nodes to proper processors and adjust pointers to the parents. This sequence of operations uses $O(\log n)$ transportation primitives and the computation cost for it is $O\left(\frac{n}{p}\log n\right)$.

5.2.1 Upward Accumulation

Consider the following straightforward algorithm for upward accumulation: Aggregate the data items stored at the leaves in their parent nodes and remove the leaves. Repeat this operation until the upward accumulation at the root node is computed. The runtime of this algorithm is proportional to the height of the tree, provided the leaf nodes can be operated on and removed in parallel. As the worst-case height of an n node

tree can be as large as $\Omega(n)$, such an algorithm may take $\Omega(n)$ iterations.

In a tree where each internal node has at least two children, at least half of the total number of nodes in the tree are leaf nodes. Thus, removing the leaf nodes in such a tree removes at least half the nodes in the tree. If this property can be ensured in every iteration, this leads to an algorithm with $O(\log n)$ iterations. Unfortunately, removal of the leaf nodes may create chains in the tree. We call a chain a *leaf chain* if the last node in the chain is a leaf, and an *internal chain* otherwise. The tree shown to the left in Figure 5.2(B) illustrates an internal chain and a leaf chain. If we can compress the chains before starting the next iteration, this will lead to an algorithm that runs in $O(\log n)$ iterations. We will show that while it is not possible to compress all chains, leaf chains can be compressed and this turns out to be sufficient.

When a node becomes a leaf node in the residual tree, its complete upward accumulation is known. For a leaf chain $v_1, v_2, \ldots, v_l$, nodes other than v_l have a part of their upward accumulation. The total upward accumulation at v_i is obtained by aggregating the stored upward accumulations in $v_i, v_{i+1}, \ldots v_l$. This is the familiar prefix sum problem and a leaf chain can be compressed in parallel using parallel prefix. This strategy does not work for an internal chain because v_i needs the complete upward accumulation at v_l, which is not available for an internal chain.

In our algorithm for computing the upward accumulations, each iteration consists of two steps: In step 1, the upward accumulations at the leaf nodes are added to their parents and the leaf nodes are removed. In step 2, the leaf chains are compressed into a single node. An example of the execution of the algorithm is shown in Figure 5.2.

Algorithm 5 UpwardTreeAccumulation

I. Remove the leaves

For each active leaf node

If the parent of the leaf is local to the processor

Aggregate the data in the leaf to its parent

Mark the leaf deleted

else

Find the processor that has the parent of the leaf

Collect the information to be sent to the processors in messages

Mark the leaf as deleted

Invoke transportation primitive to communicate the messages

Incorporate the received information about the leaves in their parents

II. Identify the leaf chains

For each active node

Retrieve the next active element in T

If the next element has only one child

if the node is a leaf

put a start mark

else

put an end mark

III. Compress the leaf chains and mark the nodes deleted

Perform segmented parallel prefix using start and end marks

Mark the nodes on the leaf chain except the top one as deleted

Figure 5.1: Algorithm for parallel upward accumulations.

Figure 5.2: Execution of the parallel upward accumulation algorithm. The initial tree is depicted in (A). The first two iterations of the algorithm are shown in (B) and (C). The left tree is after step 1 and the right tree is after step 2 of the corresponding iteration.

Figure 5.2(A) illustrates the initial tree and Figures 5.2(B) and 5.2(C) illustrate the first two iterations of the algorithm.

Lemma 6 *Calculation of upward accumulations using the above algorithm requires* $O(\log n)$ *iterations.*

Proof: Consider a modified algorithm in which, at each iteration, not only the leaf chains but all the chains are compressed. Of course, this algorithm may not compute the correct upward accumulations. However, it guarantees that at least one half of the nodes are removed from the tree at each iteration. Thus, the number of iterations is bounded by $O(\log n)$.

To prove that our original algorithm also requires $O(\log n)$ iterations, it is sufficient to show that if a node v is removed as a leaf during the k^{th} iteration by the modified algorithm, it is also removed by the original algorithm during the k^{th} iteration as a leaf. We show this by induction on the iteration number. Obviously, in the first iteration the nodes removed as leaves are the same. Assume that this holds for $k-1$ iterations.

Let a node v be removed as a leaf by the modified algorithm during step 1 of the k^{th} iteration. Then, v has never been on a chain and has at least two descendants $(u_1, ..., u_j,\ j > 1)$ removed as leaves during the $(k-1)^{th}$ iteration. Note that, there could be some chains removed between v and u_i during the $(k-1)$th iteration. Clearly, v could not be removed at a preceding iteration and $u_1, ..., u_j$ are removed as leaves at $(k-1)^{th}$ iteration by the original algorithm. For any i, consider the descendants

of v that are also ancestors of u_i in the initial tree. Each of these is removed by the modified algorithm before k^{th} iteration as part of a chain. Even though none of these nodes is removed by the original algorithm before $(k-1)^{th}$ iteration, they all form a chain and are removed during step 2 of iteration $k-1$. Thus, v is a leaf at the beginning of the k^{th} iteration in the original algorithm and is removed. This completes the proof. ■

Consider the array T storing the nodes of the tree in the postorder traversal order. As the execution of the algorithm progresses, some nodes are marked as deleted and they are never considered in subsequent iterations. Whenever the array is processed, we skip over such elements. The upward accumulations of all the children of a node may not be added to the node in the same iteration. For each node, we keep track of the number of children whose upward accumulations are yet to be added.

Step 1 of the algorithm is executed as follows: Each node that has received upward accumulations of all its children is a leaf node in the residual tree. This allows easy identification of the leaf nodes. The upward accumulation of each leaf node should be incorporated in its parent. As each node keeps track of the location of its parent in array T, the communication from leaves to the processors containing their parents can be effected using one transportation primitive. As the number of children per node is bounded by a constant, the total amount of information received by a processor is at most $O\left(\frac{n}{p}\right)$.

In step 2, each leaf chain should be compressed using a parallel prefix operation.

The number and respective lengths of the leaf chains are dependent upon the tree. The entries in T that are not deleted correspond to the postorder traversal of the residual tree. In this order, a chain of nodes occupies consecutive positions in T (considering only entries that are not marked as deleted). In the notation we used so far, in a chain $v_1, v_2, \ldots, v_l$, the node closest to the root is v_1. In the array, the chain is stored in the reverse order because we use postorder traversal. We use a variant of segmented parallel prefix to collapse the chains together in one operation. Mark the beginning of each leaf chain in T with a *start* marker. Intermediate nodes on any chain are not marked. The final node of any chain is marked with an *end* marker. Nodes that do not fall under any of the previous categories are also marked with an end marker. The identification of different categories is done as follows: If the next entry for a node in the residual tree has only one child, it must be the parent of the node. Consider a node v. If v is a leaf and its next entry is a node with a single child, it gets a start marker. If v is not a leaf and its next entry is a node with a single child, it is not marked. Otherwise, it gets an end marker. A segmented parallel prefix should be performed between matching start and end markers. The array may have many more end markers than start markers. The start marker signifies the beginning of a new segment which terminates at the first end marker encountered.

In each iteration of the algorithm, we perform $O\left(\frac{n}{p}\right)$ work on each processor for the purpose of identifying the leaves, setting up markers etc.. We also use one transportation primitive and one segmented parallel prefix per iteration. Thus, the

overall running time of the algorithm is $O\left(\frac{n}{p}\log n + \tau p \log n + \mu\frac{n}{p}\log n\right)$ for $n > p\log p$.

5.3 Downward Accumulation

Like upward accumulation, a naive algorithm that propagates downward accumulations one level at a time takes as many iterations as the height of the tree. To obtain a better bound on the number of iterations, we reverse the upward accumulation procedure described in Section 4. During the upward accumulation, the leaf chains are collapsed into single nodes. To reverse the procedure, single nodes must be expanded into leaf chains. It is difficult to predict when a node should be expanded and what should be the length of the chain if a node should indeed be expanded. We achieve this by collecting information during the execution of tree contraction which is used in computing upward accumulation so that we can exactly reverse it. The information collected consists of the iteration number at which each node is deleted and the start and end markers of the leaf chains. The downward accumulation algorithm is performed as follows: We start with performing tree contraction to record the necessary information (iteration number, start and end marks) for each node. Note that upward accumulation and the tree contraction algorithm used here are the same except that we do not need to compute upward accumulation itself for the contraction algorithm. So we do not have to perform segmented parallel prefix while compressing leaf chains in step 2. Instead, the chains are just deleted. We compute downward accumulations

while activating the deleted nodes in the reverse order of contraction. We repeat two steps, which are actually the reverse of the contraction phases: Step 1 is the reverse of step 2 in contraction (or upward accumulation), step 2 is the reverse of step 1 in contraction. We keep track of a counter (say i) that starts with the number of iterations in the contraction phase and keep decrementing it after each iteration of step 1 and step 2. The counter i is used to determine which nodes are to be activated in the iteration.

