Distribution Independent Spatial Data Structures and Algorithms with Applications

by

FATİH ERDOĞAN SEVİLGEN

B.S., Bogazici University, 1990

M.S., Syracuse University, 1996

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

May 2000

Approved ______________________

Professor Srinivas Aluru

Date ______________________

Abstract

Numerous applications in areas such as scientific computing, databases and computational geometry require representation and manipulation of multidimensional point data. A significant challenge with multidimensional data is the negative effect of the data distribution. For example, in the Fast Multipole Method (FMM), the octree data structure is typically used to represent the data. The size of an octree and in turn the run-time of the FMM depends on the distribution of the data and they are unbounded for arbitrary distributions. We refer to such algorithms and data structures as distribution-dependent. This thesis introduces new representation and manipulation methods that lead to distribution-independent algorithms.

Multidimensional points are mapped to one dimension with the goal of preserving proximity information through the use of space-filling curves. We present a simple and distribution-independent technique to implement this mapping for some curves and apply it to applications drawn from databases and parallel computing.

Hierarchical data structures are frequently used to represent multidimensional point data and generally suffer from lack of efficient dynamic operations or distribution-dependence or both. We resolve this shortcoming with a simple data structure called Distribution Independent Adaptive Tree (DIAT). This makes possible the design of efficient algorithms for simple searches and dynamic operations. DIAT trees also have the advantage of supporting advanced access patterns encountered in many applications such as all-nearest neighbor queries and spherical region queries.

DIAT trees and algorithms presented can be used to eliminate the dependence on distribution in many applications and result in faster algorithms. Through this approach it is also possible to improve the run-time of some algorithms by reducing the dependence on the number of dimensions. As evidence, we provide an optimal distribution-independent algorithm for the FMM.

The effect of the data distribution is more severe in parallel computation of the FMM, usually resulting in dynamic data decomposition and load balancing. We developed the first distribution-independent parallel algorithm for the FMM on distributed memory parallel computers, accompanied by rigorous run-time analysis. Our algorithm does not require any dynamic data decomposition or explicit load balancing. Parallel FMM algorithm employs the distributed memory tree accumulation algorithms, we devise. Parallel tree accumulations are of interest by themselves because of their various applications.

The Graduate School
Syracuse University

We, the members of the Oral Examination Committee,
hereby register our concurrence that

Fatih E. Sevilgen

satisfactorily defended his dissertation on

Friday, April 7, 2000

Examiners:

Chilukuri Mohan ______________________

James Royer ______________________

Ernest Sibert ______________________

____

Co-advisors:

Srinivas Aluru ______________________

Kishan Mehrotra ______________________

Oral Examination Chair:

Pin Yuen Chen ______________________

Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my sincere gratitude to Prof. Srinivas Aluru, my primary advisor, for his continuous help, encouragement and support during my study. Prof. Aluru always find time for frequent discussions with me and his creative thinking and insight make our discussions fruitful and interesting. My endeavors would not have been successful without him.

I would also like to thank Prof. Kishan Mehrotra for serving as my co-advisor and for his guidance and help throughout my graduate study. My deepest thanks to Prof. Pin-Yuen Chen, Prof. Chilukuri Mohan, Prof. James Royer and Prof. Ernest Sibert for serving in my dissertation committee. Special thanks are due to several EECS people for their help when they are most needed.

I would like to thank Gebze Institute of Technology for supporting me during my graduate study. My research also founded by Prof. Aluru through NSF CAREER CCR-9702991, by SU Department of Electrical Engineering and Computer Science and by The Graduate School. Thanks are also due to the NMSU Computer Science Department for providing me an enjoyable environment during my long visits.

I am especially grateful to my wife and my son, who were always with me with their constant patience and love. I am also indebted to my parents for their everlasting understanding encouragement and support. I would like to express my appreciation to friends who contributed to my work through their support and motivation. They are too many to name, thank you all.

Chapter 1

Introduction

In many scientific applications, the behavior of a physical system is captured through either 1) a discretization of the space using a finite grid, followed by determining the quantities of interest at each of the grid points/cells or 2) a simulation of the motion of (real or virtual) particles whose evolution can be related to the behavior of the system. The former are known as grid-based methods and the latter are known as particle-based methods. Both methods are typically iterative. Examples of grid-based methods are multigrid and adaptive mesh refinement (AMR) methods. Particle-based methods include the fast multipole method (FMM), molecular dynamics (MD) and smoothed particle hydrodynamics (SPH).