Step 1 of the algorithm activates the leaf chains: Nodes marked deleted in iteration i as part of a leaf chain are activated, and relevant start marks and end marks are set. As mentioned in the upward accumulation section, nodes in the same chain occupy consecutive positions. Segmented parallel prefix is performed to compute the value of downward accumulation in the newly activated chains.

Step 2 activates the leaf nodes: Leaves marked deleted in iteration i are activated. The activated leaves need to be assigned the value of downward accumulation using the value already known in the parent. If the parent is within the same processor as the leaf, the downward accumulation is computed locally. For leaves whose parents are not in the same processor, it has to request and get the values from other processors. This can be achieved using two transportation primitives, one for requests and the other for satisfying the requests. If parents keep track of when the children are marked deleted during contraction, only one transportation primitive is necessary. Step 1 and 2 are repeated until the whole tree is retrieved, and it takes $O(\log n)$ iterations, which is identical to that for contraction of the tree.

Work required for computing downward accumulation is the following: Contraction of the tree needs $O(\log n)$ iterations of $O(n/p)$ work on each processor and one transportation primitive. Expansion takes $O(\log n)$ iterations of $O(n/p)$ work on each processor, one parallel prefix and one transportation primitive. The overall running time of both phases of the algorithm is $O\left(\frac{n}{p}\log n + \tau p \log n + \mu\frac{n}{p}\log n\right)$ for $n > p\log p$.

Algorithm 6 DownwardTreeAccumulation

I. Contract the tree saving necessary information.

Set i to the number of iterations in tree contraction

II. Expand the leaf chains.

For all deleted nodes

Activate the node if it is deleted in iteration i as part of a leaf chain.

Set the start and end marks to nodes if necessary.

Perform segmented parallel prefix on the chain (between end and start marks).

III. Expand the leaf nodes.

For all deleted nodes

If the node is deleted in iteration i as a leaf

Activate the node.

If the parent is local to the processor

Process the node locally using the data from the parent.

else

Put the requests to the parents in a list.

Perform two transportation primitives to request and get the information about the parents.

Parents of the leaves send necessary information to the leaves using transportation primitive.

Process the received information to calculate the values for the leaves.

IV. Decrement i and repeat step II and III until whole tree is retrieved.

Figure 5.3: Algorithm for calculation of downward accumulations.

Chapter 6

FAST MULTIPOLE METHOD ALGORITHM USING DIAT TREES

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 a collection 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. 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.

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. 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.

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. Some of the techniques include transforming the problem to a position-velocity phase space [67, 68], imposing a grid on the system of particles and computing cell-cell interactions [49]. Such techniques either fail to model the system accurately or degrade to $O(N^2)$ complexity for non-uniform distributions of the particles.

Recently, a new class of particle simulation methods have emerged to solve the N-body problem. These approaches are based on the following observation: 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. Instead of computing individual pairwise interactions, the interactions between clusters of particles are approximated provided that the clusters are far enough compared to the size of the clusters. To carry out this procedure requires imposing a hierarchical clustering scheme on the particles and approximating the force calculations by computing cluster-to-cluster interactions whenever possible. These methods are usually referred to as *hierarchical methods* or *tree methods*. A single iteration using these methods consists of

1. Computing a hierarchical clustering scheme.

2. Computing the approximate force on each particle using the clustering scheme.

Such a method was first proposed by Appel [10], whose scheme allows for clusters with arbitrary shapes. The most significant breakthrough in this regard is Greengard's fast multipole method [45], which uses a hyperoctree data structure for clustering the particles. Though widely accepted to be an $O(n)$ algorithm, it is shown that the algorithm is dependent upon the distribution of the particles [5]. Furthermore, the complexity of navigating the hyperoctree for finding the required clusters is ignored in Greengard's algorithm.

Callahan and Kosaraju [23] recently proposed a new data structure and an algorithm for the N-body problem which has a running time of $O(N \log N)$, independent of the distribution. Although their data structure is also based on hierarchical subdivision of space occupied by the particles, the subdivision is performed along the largest coordinate which results in non-cubical subspaces. This data structure is used to find well-separated pairs of nodes. They proved that it is always possible to find $O(N)$ well-separated pairs in $O(N \log N)$ time. The N-body problem is then solved by calculating mutual potential fields for each pair and summing them up to find the overall field on a particle. The main drawback of this algorithm is that both building the data structure and forming the well-separated pairs are very complicated procedures and require extra space. Further, the method is different from the presently used methods, so the available codes should be rewritten from scratch to use this algorithm. However, the data structure that we propose employs the same subdivision strategy used in Barnes-Hut and Greengard's algorithms. This makes the procedure for building the data structure easy and permits us to use the same codes for N-body simulations with minor changes. The regularity in the shapes and locations of the cubical subspaces used by our algorithm makes it easy to perform error calculations.

In their paper, Callahan et. al. observe:

> "However, the algorithm in Greengard makes assumptions about the geometric distribution of charges, which might be justifiable on pragmatic

> grounds, but are not normally made when treating problems in computational geometry. Also, when the desired output precision is much less than the input precision, the justification becomes weaker. Without these assumptions, we cannot even see an obvious variant of the Fast Multipole Method that run in $O(n \text{ polylog}(n))$ steps."

In this chapter, we answer this in the affirmative by presenting an optimal $O(n)$ algorithm for the N-body problem using a data structure and framework very similar to the Greengard's algorithm. Though our run-time is the same as Callahan et. al.'s algorithm and both are optimal, there are some important differences: Almost all the N-body algorithms used by practitioners involve hyperoctrees. Therefore, it is easier to switch to DIAT trees as both types of trees contain same type of "cells". The regularity in the shapes and locations of the cells makes it easy to perform error calculations. We also present an algorithm to incrementally update the DIAT tree from iteration to iteration. This avoids rebuilding the tree from scratch.

6.1 Solving N-body Problem Using DIAT Trees

The starting point for our algorithm is Greengard's fast multipole method, which uses hyperoctrees. Though our algorithm is a natural variant of the Greengard's algorithm, there are very important subtle differences and its proof of optimality is far from obvious. We also prove the optimality of our algorithm under a relaxed set of assumptions that make it applicable not only to Greengard's algorithm, but other

types of algorithms as well. For instance, it follows from our proofs that Appel's original N-body algorithm [10], when run on a DIAT tree data structure is also an $O(n)$ algorithm.

Greengard's fast multipole algorithm [44] is a two-pass procedure on the quadtree. 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 expansion at a node v describes the potential field due to particles in v. At the leaf nodes, the multipole expansions can be 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 v describes the potential field due to all particles in the nodes well-separated from v. The multipole expansions are used to calculate the local expansions. However, the multipole expansion of a node only converges for the region outside the smallest spherical region (the ball) containing all the points in the node, so not all multipole expansions can be used in the calculation of a local expansion. Moreover, the distance from the center of the ball is related to the error in approximation. Two cells C_1 and C_2 with equal lengths are regarded as *well-separated* if they are not neighbors. Two nodes v_1 and v_2 are said to be *well-separated* if the cells represented by the nodes are well-separated. If v_1 and v_2 are well-separated, the multipole expansion of v_1 can be used in calculation of the local

expansion of v_2. If two nodes (two cells) are not well-separated then they are called *ill-separated*. The set of nodes which are ill-separated from v is called the *ill-separated set* of v. The local expansion of an internal node v is inherited from its parent after shifting to the center of the cell represented by v. This expansion includes the fields due to all particles that are in the the nodes well-separated from the parent. The nodes which are children of a node in the ill-separated set of the parent and well-separated from v constitute the *interaction set* of v. To obtain the complete local expansion, the multipole expansions of the nodes in the interaction set should be shifted and added to parent's local expansion. Note that the size of the cells represented by the node v and the nodes in the interaction or ill-separated sets are all equal. 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 [44].

Consider a run of the Greengard's algorithm on the quadtree having 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 and they 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 $v_1, v_2, ..., v_{k-1}$ can be avoided by using the DIAT tree.