Most of the algorithms developed for both grid-based and particle-based methods are distribution-dependent, i.e., the run-time of the algorithm not only depends on the size of the data but also depends on the distribution of the data. There are three main sources of distribution-dependence: First of all, the data structure that is used for representation of the data may be distribution-dependent which means the size of the data structure depends on the distribution. Secondly, the algorithms to access the data according to the patterns governed by the application may be distribution-dependent. Thirdly, the application itself may require some operations which are distribution-dependent. Although the sources are different, the result is usually the same; The algorithm performs well for some distributions (generally, uniform distribution) but has unknown behavior for arbitrary distributions. Another problem caused by distribution-dependence is the difficulty it imposes in the analysis of algorithms. Because of this difficulty, many algorithms are analyzed for uniform distributions that typically give the best running-time.

However, non-uniform distributions are very important in practice. For example, consider shockwave propagation. The shock front is the most active part in the overall simulation and requires more accurate calculations. This translates into a fine grid refinement requirement close to shock front, if AMR is used for the simulations. On the other hand, for places that the shockwave has not reached yet or where activity is low, a coarse grid is sufficient. Moreover, the region where fine refinement is necessary moves along with the shock front, changing the distribution from one iteration to another.

1.1 Contributions of This Research

The primary contributions presented in this dissertation are the development of the notion of distribution-independence and a comprehensive framework for distribution-independent solutions including a data structure, data access algorithms and an application that use the former two. These contributions are elaborated below:

- Dependence on distribution is classified into three categories: Undetermined dependence on distribution, parameterized dependence on distribution and complete independence from distribution. Because of the unknown behavior of the distribution-dependent algorithms, distribution-independence is very important and should be the ultimate goal of algorithm design.

- Data structures that efficiently support the access patterns required by many applications are a useful tool in designing algorithms for scientific applications. We propose such a data structure, the Distribution-Independent Adaptive Tree (DIAT Tree). The operations common to many applications and supported by the DIAT tree include the ability to retrieve neighborhood information and view global information at varying granularities, apart from maintaining the data structures in the presence of insertions, deletions and changes in data. The size (and hence the memory requirement) of the DIAT tree is independent of the distribution.

  An additional advantage of the DIAT tree is that applications that require the use of multiple methods requiring more than one access pattern can be easily accommodated. For example, in a chemical multi-body interaction problem, separate methods could be employed for accounting various types of forces: Molecular dynamics for short range interactions and the FMM for long range interactions. In some cases, different methods may be used in different algorithms for the same application. For example, FMM and AMR methods are used for the N-body problem. It is difficult to judge the performance of various methods or data structures for the same application owing to the absence of rigorously proven run-times. Using the DIAT tree for such an application enables comparison between different methods that can be applied.

- A unifying algorithm for a class of neighbor queries is presented. A novel algorithm that employs DIAT trees gives solutions to all-nearest neighbor, all-k-nearest neighbor and spherical region queries. The changes in the algorithm for various queries are only limited to three functions used in the algorithm. By using appropriate functions, all three queries can be answered. Rigorous run-time analysis is provided for the first two queries for which the algorithm is efficient and distribution-independent.

  The same algorithm for neighbor queries can also be used for FMM. Once again, using the DIAT tree enables the development of a distribution-independent algorithm for a very important method in scientific computing.

- Developing parallel algorithms for methods in which the computation depends on the distribution of the data is considered a challenge. For example, in FMM, the positions of the particles change from one iteration to the next. The computation depends upon the positions of the particles and thus the workload distribution changes. Due to these difficulties, algorithms usually depend upon heuristic methods to perform data decomposition and entail unstructured and irregular communications. Such heuristic methods make it impossible to rigorously analyze the run-time of the corresponding parallel algorithms.

A new strategy for developing parallel algorithms with rigorous running time is introduced. The strategy is based on designing parallel algorithms that are distribution-independent. Once distribution-independence is achieved, algorithms using only a static data decomposition and requiring no explicit dynamic load balancing could be possible. This in turn would lead to a rigorous analysis of the running time. Such a parallel algorithm for FMM is presented as evidence to the applicability of this strategy. The algorithm is a parallelization of the distribution-independent sequential framework proposed in this dissertation. The algorithm is developed by 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 and basic communication operations such as parallel prefix, all-to-all communication and sorting.

- There are two additional side results presented in this dissertation which are useful as general tools in algorithm design:

1. A simple, distribution-independent and efficient technique for ordering arbitrary and dynamic multidimensional points using space-filling curves is presented. Space-filling curves are used in many applications ranging from scientific computing to databases to map multidimensional points to one dimensional ordering; this is because of their property of preserving proximity information. The technique introduced is applied to applications drawn from databases and parallel computing. Distribution independent algorithms for storing database records involving multiple keys and answering range queries are presented. The use of this technique for parallel domain decomposition and incremental load balancing are demonstrated.