On the other hand, the local expansion of a node is not directly calculated from the local expansion of its parent. The nodes in the interaction set contribute to the local expansion also. While calculating the local expansions of the nodes $v_1, v_2, ..., v_k$ in a path, their interaction sets are utilized. In the DIAT tree, the nodes $v_1, v_2, ..., v_{k-1}$ are not available, so their interaction sets should be accounted for in the calculation at the node v_k. The nodes in the different interaction sets represent cells of different sizes. Further, the nodes in the interaction sets may not be available in the DIAT tree. So, the algorithm for computing local expansions should have the capability to deal with such complications.

Consider two cells C and D, not necessarily of the same length. Define a predicate $well\text{-}separated(C, D)$ to be true if D's multipole expansion converges at any point in C, and false otherwise. We use $ip(C)$ to denote the immediate supercell of a cell C. The set of cells that are not well-separated from the cell C is called *ill-separated set of* C and is defined by $IS^{=}(C) = \{D \mid length(C) = length(D), \neg well\text{-}separated(C, D)\}$. Similarly, the set of well-separated cells with which the interaction is to be calculated is called *the interaction set of* C and is defined by $I^{=}(C) = \{D \mid length(C) = length(D), well\text{-}separated(C, D), \neg well\text{-}separated(ip(C), ip(D))\}$. The superscript "=" is used to indicate that cells of same length are being considered.

Now, consider two nodes v_1 and v_2 in the DIAT tree. We say that v_1 and v_2 are *well-separated* if and only if $S(v_1)$ and $S(v_2)$ are well-separated. Otherwise, we say that v_1 and v_2 are *ill-separated*. For a node v in the DIAT tree, define the

minimal ill-separated set IS(v) as the set of all nodes ill-separated from v and having the small cell smaller and large cell larger than $S(v)$. More precisely, $IS(v) = \{u \mid ill\text{-}separated(v,u), length(S(u)) \leq length(S(v)) \leq length(L(u))\}$. The *interaction set I(v)* of a node v in the DIAT tree is defined as the set of nodes ill-separated from the parent of v but well-separated from v. More precisely, $I(v) = \{u \mid well\text{-}separated(v,u),\ ill\text{-}separated(v, parent(u))\}$[1]. So, for all $u \in I(v)$, there exists a node $w \in IS(parent(v))$ such that u is a descendant of w.

We devise an $O(n)$ N-body algorithm for any predicate *well-separated* that satisfies the following conditions:

C1. The relation *well-separated* is symmetric for equal length cells, i.e., $length(C) = length(D) \Rightarrow well\text{-}separated(C,D) = well\text{-}separated(D,C)$.

C2. For any cell C, $|IS^{=}(C)|$ is bounded by a constant.

C3. If two cells C and D are not well-separated, any two cells C' and D' such that $C \subseteq C'$ and $D \subseteq D'$ are not well-separated as well.

These three conditions are respected by the various well-separatedness criteria used in N-body algorithms and in particular, the Greengard's algorithm. Well-separatedness criteria is used to decide if the approximate interaction on one cluster due to another should be directly computed. In N-body methods, this decision is solely based on the geometry of the clusters and their relative distance and is oblivious to the number of particles or their distribution within the clusters. Given two cells C and D of the

[1]We use $parent(v)$ to denote the parent of the node v.

same length, it follows that if D can be approximated with respect to C, then C can be approximated with respect to D as well, as stipulated by Condition C1. As for Condition C2, note that if the well-separatedness criteria depends only upon the geometry and relative position of the cells, then the size of the ill-separated sets of cells of the same length should be the same. To compute the interactions on a particular cell, all the cells that are in the ill-separated set of its parent cell must be considered. It follows that, if $|IS^{=}(C)|$ is not a constant for cells of a particular length, we can construct an input that requires processing the ill-separated sets of $O(n)$ such cells, making an $O(n)$ algorithm impossible. Condition C3 merely states that two cells C' and D' are not well-separated unless every subcell of C' is well-separated from every subcell of D'.

Our DIAT tree algorithm is as follows: For each node v in the DIAT tree, we wish to compute the multipole expansion $\phi(v)$ and the local expansion $\psi(v)$. Both $\phi(v)$ and $\psi(v)$ are with respect to the cell $S(v)$. The multipole expansions can be computed by a simple bottom-up traversal in $O(n)$ time. At a node v, $\phi(v)$ is computed by shifting the multipole expansions of the children of v to the center of the cell $S(v)$ and adding them.

Our algorithm to compute the local expansions is given in Figure 6.1. The computations are done using a top-down traversal of the tree. To compute $\psi(v)$, we have to consider the set of nodes that are ill-separated from its parent, i.e., $IS(parent(v))$. We recursively decompose nodes in $IS(parent(v))$ until each node is either 1) well-separated from v or 2) ill-separated from v and the length of the small cell of the

---

Algorithm 7 Compute-Local-Exp (v)

I. Find the ill-separated set $IS(v)$ and interaction set $I(v)$ for v

$E(v) = IS(parent(v))$, $I(v) = \emptyset$

While $E(v) \neq \emptyset$ do

Pick some $u \in E(v)$

$E(v) = E(v) - \{u\}$

If $well\text{-}separated(v, u)$

$I(v) = I(v) \cup \{u\}$

Else if $S(u)$ is smaller than $S(v)$

$IS(v) = IS(v) \cup \{u\}$

Else

$E(v) = E(v) \cup children(u)$

II. Calculate the local expansion at v

Assign shifted $\psi(parent(v))$ to $\psi(v)$

For each node $u \in I(v)$

Add shifted $\phi(u)$ to $\psi(v)$

III. Calculate the local expansions at the children of v with recursive calls

For each child u of v

Compute-Local-Exp (u)

Figure 6.1: Algorithm for calculating local expansions of all nodes in the tree rooted at v.

node is smaller than the large cell of v.

To analyze the running time of this algorithm, we need to prove some properties on interaction sets and minimal ill-separated sets.

Lemma 7 *For any cell C, the size of the interaction set is bounded by a constant i.e., $|I^{=}(C)| = O(1)$.*

Proof: Consider the ill-separated set of the immediate supercell of C, $IS^{=}(ip(C))$. By Condition C2, $|IS^{=}(ip(C))| = O(1)$. Consider any $D \in I^{=}(C)$. It is an immediate subcell of a cell in $IS^{=}(ip(C))$. The number of immediate subcells of a cell is no greater than 2^d. Thus, $|I^{=}(C)| \leq 2^d \times |IS^{=}(ip(C))| = O(1)$. ■

Lemma 8 *For any node v in the DIAT tree, the size of the minimal ill-separated set of v is bounded by a constant i.e., $|IS(v)| = O(1)$.*

Proof: Consider any node v and the interaction set $IS^{=}(S(v))$. Each $u \in IS(v)$ can be associated with a cell $C \in IS^{=}(S(v))$ such that $S(u) \subseteq C$. The cell C cannot be associated with any other node in $IS(v)$ because 1) $length(L(u)) \leq length(S(v)) = length(C)$. Thus any subcell of C which is not a subcells of $S(u)$ is not represented in the DIAT tree. 2) For any two nodes $u_1, u_2 \in IS(v)$, $S(u_1)$ and $S(u_2)$ are disjoint. It follows that $|IS(v)| \leq |IS^{=}(S(v))| = O(1)$ (by Lemma 7). ■

Lemma 9 *The sum of interaction set sizes over all nodes in the DIAT tree is linear in the number of nodes in the DIAT tree i.e., $\sum_v |I(v)| = O(n)$.*

Proof: Let v be a node in the DIAT tree. Consider any $w \in I(v)$. Either $w \in IS(parent(v))$ or w is in the subtree rooted by a node $u \in IS(parent(v))$. Thus,

$$\begin{aligned}\sum_v |I(v)| &= \sum_v |\{w \mid w \in I(v)\}| \\ &= \sum_v |\{w \mid w \in I(v), w \in IS(parent(v))\}| \\ &\quad + \sum_v |\{w \mid w \in I(v), w \notin IS(parent(v))\}|.\end{aligned}$$

Consider these two summations separately. The bound for the first summation is easy; From Lemma 8, $|IS(parent(v))| = O(1)$. So,

$$\sum_v |\{w \mid w \in I(v), w \in IS(parent(v))\}| = \sum_v O(1) = O(n).$$

The second summation should be explored more carefully.

$$\sum_v |\{w \mid w \in I(v), w \notin IS(parent(v))\}| = \sum_w |\{v \mid w \in I(v), w \notin IS(parent(v))\}|.$$

In what follows, we bound the size of the set $M(w) = \{v \mid w \in I(v),\ w \notin IS(parent(v))\}$ for any node.