2. An algorithm to compute upward and downward tree accumulations on coarse-grained distributed memory parallel computers is presented. The upward accumulations problem is to aggregate data in the subtree under each node of the tree. The downward accumulation problem is to aggregate data at all the ancestors of each node. Parallel tree accumulations are part of a variety of applications. Parallel tree accumulations on PRAM model are studied extensively and optimal solutions have been provided. However, for distributed memory parallel computers, the proposed solutions are not satisfactory. Algorithms for different architectures such as hypercubes and meshes are studied, but they use communication techniques that are particular to their target architecture. The algorithms proposed in this dissertation achieve similar running times by using architecture independent

primitive operations.

1.2 Organization of the Dissertation

The organization of the rest of this dissertation is as follows:

Chapter 2 provides a brief survey of related work in the literature.

Chapter 3 introduces the concept of distribution-independence and discusses its importance for both designing data structures and algorithms. Three different categories of distribution-dependence are defined and explained in detail using sorting algorithms as examples. This chapter also contains the basic terminology and techniques that are used in the rest of the dissertation. A cell hierarchy is defined and an algorithm to find the smallest cell in the hierarchy containing two given cells is described.

Chapter 4 is on how to use the Space Filling Curves (SFC) for arbitrary multi-dimensional point data. Some of the various ways to define SFCs and the problems in using SFC's for arbitrary distribution of points are presented. A novel technique is proposed to solve these problems using the methods detailed in Chapter 3. The applications of the technique to two separate areas (i.e., databases and parallel computing) are investigated and experimental results are provided. As a database application, a new distribution-independent algorithm for range queries is developed.

Chapter 5 presents a new data structure for multidimensional data called Distribution Independent Adaptive Tree (DIAT). DIAT trees are derived from hyperoctrees. After a brief introduction of hyperoctrees, their disadvantages for arbitrary distributions are identified and DIAT trees are proposed as a solution. DIAT trees are distribution independent and lead to efficient query and update operations. How to construct, search and dynamically update DIAT trees are discussed and algorithms for each operation are provided.

Chapter 6 provides a unified algorithm using DIAT trees for three neighbor queries, namely all-nearest neighbors, all-k-nearest neighbors and spherical region queries. We also discuss the use of DIAT trees together with existing algorithms for closest pair and approximate nearest neighbor queries.

Chapter 7 describes a new sequential algorithm for the Fast Multipole Method (FMM) using DIAT trees. The algorithm presented is both distribution-independent and optimal. A brief overview of the FMM is also presented. Simple strategies to incrementally update the DIAT tree while the particles move are presented as an additional advantage of using DIAT trees.

Chapter 8 is on tree accumulations on coarse-grained distributed memory parallel computers. After a brief description of upward and downward tree accumulations which are abstract operations useful in many applications involving trees, algorithms for computing accumulations that are both rigorously analyzable and practical are presented. The model of parallel computation assumed in this dissertation is also presented in this chapter.

Chapter 9 introduces a new parallel algorithm for the FMM. The algorithm presented is the first distribution-independent parallel algorithm for the FMM on distributed memory parallel computers. Four steps of the algorithm namely, tree construction, multipole expansion calculation, partial local expansion calculation and local expansion calculation are discussed in detail, accompanied by rigorous run-time analysis.

Chapter 10 summarizes the research presented in this dissertation, highlights major contributions, and discusses possible extensions and directions for future research.

.

Chapter 2

Literature Review

2.1 Space-Filling Curves

Space-filling curves (SFC's) are mappings from multidimensional data to one dimension. Popular curves designed with the goal of preserving proximity information during this mapping include the Z-curve [70], the graycode curve [40] and the Hilbert curve [51, 78]. A detailed discussion about space-filling curves can be found in [87].

The proximity information preserving properties of space-filling curves are studied experimentally by Abel *et al.* [1] and Jagadish [57]. In [57], different criteria to evaluate SFC's for the class of database applications are examined. In their experimentation, Abel *et al.* [1] used operations drawn from Geographical Information Systems. Both these two studies conclude that Z-curve and the graycode curve have similar performance while Hilbert curve is better for most of the experiments. They also point out the major weakness of the Hilbert curve, which is the lack of inexpensive mapping algorithms (see [22, 24, 64, 107] for mapping algorithms).

Some analytical results on space-filling curves are given by Jagadish [56], Asano *et al.* [15] and Moon *et al.* [69]. Jagadish suggests the average number of *runs* (i.e., continuous SFC segments) in a specified region as a measure of how well proximity information is preserved. He gives closed form expressions for average number of runs in a square region for Hilbert curve on a two-dimensional grid. He proves that the expected number of runs approaches to 2 for a 2×2 square to show that the mapping is optimal for this case. Moon *et al.* [69] improve these results and give asymptotic solutions for any given region of an arbitrary shape. They also analytically show that the Hilbert curve achieves better performance in terms of runs than the Z-curve. Asano *et al.* [15] propose new SFC's which are optimal for certain conditions. While these results give an estimate for the performance of real applications, researchers agree that they should not be used as sole performance measure.

There has been a significant interest in the scientific community in using space-filling curves for a variety of applications. Several researchers have suggested and used space-filling curves for secondary storage in databases [39, 40, 54, 57, 72]. They are used for reducing the bandwidth of digitally sampled signals [22], for reducing pen plotting time [10] and for generating graphics displays [77]. Using space-filling curves, heuristic algorithms have been designed for NP-hard problems such as the traveling salesperson problem [80] and digital halftoning [14]. Pilkington *et al.* [79] 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.* [106].

Storing database records using space-filling curves is first suggested by Orenstein *et al.* [72]. 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 [32]) afterwards. Since efficient storage structures are available in one dimension, it is supposed that the resulting multidimensional storage structures are also efficient.

Range query is a common type of query in database applications. Range queries using SFC's has been studied by many researchers including Orenstein *et al.* [72] and Faloutsos [39]. In [72], some heuristics for reducing the run-time for range queries are presented. 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.

There is currently significant interest in the research community for using space-filling curves for graph partitioning. Ou [75] and Pilkington *et al.* [79] have shown that space-filling curve based partitioning techniques provide a partition of quality comparable to Recursive Coordinate Bisection. Further, it beats Recursive Coordinate Bisection in running time for almost all experiments.

Interleaving the bits of the coordinates of the data points has been used for mappings based on space-filling curves. This method relies on the granularity of the grid used by the SFC. For non-uniform data distributions, all the algorithms that use SFC's 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. In Chapter 4, we propose a distribution-independent technique to use SFCs and present applications of the technique in databases and graph portioning.

2.2 Hierarchical Tree Data Structures

Hierarchical tree data structures are a class of multidimensional data structures based on recursive decomposition of the region containing the data. There are many different kinds of data structures proposed in the literature including *k-d* trees [18], quadtrees [41, 92], cell trees [30], R-trees [59], fair split trees [60], BD trees [71] and *k-d* tries [73]. A comprehensive survey of hierarchical tree data structures is given in [90, 94].

In this dissertation, the focus is on hyperoctrees. They are known as quadtrees in two-dimensions and octrees in three-dimensions. Popular application areas of hyperoctrees include databases [41, 73], scientific computing [49, 106] and computer graphics [33] (see also [91] for various applications). Hyperoctrees and applications employing them are distribution-dependent [9] because recursively subdividing a region may result in just one occupied region for an arbitrarily large number of steps. In terms of the tree, there can be an arbitrarily long path without any branching. In [36], Devroye shows that hyperoctrees often perform well for the range of distributions encountered in practice even though they are distribution-dependent.

Compressing such paths without branching is proposed by Clarkson [30] in the context of the all-nearest neighbors query. If the properties of interest to the problem at hand depend only on the points, all the nodes along such a path essentially contain the same information. Thus, compression does not cause any loss of information. Clarkson's *cell tree* is a distribution-independent alternative to hyperoctrees. He presents a randomized algorithm to construct such a tree for n points in $O(n \log n)$ expected time. An $O(n \log n)$ time deterministic algorithm for the all-nearest neighbors problem is presented by Vaidya in [105]. Though Vaidya does not explicitly mention it, the computational tree implied by his algorithm is equivalent to a cell tree. This analogy can be used to derive an $O(n \log n)$ deterministic algorithm for cell tree construction.

In this dissertation, the term *compressed hyperoctree* is used instead of cell tree. Although compressed hyperoctrees are distribution-independent, their height is $\Omega(n)$ in the worst-case. Thus, algorithms that require top-down or bottom-up traversal of compressed hyperoctrees such as searches are not efficient [9]. Bern [20] proposes another $O(n \log n)$ time deterministic algorithm for construction of compressed hyperoctrees and uses centroid decomposition [28] for $O(\log n)$ time searches. Callahan and Kosaraju's *fair split trees* achieve similar run-time complexities by allowing non-cubical regions [26].