Since $w \notin IS(parent(v))$, there exists a node $u \in IS(parent(v))$ such that w is in the subtree rooted by u. Consider $parent(w)$: The node $parent(w)$ is either u or a node in the subtree rooted at u. In either case, $length(S(parent(w))) \leq length(S(parent(v)))$. Thus, for each $v \in M(w)$, there exists a cell C such that $S(v) \subseteq C \subseteq S(parent(v))$ and $length(S(parent(w))) = length(C)$. Further, since v

and w are not well-separated, C and $S(parent(w))$ are not well-separated as well by Condition C3. That is to say $S(parent(w)) \in IS^{=}(C)$ and $C \in IS^{=}(S(parent(w)))$ by Condition C1. By Condition C2, we know that $|IS^{=}(S(parent(w)))| = O(1)$. Moreover, for each cell $C \in IS^{=}(S(parent(w)))$, there are at most 2^d possible choices of v because $length(C) \leq length(S(parent(v)))$. As a result, $|M(w)| \leq 2^d \times O(1) = O(1)$ for any node w. Thus, $\sum_v |I(v)| = \sum_w |\{v \mid w \in I(v),\ w \notin IS(parent(v))\}| = \sum_w O(1) = O(n)$. ■

Theorem 4 *Given a DIAT tree for n particles, the N-body problem can be solved in $O(n)$ time.*

Proof: The new algorithm still consists of two traversals of the DIAT tree. Computing the p-term multipole expansion at a node takes constant time. Because the number of nodes in the DIAT tree is $O(n)$, total time for bottom-up traversal is $O(n)$. The work involved in the calculation of the local expansion at a node v can be considered as exploring a collection of $|IS(parent(v))|$ rooted subtrees, one subtree for each node in minimal ill-separated set $IS(parent(v))$ of the parent of v. The nodes explored are either in minimal ill-separated set or in interaction set of v. In Lemma 8 we proved that the size of the minimal ill-separated set of a node is constant. Then, the total size of minimal ill-separated sets for all nodes in the DIAT tree is bounded by $O(n)$. In Lemma 9, we show that the total size of interaction sets is also bounded by $O(n)$. Thus, running the multipole algorithm on the DIAT tree takes $O(n)$ time. This is irrespective of the distribution of the particles. ■

Another distinguished benefit of the multipole algorithm designed to run on the DIAT tree is that the precision parameter p can be chosen independent of N. In the 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 quadtree, 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 DIAT 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.

6.1.1 Incremental Updates

As noted before, each iteration of the N-body problem consists of two phases: Constructing the DIAT tree and computing the multipole and local expansions for each node of the tree. The running time for the first phase is $O(n \log n)$ by Theorem 3 and the second phase requires $O(n)$ time by Theorem 4. So, the running time of the overall algorithm is $O(n \log n)$. As the running times of both phases are optimal, this is the optimal solution. Note that considerable portion of this time is spent in creating the data structure.

At each iteration, each particle moves by only a small distance and the probability of moving out of its cell boundary is very small. Otherwise, the simulation is not accurate. Since there is a very limited change in the DIAT tree from one iteration to another, we start with the DIAT tree from the previous iteration and update it instead of rebuilding it from scratch.

The algorithm to incrementally update the DIAT tree is as follows: While updating the positions of the particles, they are checked to find if they move out of their cells. The particles that moved out of their large cells are deleted from the DIAT tree. Deletion is simple as only leaves are deleted. Construct a temporary DIAT tree for these deleted particles using the same root cell. Now, these two DIAT trees can be merged to obtain the required DIAT tree for the next iteration. In case a particle moves out of the root cell, find a cell that contains the new position of the particle and has the root cell as one of its immediate subcells. This cell is made the new root cell. Update the large cell at the root and construct the DIAT tree for the deleted particles using the new root cell.

Suppose there are at most k particles that move out of their boundaries at each iteration. The time required to construct the temporary DIAT tree is $O(k \log k)$ and $O(n)$ time is required for merging. Thus, the running time of incrementally updating the DIAT tree is $O(n + k \log k)$. This result improves the running time of N-body algorithm to $O(n + k \log k)$ with $O(n \log n)$ pre-processing time.

Chapter 7

PARALLEL N-BODY ALGORITHM

Considerable research effort is directed at developing parallel implementations of the FMM. Parallel FMM is of interest because 1) the method is computationally demanding, 2) several applications require simulating a large ensemble of data sources, and 3) a large number of iterations are required for studying the evolution.

Certain fundamental characteristics of the FMM translate to difficult challenges for efficient parallelization. The FMM computation consists of a tree construction phase followed by a force computation phase. The data decomposition required for efficient tree construction may conflict with the data decomposition required for force computation. The distribution of the particles changes from iteration to iteration, which has the effect of changing the workload distribution and requires the ability to dynamically rearrange computations. Due to these difficulties, the algorithms

usually depend upon heuristic methods to perform data decomposition and entertain unstructured and irregular communication. For instance, an algorithm may keep track of the workload distribution during the current iteration and use it as an estimate to effect data decomposition for the next iteration. Such heuristic methods, coupled with the distribution-dependent nature of the octree-based FMM make it impossible to rigorously analyze the run-time of the parallel algorithms. The efficacy of the algorithms is often demonstrated by experimental evaluation alone.

Most of the parallelizations employ the octree-based FMM computation, and thus inherit the distribution-dependent nature of the algorithm [61, 89, 109, 114, 116]. The run-times of these algorithms are either not analyzed or analyzed only for uniform distribution due to aforementioned difficulties and/or distribution-dependences. The only exception is an algorithm developed by Teng [109], in which he defines a communication graph to capture the communication pattern underlying the FMM computation and presents an algorithm to partition graph to achieve balanced load and balanced communication simultaneously. The degree of non-uniformity in the distribution of the particles is parameterized by μ, such that the height of the octree for the particles is $O(\log n + \mu)$. The number of operations executed by the algorithm is a function of μ and hence the algorithm is distribution-dependent.

Callahan and Kosaraju developed a distribution-independent parallel algorithm for the FMM using their sequential framework. The algorithm runs on the CREW PRAM model in $O(\log^2 n)$ time, using $O(n)$ processors [23]. The PRAM model

ignores the problems associated with data decomposition and the cost of communication, and hence does not capture the essential characteristics of parallel computers. A straightforward translation of their algorithm does not result in an efficient implementation on parallel computers.

A variation of the interaction computation is given by the Barnes-Hut algorithm [14], which allows interaction between a particle and a cluster but disallows cluster-cluster interactions. The FMM has been incorporated into the Barnes-Hut method and several parallel algorithms have been developed for this method [63, 95, 104]. These methods are also dependent on the distribution.

Our interest lies in designing a parallel FMM algorithm that is distribution-independent and rigorously analyzable. We present such an algorithm in this paper. Our algorithm is a parallelization of the sequential framework proposed by Aluru et. al. [7] and is developed taking into detailed account the physically distributed nature of memory in parallel computers. It is presented in an architecture-independent fashion, using only well-understood communication operations such as parallel prefix, all-to-all communication and sorting. Our parallel FMM algorithm has the following characteristics:

- The algorithm is distribution-independent.
- The run-time of the algorithm can be rigorously proven on any model of parallel computation.
- The algorithm uses only a static data decomposition.

- The algorithm does not require any dynamic data decomposition or explicit load balancing, either within an iteration or across iterations.
- The algorithm can be efficiently implemented on any model of parallel computation that admits an efficient sorting algorithm.

7.1 Parallel Fast Multipole Method

Our parallel FMM algorithm consists of four steps in each iteration:

1. Constructing the compressed octree for the particles.
2. Calculating multipole expansions.
3. Calculating partial local expansions.
4. Calculating local expansions.

The local expansions at the leaf nodes are used to compute the force acting on each particle. The particle positions are updated and this completes one iteration of the N-body problem.

The data decomposition used by our algorithm is extremely simple: Initially, the n particles are distributed across the processors such that each processor has approximately the same number of particles. Throughout the algorithm, we store the compressed octree data structure in an array according to its postorder traversal. This array is uniformly distributed across the processors. We do not use any load

estimation or load balancing either within a single iteration, or in between multiple iterations. Instead, we prove that each processor does the same amount of work during each step of our algorithm.

7.1.1 Compressed Octree Construction

Our algorithm for constructing compressed octrees is based on Bern et. al.'s algorithm [18] for constructing quadtrees on the PRAM model.