A different technique to rectify the distribution-dependent nature of the hyperoctrees is used by Ohsawa *et al.* [34, 71] and resulted in BD-trees. 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. Arya *et al.* [12] improve these results and propose a height balanced tree, BBD-tree. An empirical performance comparison of some spatial data structures and BD trees is given in [35].

Algorithms for dynamic operations such as insertions and deletions on hyperoctrees are presented by Overmars [76]. However, these are just straightforward algorithms having unbounded running time in the worst case. Optimal dynamic operations for compressed hyperoctrees and similar data structures are presented in [19, 21, 26, 93] by using auxiliary data structures, such as topology trees [42] or dynamic trees [101]. However, these auxiliary data structures are of considerable complexity. In Chapter 5, we combine compressed hyperoctrees with simple balanced binary search trees and call the resulting data structure *Distribution Independent Adaptive Tree (DIAT Tree).* DIAT trees are distribution-independent and admit distribution-independent and efficient operations such as searches, insertions and deletions.

2.3 Neighbor Queries

All-nearest neighbors query is to find the point closest to each point in a set of n points. It is a simpler special case of the all-k-nearest neighbor query for $k = 1$. Clarkson presents a deterministic algorithm to answer an all-nearest neighbors query using the compressed hyperoctree [30]. The running time of the algorithm is linear in the number of points. Combining with the randomized compressed hyperoctree construction algorithm he presents, the total expected running time of an all-nearest neighbors query is $O(c^d n \log n)$ where d is the number of dimensions and c is a constant.

Vaidya [105] presents a deterministic algorithm to compute the all-nearest neighbors. The worst-case running time of Vaidya's algorithm is $O((cd)^d n \log n)$. Callahan *et al.* [25] have a different approach using their fair split tree. They compute a well-separated pair decomposition. The nearest neighbor of each point can be determined directly from this decomposition. The overall running time of their algorithms is same as Vaidya's.

Callahan *et al.* and Vaidya extend their algorithms for all-nearest neighbors to answer all-k-nearest neighbors queries as well. While the running time of Vaidya's algorithms is $O((cd)^d(n \log n + kn \log k))$, the algorithm in [25] has the running time of $O(c^d kn)$ for a given well-separated decomposition.

In Chapter 6, we improve the running time for both neighbor queries. We employ DIAT trees and reduce the dependence of the running time on the number of dimensions. The worst-case running time of the all-nearest neighbors algorithm is $O(dn \log n)$ by assuming an extended computational model in which logarithm and bitwise-or operations can be calculated in constant time. Another distinct advantage of the algorithm is that the repetitive application of neighbor queries in a dynamical environment can be handled by only paying the cost of dynamic operations plus the query processing which is linear to the number of points.

Approximate closest-point queries has been studied in [12, 13, 20, 27, 31, 55]. To improve the query time, data is first preprocessed by using a variety of data structures. Arya *et al.* [13] present a query algorithm with $O((1/\varepsilon)^d \log n)$ running time and $O((1/\varepsilon)^d n)$ preprocessing time. The dependence on ε is later reduced by Clarkson [31] and Chan [27]. Arya *et al.* [12] presents optimal $O(dn)$ preprocessing at the cost of $O(d^d \log n)$ query time.

Dynamic closest pair problem is to maintain the pair of points minimizes inter-point distance under insertions and deletion. It is discussed in [21, 26, 93]. Callahan *et al.* [26] give an algorithm with running time $O(d \log^2 n)$ for both insertion and deletions. Schwarz *et al.* [93] presents an $O(c^d \log n)$ amortized time algorithm for a restricted case when only insertions are allowed.

2.4 Parallel Tree Accumulations

Parallel tree accumulations are abstract operations on trees used as a part of a variety of applications ranging from graph problems to scientific computing. Simple application of tree accumulations include broadcast and gather operations on trees and calculation of the depth of the nodes and sizes of the subtrees [65]. They are also used for determining minimum covering and maximal independent sets, drawing and labeling trees and evaluating expression trees [37, 45, 50, 65, 66, 84]. In the Fast Multipole Method, the multipole expansion and local expansion calculations require upward and downward accumulations respectively [95].

A related problem is that of parallel tree contraction which is introduced by Miller *et al.* [65]. 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 popular methods of parallel tree contraction on PRAM model. They are proposed by Miller *et al.* [65] and Kosaraju *et al.* [60], together with their applications. The first method is based on repeatedly raking all the leaf nodes and compressing the chains obtained after each rake operation. In the second method, every other leaf, in the order from right to left, are raked with their parents.

Parallel tree accumulations on PRAM model using the aforementioned two tree contraction algorithms are studied in [2, 43, 60]. Optimal $O(\log n)$ time algorithms using $O(n/\log n)$ EREW PRAM processors are presented. Using tree contraction methods, the application of a proper accumulation function when the nodes are removed results in an upward accumulation algorithm for the root node. For upward accumulation at the other nodes, an additional expansion phase is used [47, 65]. The downward accumulation can also be calculated similarly using two phases. Gibbons [46] presents a CREW PRAM algorithm for downward accumulations that runs in

$O(\log h)$ time where h is the height of the tree.

Tree contractions on distributed memory computers are studied by many researchers [16, 44, 53, 86]. They propose algorithms for different architectures such as hypercubes and meshes as an extension of one of the two popular techniques on the PRAM model. The algorithms use communication techniques that are particular to their target architecture to improve the running time. In Chapter 8, we present architecture-independent algorithms on distributed memory parallel computers for upward and downward accumulations. The algorithms are both rigorously analyzable and practical using some simpler operations such as sorting and parallel prefix. The upward accumulation algorithm requires only contraction phase. For many downward accumulation calculations on the same tree, only one contraction phase is sufficient.

2.5 Fast Multipole Method

The Fast Multipole Method (FMM) is a robust technique for the rapid evaluation of the combined effect of pairwise interactions of n objects. The method was first developed by Rokhlin [85] to solve integral equations and shot into prominence with application of the FMM [49] to solve the N-body problem by Greengard and Rokhlin. Over the past decade, the FMM has become a standard numerical method. The essential idea of the FMM is to approximate interactions between distant clusters of particles instead of computing individual pairwise interactions. Similar idea is also used by Appel [11] for N-body simulations. In his approach, each cluster is considered as a particle at the center of mass of the cluster, instead of using multipoles.

The FMM is applied to solve a number of problems in fluid dynamics, molecular dynamics, elliptic partial differential equations, and even computer graphics [23, 29, 38, 81, 85, 88, 102]. Although the basic method is same, interactions are different. Thus, for each application, new equations should be derived.

Most FMM methods use the octree data structure [49, 62, 83, 100, 104, 106, 108]. The octree-based FMM is often incorrectly claimed to reduce the computation required per iteration from $O(n^2)$ to $O(n)$. However, it is shown that the run-time depends upon the distribution of the particles [4, 5]. The run-time of the FMM is proportional to the number of nodes in the octree data structure which in turn depends upon the distribution of the particles.

Aluru *et al.* [7] use compressed octrees and modify the FMM algorithm to achieve the run-time complexity of $O(n)$ independent of the distribution of particles. Callahan *et al.* [25] present another FMM algorithm using fair split trees. Once again, the FMM algorithm runs in $O(n)$ time given the tree. These two are the only sequential FMM algorithms known to be distribution-independent.

Considerable research effort is directed at developing parallel implementations of the FMM. Most of the parallelizations employ the octree-based FMM computation, and thus inherit the distribution-dependent nature of the algorithm [62, 83, 104, 106, 108]. The run-times of these algorithms are either not analyzed or analyzed only for uniform distribution due to the dependence of the computation on the distribution of the data sources, usually resulting in dynamic data decomposition and load balancing problems. The only exception is an algorithm developed by Teng [104], in which he defines a communication graph to capture the communication pattern underlying the FMM computation and presents an algorithm to partition the 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 *et al.* develop 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 [25]. 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. In Chapter 9, we propose the first provably efficient and distribution-independent parallel algorithm for the FMM on distributed memory parallel computers. The algorithm is presented in an architecture-independent fashion, using only well-understood and basic communication operations such as parallel prefix, all-to-all communication and sorting. It uses only a static data decomposition and does not require any explicit dynamic load balancing, either within an iteration or across iterations.

A variation of the interaction computation is given by the Barnes-Hut algorithm

[17], 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, 89, 100]. These methods are also dependent on the distribution.

.

Chapter 3

Basic Definitions and Concepts

3.1 Distribution-Independence

A number of applications require representing and manipulating spatial data such as points and grids. Examples of these applications can be found in computational science, databases, geographical information systems and computational geometry. An important problem in designing algorithms for these applications is to design a data structure that provides the ability to process and store the dynamic spatial data and retrieve information about the stored data.

There are widely recognized difficulties in both representation and manipulation of spatial data. For example in the FMM, the particles should be organized into hierarchy of clusters. Typically, hyperoctrees are used to cluster particles. If the particles are uniformly distributed, the size of the hyperoctree is linear in the number of particles and optimal running time for the FMM is achieved. However, non-uniform distributions are very common in applications using FMM such as cosmological simulations. The run-time of the FMM and the size of hyperoctrees are unknown for arbitrary distributions.