To construct the compressed octree, we first sort the particles in parallel based on their positions and using the space-filling curve ordering. This orders the leaf nodes of the compressed octree. The next step is to construct all the internal nodes in the tree. Every internal node is the least common ancestor of at least one consecutive pair of leaves. More specifically, let u be a node in the compressed octree and let $v_1, v_2, \ldots, v_k$ be its children ($2 \leq k \leq 8$). Consider any two consecutive children v_i and v_{i+1} ($1 \leq i \leq k-1$). Then, u is the least common ancestor of the rightmost leaf in v_i's subtree and the leftmost leaf in v_{i+1}'s subtree. If we generate the least common ancestors of every consecutive pair of leaf nodes, we are guaranteed to generate every internal node in the compressed octree. However, there may be duplicates – each internal node is generated at least once but at most seven times. By generating an internal node, we mean generating the small cell of the internal node.

The least common ancestor for every consecutive pair of leaf nodes is generated in parallel. This results in $O(n)$ internal nodes, with possibility of some duplicates. The internal nodes are then mixed with the leaf nodes and they are sorted together

Algorithm 8 Construct-Compressed-Octree

I. Sort and distribute the particles (leaf cells).

II. Create the internal cells.

IV. Sort the internal cells together with the leaf cells.

V. For each cell, create the virtual parent cell.

VI. Sort the virtual parent cells with the actual parents to match them.

VII. Assign pointers between matching cells.

Figure 7.1: Algorithm for parallel construction of compressed octree.

in parallel, again using the space-filling curve ordering. This results in the postorder traversal of the internal nodes of the tree. Duplicates are brought together due to sorting, and can be easily eliminated. We now have all the nodes of the compressed octree stored in an array according to its postorder traversal. This array is referred to as T, for the remainder of this paper.

It remains to assign pointers to complete the construction of the tree. As the nodes are stored in an array, pointer to a node is the same as the index of the array where the node is stored. For each node, we wish to determine its parent cell, search for and locate it and set up the pointers. The details of the process are as follows: For each node, we can determine its parent cell by finding the smallest cell containing the small cells at the node and the its right neighbor in the postorder array. Such a generated parent cell is called a *virtual parent cell.* We generate the virtual parent

cells of all nodes in parallel. To this, we add the copies of the cells corresponding to all the internal nodes (actual parents), mix them together and sort them in parallel to create an array T'. In T', the virtual parent cells and the corresponding actual parent cells come together. Each virtual parent cell brings with it information on the index of the child node that generated it. Similarly, each actual parent cell contains information on the index of the parent node. The child information from the virtual parent cell should be communicated to the actual parent. The parent information from the actual parent cell should be communicated to the child. As each node has a unique parent, the total length of T' is $O(n)$ and each processor has an $O\left(\frac{n}{p}\right)$ sized subarray of T' as a result of parallel sorting. For the purpose of communication, unit sized information is generated from each entry of T'. Thus, the total amount of outgoing information at each processor is $O\left(\frac{n}{p}\right)$. Each processor receives information of size $O(1)$ for each entry of T it stores. This is because each node has a unique parent and at most 8 children. Thus, the total amount of incoming information at each processor is also $O\left(\frac{n}{p}\right)$. The communication itself is done by invoking the transportation primitive.

The parallel algorithm for compressed octree construction uses 3 parallel sort operations and 1 invocation of the transportation primitive. The running time of the algorithm is $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p \log^2 p\right)\right)$ for $n > p^2 \log p$.

7.1.2 Multipole Expansion Calculation

Sequentially, the multipole expansions at the nodes of the compressed octree are calculated using a bottom-up traversal. The multipole expansions at the leaf nodes are computed directly. For an internal node, its multipole expansion is computed by aggregating the multipole expansions of its children. The problem is akin to summing up all the leaves in the subtree for each node of a tree where the leaves store numbers and is known as upward tree accumulation.

In a parallel algorithm, the multipole expansions can be calculated concurrently for the leaf nodes. The multipole expansions of the internal nodes are calculated using the parallel upward accumulation algorithm described previously. Note that shifting a mulripole exoansiuon from a child to its parent takes constant time. Thus multipole expansion calculation takes $O\left(\frac{n}{p}\log n + \tau p \log n + \mu\frac{n}{p}\log n\right)$ time for $n > p\log p$.

7.1.3 Partial Local Expansion Calculation

The partial local expansion at a node is obtained by adding the multipole expansions of all nodes in its interaction set. Sequentially, the interaction sets are computed in a systematic manner during a top-down traversal. Before the interaction set at a node is computed, the interaction sets of all nodes along the path from the root to the node are computed. To compute interaction sets in parallel, we need a mechanism to independently generate the interaction sets. A further complication arises due to the fact that while the sum of the sizes of the interaction sets of all nodes in the tree is

$O(n)$, we can only place a bound of $O(n)$ on the size of a single interaction set. Thus, computing one interaction set may take time of the same order as computing all the interaction sets. Therefore, simply computing each interaction set independently is not enough.

Instead of computing the interaction set of each node v, we can reverse the perspective and try to compute the set of nodes which have v in their interaction sets, i.e., compute the set $\{w | v \in I(w)\}$. Then we may try to add the multipole expansion of v to all these nodes. This strategy would be beneficial if the size of each such set is a constant. Unfortunately, this cannot be guaranteed and the size of such a set is bounded only by $O(n)$.

$$I(v) = \{w |\ well\text{-}sep(v,w), [w \in P(parent(v)) \vee \{length(S(v)) < length(S(parent(w))),$$
$$\neg well\text{-}sep(v, parent(w)),\ \exists u \in P(parent(v)), w \text{ is a descendant of } u\}]\}$$

We will show that by employing a combination of the two strategies, we can ensure that the set sizes are constant. Recall that the interaction set of a node v is defined as

Each node in $I(v)$ is either in $P(parent(v))$ or a descendant of a node in $P(parent(v))$. We divide the interactions according to these two cases and define two additional sets, the *forward interaction set* $I_f(v)$ and the *backward interaction set* $I_b(v)$ as follows:

$$\begin{aligned} I_f(v) &= \{w \mid w \in I(v),\ w \in P(parent(v))\} \\ &= \{w \mid well\text{-}sep(v,w),\ w \in P(parent(v))\} \end{aligned}$$

$$\begin{aligned} I_b(v) &= \{w \mid v \in I(w),\ v \notin P(parent(w))\} \\ &= \{w \mid well\text{-}sep(w,v),\ \exists u \in P(parent(w)), v \text{ is a descendant of } u, \\ &\qquad \neg well\text{-}sep(w, parent(v)),\ length(S(w)) < length(S(parent(v)))\} \end{aligned}$$

Note that the interaction set for a node v satisfies

$$I(v) = I_f(v) \cup \{w \mid v \in I_b(w)\}.$$

In the following, we seek to establish that $|I_f(v)| = O(1)$ and $|I_b(v)| = O(1)$ for any node v and show how to compute these sets in parallel for all nodes in the tree. For a cell C, the set $P^{=}(C)$ can be formed with only the knowledge of C. This is because the set $P^{=}(C)$ contains cells that are of the same size as C but that are close enough that their multipole expansions do not converge in C. These cells are all contained in a constant number of layers around cell C. Furthermore, the size of the set $P^{=}(C)$ is constant. The following two lemmas assert that for a node v, if we search the cells in $P^{=}(S(parent(v)))$ in the compressed octree we can find the nodes in $I_f(v)$ and if we search the immediate subcells of $P^{=}(S(parent(v)))$ we can find the nodes in $I_b(v)$.

Lemma 10 *Let v and w be two nodes in a compressed octree. Then*

$$\begin{aligned} w \in I_f(v) \quad \textit{iff}\ & well\text{-}sep(v,w),\ \neg well\text{-}sep(parent(v),w), \\ & \exists C\ s.t.\ S(w) \subseteq C \subseteq L(w),\ C \in P^{=}(S(parent(v))). \end{aligned}$$

Proof: Let $w \in I_f(v)$. By definition, $well\text{-}sep(v,w)$ and $w \in P(parent(v))$. By the definition of proximity set, $\neg well\text{-}sep(parent(v),w)$ and $length(S(w)) \leq length(S(parent(v)))$

$\leq length(L(w))$. Thus, $\exists C$ such that $length(S(parent(v))) = length(C)$ and $S(w) \subseteq C \subseteq L(w)$. Since $\neg well\text{-}sep(parent(v), w)$ and $S(w) \subseteq C$, C and $S(parent(v))$ are not well-separated. Then, $C \in P^{=}(S(parent(v)))$.