A data structure is called *distribution-independent* if its size is a function of the number of points only and is not dependent upon the distribution of the input points. Similarly, an algorithm is called *distribution-independent* if the run-time of the algorithm is independent of the distribution of particles and rigorously analyzable. Otherwise, the data structure or the algorithm is said to be *distribution-dependent*. Both hyperoctrees and the FMM using hyperoctrees are distribution-dependent.

Distribution-independence is a desirable property for data structures and algorithms because otherwise unpredictable behavior is imminent. The algorithm may be proved to perform well for some specific data distributions but the run-time may be unbounded for arbitrary distributions. Even the run-time may not be stated definitely for the distributions other than those specific ones. This is called *complete distribution-dependence* in which dependence on distribution does not allow analysis of the run-time of the algorithms or size of the data structure. On the other hand, the distribution can be parameterized in some cases. For example, in the analysis of an all-nearest neighbors algorithm, Clarkson [30] uses the fraction of the longest distance to the shortest distance between any two points. The fraction represents how badly the points are distributed: as the distribution becomes worse, the number becomes bigger. Such a dependence is called *parametric distribution-dependence* where the run-time is a function of a parameter describing the distribution.

Clearly, not all distributions can be modeled on a given machine due to precision limits. Precision limits can be improved by implementing multiple-precision operations in software to accommodate a wider range of distributions. It is true that the running time of basic operations will then depend upon the precision used, which in turn depends on distribution. Thus, the running time of any algorithm ultimately depends on the distribution. However, an algorithm in which the number of operations executed depends upon the distribution is undesirable. An analogy can be drawn to a sorting algorithm whose running time depends on the size of the numbers to be sorted. The complexity of a sorting algorithm is $O(n \log n)$, provided basic operations on the numbers to be sorted (like comparison, copying) can be performed in constant time. The complexity of the algorithm does not remain $O(n \log n)$ if this assumption is not valid. However, there is a distinct advantage to having a sorting algorithm in which the number of operations is independent of the size of the input numbers. Such an algorithm can sort 128-bit numbers on a machine with 128-bit words with the same number of operations as it sorts 32-bit numbers on a 32-bit word machine.

On the contrary, consider the following recursive sorting algorithm to sort n integers: Find the maximum (*max*) and minimum (*min*) values among the integers. Calculate the integer *mid* which is in the middle by adding *max* and *min* and dividing by 2. Partition the integers into two, the integers larger than *mid* are in one partition and the integers smaller than *mid* are in the other. Recursively sort the partitions using the same procedure. To analyze the algorithm, let us examine the size of the binary tree representing the recursive calls. In this recursion tree, some nodes may have only one child where all the integers fall into one partition. Such nodes can be found anywhere in the tree. Even chains of such nodes are possible. The size of the tree depends on the number of nodes having only one child. It is possible to generate distributions for any number of such nodes, so the size of the tree, which is exactly the number of recursive calls, is unbounded for arbitrary distributions. Therefore, the algorithm is distribution-dependent. If we cannot find a parameter and show the dependence of the running-time rigorously, the distribution-dependence of the algorithm is complete. However, the height of the recursion tree is bounded by the fraction δ of largest difference between two integers to smallest difference. The amount of work on each level cannot be more than n. Thus, the running time is $O(\delta n)$ where δ is the parameter describing the distribution. Consequently, the sorting algorithm is parameterized distribution-dependent.

3.2 Cell Hierarchy

Call any d-dimensional hypercubic region as a *cell* for any fixed d. The *cell hierarchy* is defined recursively on a special cell called the *root cell* by the following: The root cell is in the cell hierarchy. If a cell is in the cell hierarchy, then the 2^d equal-sized hypercubic subregions obtained by bisecting along each coordinate of the cell belong to the cell hierarchy. Note that for simplicity of presentation, the following convention is adopted: After a cell hierarchy is described, the term cell refers only to cells in the cell hierarchy.

The root cell uniquely specifies the cell hierarchy. However, the choice of the root cell is not unique for many applications. For example, for point based applications, any cell containing all the points can be considered as a root cell and each of them results in a different cell hierarchy. Even the smallest cell containing all the points is not unique for an arbitrary point set.

In the definition of cell hierarchy, the stopping condition for recursive subdivision is not specified, although applications usually dictate such conditions. Thus, a cell hierarchy contains countably infinite number of cells. For computer graphics applications, further subdivision is not necessary when the required resolution is obtained.

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 *length of a cell* C, denoted $length(C)$, to be the span of C along any dimension.