Now, let $well\text{-}sep(v, w)$, $\neg well\text{-}sep(parent(v), w)$ and C be such that $S(w) \subseteq C \subseteq L(w)$ and $C \in P^{=}(S(parent(v)))$. Then, $length(S(w)) \leq length(S(parent(v))) \leq length(L(w))$ and by definition, $w \in P(parent(v))$. Thus, by the definition of $I_f(v)$, $w \in I_f(v)$. ■

Corollary 5 *The size of the set $I_f(v)$ is bounded by a constant.*

Proof: Follows from the previous lemma and the fact that the size of the proximity set of a cell is bounded by a constant. ■

Lemma 11 *Let v and w be two nodes in a compressed octree. Then,*

$$w \in I_f(v) \quad iff \ well\text{-}sep(v, w),\ \neg well\text{-}sep(parent(v), w),$$
$$\exists C \ s.t.\ S(w) \subseteq C \subseteq L(w),\ C \in P^{=}(S(parent(v))).$$

Proof: Let $w \in I_b(v)$. By definition, $well\text{-}sep(w, v)$ and $\neg well\text{-}sep(w, parent(v))$. $\exists C'$ such that $S(w) \subset C'$ and $length(S(w)) < length(C') = length(S(parent(v)))$, since $length(S(w)) < length(S(parent(v)))$. Furthermore, $\exists u \in P(parent(w))$ such that u is an ancestor of v so, $length(S(parent(v))) \leq length(S(u)) \leq length(S(parent(w)))$. Therefore, an immediate subcell C of C' satisfies $S(w) \subseteq C \subseteq L(w)$. Because $\neg well\text{-}sep(w, parent(v))$, C' and $S(parent(v))$ are not well-separated, $C' \in P^{=}(S(parent(v)))$.

On the other hand, let $well\text{-}sep(w, v)$ and $\neg well\text{-}sep(w, parent(v))$ and let C be such that it is an immediate subcell of a cell $C' \in P^{=}(S(parent(v)))$ and $S(w) \subseteq C \subseteq L(w)$. Then, $length(S(w)) < length(S(parent(v)))$. Since $C \subseteq L(w)$, $C' \subseteq S(parent(w))$ and $\neg well\text{-}sep(parent(w), parent(v))$. Then, there is an ancestor u of v such that $length(S(u)) \leq length(S(parent(w))) \leq length(L(u))$. Note that $\neg well\text{-}sep(parent(w), u)$ because both $\neg well\text{-}sep(parent(w), parent(v))$ and u is either $parent(v)$ or an ancestor of $parent(v)$. So, by definition, $u \in P(parent(w))$. It follows from the definition of $I_b(v)$ that $w \in I_b(v)$. ■

Corollary 6 *The size of the set $I_b(v)$ is bounded by a constant.*

Proof: Follows from the previous lemma and the fact that the number of immediate subcells of a cell and the size of the proximity set of a cell is bounded by a constant. ■

Our algorithm for computing partial local expansions is shown in Figure 7.2. Consider the computation of forward interaction sets and the corresponding accumulation of multipole expansions. We use Lemma 3 to identify the forward interaction sets. For each node v, compute the cells in $P^{=}(S(parent(v)))$. For each cell $C \in P^{=}(S(parent(v)))$, search for the node whose small cell is contained in C and whose large cell is at least as large as C. Retrieve the nodes and their multipole expansions. The multipole expansion of such a retrieved node w is added provided w is well-separated from v but not well-separated from its parent. To see if w is well-separated from v's parent, only the small cell information of v's parent is needed.

Algorithm 9 Partial-Local-Expansion-Calculation

I. Compute all forward interaction sets in parallel.

II. For each node, incorporate the multipole expansions of the cells in its forward interaction set into its partial local expansion.

III. Compute all backward interaction sets in parallel.

IV. For each node, incorporate its multipole expansion into the partial local expansion of each node in its backward interaction set.

Figure 7.2: Algorithm for parallel calculation of partial local expansions.

Note that this can be immediately obtained by finding the smallest cell that contains $S(v)$ and its neighbor's small cell in the array T.

Such a search should be done for all nodes together in parallel. Each processor scans its portion of T and generates the cells in $P^{=}(S(parent(v)))$ for every node v. Since T is a sorted array of cells, cell searches can be done by finding the destination processor for each cell search and forwarding the search to the processor. However, this does not lead to balanced communication as the number of nodes from which a single node receives requests may not be constant and bounded only by $O(n)$. To get around this problem, we perform searches in the following way: For each node v in T, we put $S(v)$ and the cells in $P^{=}(S(parent(v)))$ together into an array T' and sort T' in parallel. Along with each $S(v)$, we also carry its multipole expansion. The size of the array T' is $O(n)$, and when it is generated, each processor will have an $O\left(\frac{n}{p}\right)$ sized

subarray of it. Sorting is done so that each cell having a multipole expansion comes before all the cells searching for it that are grouped together. Now, we can broadcast the multipole expansions using segmented parallel prefix to the cells searching for such information. The cells having the multipole expansions show the separation of the segments. We send the multipole expansion at each search cell to its originating node using the transportation primitive. Each entry of T' generates only constant sized information. Similarly, each processor receives $O(1)$ amount of information for each node in array T. Thus, the total amount of outgoing and incoming communication at any processor is bounded by $O\left(\frac{n}{p}\right)$ and this leads to balanced communication. After the multipole expansions are received, they are incorporated if found appropriate according to Lemma 3.

An almost identical strategy works for the computation of the backward interactions sets. We highlight only the deviations: Use Lemma 5 and for each node v, generate the immediate subcells of $P^{=}(S(parent(v)))$. The multipole expansion at v and $S(v)$ are copied and carried along with each of these cells. After sorting, the array T' contains segments as before. Within each segment, the first entry is a cell coming from T and rest of the entries are identical cells whose multipole expansions should be added to the first entry. This addition is accomplished in parallel for all segments by reduction (reduction is the problem of computing the total sum) using a segmented parallel prefix.

The partial local expansion calculation uses 2 parallel sort, 2 segmented parallel

prefix and 2 transportation primitive operations. Thus, for $n > p^2 \log p$, the partial local expansion calculation requires $O\left(\frac{n \log n}{p} + \tau p + \mu\left(\frac{n}{p} + p \log^2 p\right)\right)$ time.

7.1.4 Local Expansion Calculation

The local expansion for a node in the compressed octree is obtained by adding the partial local expansions along the path from the root to the node. This is same as a downward accumulation on the tree. Thus the local expansions for all nodes can be calcualted using the parallel downward accumulation algorithm described previously. Further, the contraction pahse of the downward accumulation is not necessary since the information gathered during the contraction phase can be collected during the upward acuumulation used for mulytipole expansion calculation. Note that shifting a local exoansiuon from a parent to its child takes constant time. Thus local expansion calculation takes $O\left(\frac{n}{p} \log n + \tau p \log n + \mu \frac{n}{p} \log n\right)$ time for $n > p \log p$.

7.1.5 Parallel Run-time Analysis

For each iteration of FMM consisting of both compressed octree construction and force calculation, $O(\frac{n \log n}{p})$ sequential work is performed by each processor and 5 sort operations, $O(\log n)$ parallel prefixes and $O(\log n)$ transportation primitives are performed collectively. The total running time of the algorithm is $O\left(\frac{n \log n}{p} + \tau p \log n + \mu\left(\frac{n \log n}{p} + p \log^2 p\right)\right)$. We should point out following observations about the running time:

- Although we assume that local expansion and multipole expansion calculations

take constant time, these constants are considerably big. Thus, the time spent for force calculation (i.e., calculation of multipole and local expansions) dominates the total running time in actual simulations. In our algorithms, the force calculation is distributed to the processors as evenly as possible and each processor performs only $O(\frac{n}{p})$ force calculation. The rest of the running time is spent for parallel computation overhead (which is simple array manipulations) and communication.

- Our tree construction algorithm uses logarithm and floor operations. In the analysis of the tree construction algorithm, only, we assume an extended model of computation in which these operations can be performed in constant time. The computational complexities of rest of the algorithms are analyzed for the standard algebraic model of computation.

- From an inspection of the run-times of the primitives described in Section 2, it is evident that the run-time is dominated by the sort operations for n sufficiently larger than p. Thus, run-time of our parallel algorithm is bounded by the efficiency of parallel sorting which is an inevitable factor for any FMM algorithm as sorting can be reduced to compressed octree construction.

Chapter 8

CONCLUSION AND OPEN PROBLEMS

Bibliography

[1] D.J. Abel, J.L. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing, 24* (1983) 1-13.

[2] D.J. Abel, D.M. Mark, A comparative analysis of some two-dimensional orderings; *International Journal of Geographical Information Systems, 4(1)* (1990) 21-31.

[3] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick and T. Przytycka, A simple parallel tree contraction algorithm, *Journal of Algorithms,10* (1989) 287-302.

[4] S. Aluru, G.M. Prabhu and J. Gustafson, Truly distribution-independent hierarchical algorithms for the N-body problem, *Proc. Supercomputing '94,* (1994) 420-428.

[5] S. Aluru, Greengard's N-body algorithm is not order N. *SIAM Journal on Scientific Computing, 17* (1996) 773-776.

[6] S. Aluru, F.E. Sevilgen, Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves, Proc of HiPC '97, to appear.

[7] S. Aluru, G.M. Prabhu, J. Gustafson and F. Sevilgen, Distribution-independent hierarchical algorithm for the N-body problem, *Journal of Supercomputing, 12* (1998) 303-323.

[8] D.P. Anderson, Techniques for reducing pen plotting time, *ACM Transaction on Graphics, 2(3)* (1983) 197-212.

[9] R.J. Anderson, Tree Data Structures for N-Body Simulations, *Proc. Annual Symposium on Foundations of Computer Science*, (1996) 224-233.

[10] A.W. Appel, An efficient program for many-body simulation, *SIAM Journal of Scientific and Statistical Computing, 6* (1985) 85-103.

[11] T. Asano, D. Ranjan and T. Roos, Digital halftoning algorithms based on optimization criteria and their experimental evaluation, *IEICS Transaction on Fundamentals, E79-A(4)* (1996) 524-532.

[12] T. Asano, D. Ranjan, T. Ross, E. Welzl and P. Widmayer, Spacefilling curves and their use in the design of geometric data structures, *Lecture Notes in Computer Science, 911* 36-48.

[13] M. Atallah and S. Hambrusch. Solving tree problems on a mesh-connected processor array. *Information and Control*, 69:168–187, 1986.

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

[15] J. J. Bartholdi and L. K. Platzman, Heuristic based on spacefilling curves for combinatorial problems in Euclidean space, *Management Science, 34(3)* (1988) 291-305.

[16] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl, "Meshless methods: An overview and recent developments, *Computer Methods in Applied Mechanics and Engineering 139*, (1996) 3-47.

[17] J.L. Bentley, Multidimensional binary search trees used for associative search, *Comm. of ACM, 18(9)* (1975) 509-517.

[18] M. Bern, D. Eppstein and S.H. Teng, Parallel Construction of Quadtrees and Quality Triangulations, *Algorithms and Data Structures, WADS '93* (1993) 188-199.

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

[20] T. Bially, Space-filling curves: their generation and their application to bandwidth reduction, *IEEE Transactions on Information Theory, 15* (1969) 658-664.

[21] J.A. Board, W.J. Blamke, D.C. Gray, Z.S. Hukara, W.D. Elliott and J.F. Leathrum, Scalable implementations of multipole-accelerated algorithms for

molecular dynamics, *Proc. Scalable High Performance Computing Conference*, (1994) 87-94.

[22] A. R. Butz, Alternative algorithm for Hilbert's space-filling curve, *IEEE Transactions on Computers, 20* (1971) 424-426.

[23] P.B. Callahan and S.R. Kosaraju, A decomposition of multidimensional point sets with applications to k-nearest neighbors and n-body potential fields, *J. of the ACM, 42(1)* (1995) 67-90.

[24] F. Chen and D. Suter, Using a fast multipole method to accelerate spline evaluations, *Computational Science & Engineering, 5* (1998) 24-30.

[25] K. L. Clarkson, Fast algorithms for the all nearest neighbors problem, *Proc. Foundations of Computer Science* (1983) 226-232.

[26] D. Comer, The ubiquitous B-tree, *ACM Computing Surveys, 11(2)* (1979) 121-137.

[27] S.P. Dandamudi and P.G. Sorenson, Algorithms for BD Trees, *Software—Practice and Experience, 16 (12)* (1986) 1077-1096.

[28] S.P. Dandamudi and P.G. Sorenson, An empirical performance comparison of some variations of the *k-d* Tree and BD Tree, *International Journal of Computer and Information Sciences, 14(3)* (1985) 135-159.

[29] L. Devroye, A note on the average depth of tries, *computing, 28* (1982) 367-371.

[30] M.D. Dikaiakos and J. Stadel, A performance study of cosmological simulations on message-passing and shared-memory multiprocessors, *Proc. International Conference on Supercomputing* (1996) 94-101.

[31] K. Diks and T. Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theoretical Computer Science*, 203:3–29, 1998.

[32] G. Eisenhauer and K. Schwan, Design and analysis of a parallel molecular dynamics application, *Journal of Parallel and Distributed Computing, 35* (1996) 76-90.

[33] W.D. Elliott and J.A. Board, Fast Fourier transform accelerated fast multipole algorithm, *SIAM Journal on Scientific Computing,17(2)*, (1996) 398-415.

[34] C. Faloutsos and S. Roseman, Fractals for secondary key retrieval, *Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (1989) 247-252.

[35] C. Faloutsos, Gray codes for partial match and range queries, *IEEE Transactions on Software Engineering, 14(10)* (1988) 1381-1393.

[36] C. Faloutsos, Multiattribute hashing using Gray codes, *Proc. ACM SIGMOD International Conference on the Management of Data* (1986) 227-238.

[37] R.A. Finkel and J.L. Bentley, Quad trees: A data structure for retrieval on composite keys, *Acta Informatica, 4(1)* (1974)

[38] M. Freeston, The BANG file: a new kind of grid file, *Proc. of the SIGMOD Conference, San Francisco,* (1987) 260-269.

[39] H. Gazit, G.L. Miller and S.H. Teng, Optimal tree contraction in the EREW model, *Concurrent Computations: Algorithms, Architecture, and Technology,* Plenum, New York, (1988) 39-156.

[40] A. Gibbons and Y. Srikant. A class of problems efficiently solvable on mesh-connected computers including dynamic expression evaluation. *Information Processing Letters*, 32:305–311, 1989.

[41] J. Gibbons. *Algebras for Tree Algorithms.* Ph.D. Thesis, Oxford University, 1991.

[42] J. Gibbons. Computing downwards accumulations on trees quickly. Technical report, University Auckland, 1995.

[43] J. Gibbons, W. Cai and D.B. Skillicorn, Efficient parallel algorithms for tree accumulations, *Science of Computer Programming, 23* (1994) 1-18.

[44] L. Greengard, *The rapid evaluation of potential fields in particle systems*, MIT Press, Cambridge, MA, 1988.

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

[46] X. He. Efficient parallel algorithms for solving some tree problems. *Proc. 24th Allerton Conference on Communication, Control and Computing*, pages 777–786, 1986.

[47] L. Hernquist and N. Katz, TREESPH: A unification of SPH with the hierarchical tree method, *The Astrophysical journal. Supplement series, 70* (1989) 419.

[48] D. Hilbert, Uber die stegie Abbildung einer Linie auf Flachenstuck, *Math. Ann., 38* (1891) 459-460.

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

[50] T. Hsu and V. Ramachandran. Efficient massively parallel implementation of some combinatorial algorithms. *Theoretical Computing Science*, 162:297–322, 1996.

[51] A. Hutflesz, S.W. Six, P. Widmayer, Globally order preserving multidimensional linear hashing, *Proc. International Conference on Data Engineering* (1988) 572-579.

[52] A. Hutflesz, P. Widmayer and C. Zimmermann, Global order makes spatial access faster, *International Workshop on Database Management Systems for Geographical Applications*, ESPRIT Basic Research Series Proc., Springer, 161-176.

[53] H.V. Jagadish, Analysis of the Hilbert curve for representing two-dimensional space, *Information Processing Letters, 62* (1997) 17-22.

[54] H.V. Jagadish, Linear clustering of objects with multiple attributes, *Proc. ACM SIGMOD International Conference on the Management of Data* (1990) 332-342.

[55] J. Jájá, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Co., 1992.

[56] I. Kamel and C. Faloutsos, On Packing R-trees, Technical Report No. CS-TR-3009, University of Maryland Department of Computer Science, December 1992.

[57] J. Katzenelson, Computational structure of the N-body problem, *SIAM Journal of Scientific and Statistical Computing, 10* (1989) 787-915.

[58] S.R. Kosaraju and A.L. Delcher, Optimal parallel evaluation of tree-structured computations by raking, *Proc. 3rd Aegean Workshop on Computing* (1988) 101-110.

[59] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing,* The Benjamin/Cummings Publishing Co., 1994.

[60] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Addison-Wesley, Reading, MA, 1981.

[61] J.F. Leathrum, *Parallelization of the fast multipole algorithm: algorithm and architecture design*, Ph.D. thesis, Duke University, 1992.

[62] L.D. Libersky, A.G. Petschek, T.C. Carney, J.R. Hipp and F.A. Allahdadi, High strain Lagrangian hydrodynamics: A three dimensional SPH code for dynamic material response, *Journal of Computational Physics, 109* (1993) 67-75.

[63] P. Liu, The parallel implementation of N-body algorithms, Technical Report No. 94-27, DIMACS Center, Rutgers University, Piscataway, New Jersey.

[64] F. Manola and J.A. Orenstein, Toward a general spatial data model for an object-oriented DBMS, *Proc. International Conference on Very Large Data Bases* (1986) 328-335.

[65] G.L. Miller and J.H. Reif, Parallel tree contraction and its application, *Proc. IEEE Symposium on the Foundation of Computer Science* (1985) 473-489.

[66] G. Miller and J. Reif. Parallel tree contraction part2: Further application. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

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

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

[69] B. Moon, H.V. Jagadish, C. Faloutsos and J.H. Saltz, Analysis of clustering properties of Hilbert space-filling curve, Technical Report No. CS-TR-3590, University of Maryland Department of Computer Science, March 1996.

[70] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM, Ottawa, Canada (1966).

[71] H.J. Neeman, *Autonomous Hierarchical Adaptive Mesh Refinement for Multi-stage Simulations*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, IL, 1996.

[72] Y. Ohsawa and M. Sakauchi, Multidimensional data management structure with efficient dynamic characteristics, *Systems, Computers, Controls, 14(5)* (1983) 77-87.

[73] J.A. Orenstein, A comparison of spatial query processing techniques for native and parameter spaces, *Proc. ACM SIGMOD International Conference on the Management of Data* (1990) 343-352.

[74] J.A. Orenstein, Redundancy in spatial databases, *Proc. ACM SIGMOD International Conference on the Management of Data* (1989), Portland, 294-305.

[75] J.A. Orenstein and T.H. Merrett, A class of data structures for associative searching, *Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (1984) 181-190.

[76] J.A. Orenstein, Multidimensional tries used for associative search, *Information Processing Letters, 14(4)* (1982) 150-157.

[77] C.W. Ou, *Partitioning and Incremental Partitioning of Adaptive Irregular Problems,* Ph.D. Thesis, Syracuse University, 1995.

[78] C.W. Ou and S. Ranka. Parallel Remapping Algorithms for Adaptive Problems, *Proc. Frontiers' 95,* (1995) 367-374.

[79] M.H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science Vol. 156, Springer-Verlag, 1983.

[80] M. Parashar and J.C. Browne, Distributed dynamic data-structures for parallel adaptive mesh-refinement, *Proc. International Conference for High Performance Computing* (1995) 22-27.

[81] M. Parashar J.C. Browne, C. Edwards and K. Klimkowski, A common computational infrastructure for adaptive algorithms for PDE solutions, *Proc Supercomputing '97* (1997).

[82] E.A. Patrick, D.R. Anderson and F.K. Bechtel, Mapping multidimensional space to one dimension for computer output display, *IEEE Transactions on Computers, C-17* (1968) 949-953.

[83] G. Peano, Sur une courbe qui remplit toute une aire plaine, /em Math. Ann., 36 (1890).

[84] J.R. Pilkington and S.B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, *IEEE Transaction on Parallel and Distributed Systems, 7(3)* (1996) 288-300.

[85] L.K. Platzman and J.J. Bartholdi III, Spacefilling curves and the planar traveling salesman problem, *Journal of the ACM., 36(4)* (1989) 719-737.

[86] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan and D. Gardner, Parallel transient dynamics simulations: Algorithms for contact detection and smoothed particle hydrodynamics, *Journal of Parallel and Distributed Computing, 50* 1998 104-122.

[87] G.J. Pringle, *Numerical Study of Three-Dimensional Flow using Fast Parallel Particle Algorithms*, Ph.D. thesis, Napier University, Edinburgh (1994).

[88] S. Ranka, R.V. Shankar and K.A. Alsabti, Many-to-many communication with bounded traffic, *Proc. Frontiers of Massively Parallel Computation* (1995), 20-27.

[89] W.T. Rankin and J.A. Board, A portable distributed implementation of the parallel multipole tree algorithm, Technical Report No. TR95-002, Duke University, Department of Electrical Engineering, (1995).

[90] J. Reif and S. Tate. Dynamic parallel tree contraction. *Proc. Symposium on Parallel Algorithms and Architectures*, pages 114–121, 1994.

[91] E. Reingold and J. Tilford. Tidier drawings of trees. *IEEE Transaction on Software Engineering*, 7(2):223–228, 1981.

[92] V. Rokhlin, Rapid solution of integral equations of classical potential theory, *Journal of Computational Physics, 60* (1985) 187-207.

[93] K. Ryu and J. Jájá. Efficient algorithms for list ranking and for solving graph problems on the hypercube. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):83–90, 1990.

[94] H. Sagan, *Space-filling curves*, Springer-Verlag, 1994.

[95] J.K. Salmon, *Parallel hierarchical N-body methods*, Ph.D. thesis, California Institute of Technology, 1990.

[96] J.K. Salmon. G.S. Winckelmans and M.S. Warren, Fast Parallel Treecodes for Gravitational and Fluid Dynamical N-body Problems, *Intl. J. Supercomputer Appl., 8* (1994).

[97] H. Samet, Storing a collection of polygons using quadtrees, *ACM Transaction on Graphics, 4(3)* (1985) 182-222.

[98] H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys, 16(2)* (1984) 187-260.

[99] H. Samet, *The Design and Analysis of Spatial Data Structures,* Addison-Wesley, 1990.

[100] H. Samet, *Applications of Spatial Data Structures,* Addison-Wesley, 1990.

[101] F. Sevilgen, S. Aluru, and N. Futamura. A provably efficient, distribution-independent parallel fast multipole method. *Technical Report, NMSU-CSTR-9901*, 1999.

[102] J. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56:156–180, 1999.

[103] J. Sibeyn and I. Stadtwald. List ranking on meshes and networks. *Proc. of Computing Science in Netherlands*, pages 271–280, 1994.

[104] J.P. Singh, C. Holt, J.L. Hennessy and A. Gupta, A parallel adaptive fast multipole method, *Proc. Supercomputing '93* (1993) 54-65.

[105] J.M. Song and W.C. Chew, Multilevel fast multipole algorithm for solving combined field equations of electromagnetic scattering, *Microwave and Optical Technology Letters, 10* (1995) 14-19.

[106] L. Stiller, L.L. Daemen and J.E. Gubernatis, N-body simulations on massively parallel architectures, *Journal of Computational Physics*, 115 (1994) 550-552.

[107] R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[108] V.E. Taylor, R.L. Stevens and K.E. Arnold, Parallel molecular dynamics: Implications for massively parallel machines, *Journal of Parallel and Distributed Computing, 45* (1997) 166-175.

[109] S.H. Teng, Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation, *SIAM Journal on Scientific Computing, 19(3)* (1998) 635-565.

[110] H. Tropf, H. Herzog, Multidimensional range search in dynamically balanced trees, *Angewandte Informatik, 23(2)* (1981) 71-77.

[111] M.S. Warren and J.K. Salmon, A Parallel Hashed Oct-Tree N-Body Algorithm, *Proc. Supercomputing '93* (1993) 12-21.

[112] P.M. Vaidya, An O(n log n) Algorithm for the All-Nearest-Neighbors problem, *Discrete Computational Geometry (4)*, 101-115 (1989).

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

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

[115] I. H. Witten and B. Wyvill, On the generation and use of space-filling curves, *Software-Practice and Experience, 13(6)*, (1983) 519-525.

[116] F. Zhao and L. Johnsson, The parallel multipole method on the connection machine, *SIAM Journal of Scientific and Statistical Computing, 12* (1991) 1420-1437.