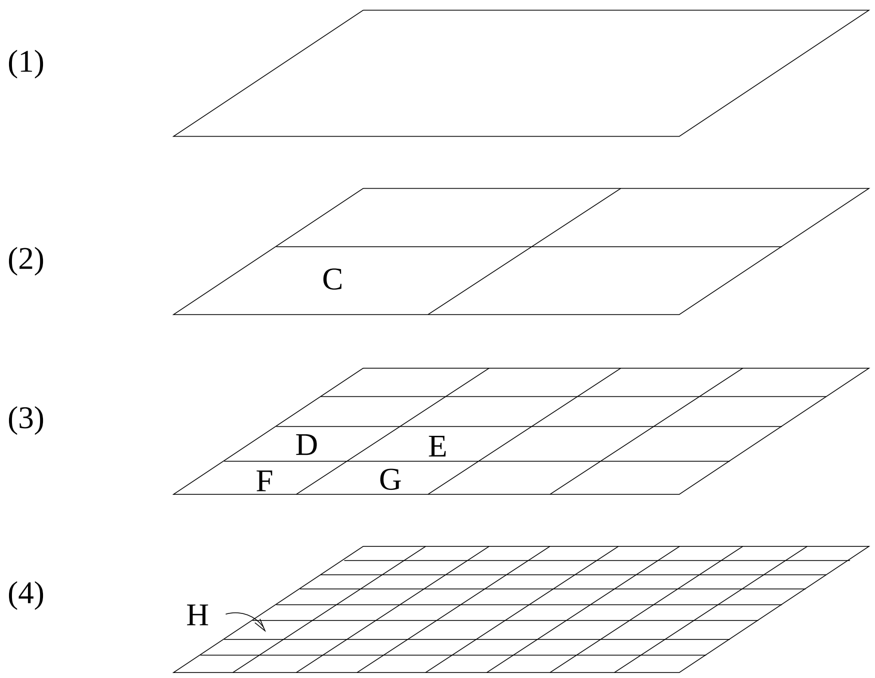

Figure 3.1: Illustration of hierarchy of cells in two dimensions [62].

A two dimensional example of such a hierarchy is given in Figure 3.1. Each subsequent layer represents a subsequent subdivision process. The cell in the first layer is the root cell. The cells D and E are disjoint but C and H are not. Indeed, H is a subcell of C. The cells D and E are immediate subcells of C. All the cells in a layer have the same length.

3.3 Smallest Cell Technique

Consider a cell hierarchy. The smallest cell technique provides a constant time solution to the following problem: Given two cells, find the smallest subcell of the root cell that contains the cells as subcells. The technique is first proposed by Clarkson [30] to find the smallest cell containing two points in the context of nearest neighbor problem. We extend the technique to find the smallest cell containing two cells. While the points can be considered as zero length cells, the same results apply for the points as well. The smallest cell technique is used to find the lowest common ancestor of two nodes in hyperoctrees and sorting the points according to a space-filling curve in subsequent chapters.

If one of the given two cells is a subcell of the other, there is no need for further calculation. The cell with greater length is the smallest cell being searched. The smallest cell technique is used otherwise.

A rectangular box can be uniquely specified by two opposite corners on one of its main diagonals. For example, in two dimensions, if the corners are (x_1, y_1) and

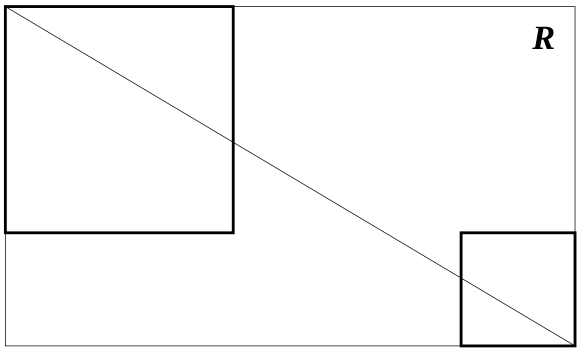

Figure 3.2: Illustration of the smallest rectangle R containing two cells. The diagonal of R joining a corner of each cell is shown.

(x_2, y_2), the rectangle formed by (x_1, y_1), (x_1, y_2), (x_2, y_1) and (x_2, y_2). Consider the smallest box R that contains the given two cells. The box R is unique and the line segment joining one corner of each cell that gives the largest distance between the two cells is a diagonal of R. A two dimensional example is provided in Figure 3.2, reproduced from [3].

The smallest cell enclosing the two cells is also the smallest cell that encloses the box R for which the algorithm in [3] is used. First, some terminology:

A line is called a *k-boundary* if it contains an edge of a cell of length $\frac{l}{2^k}$. Any *boundary* is perpendicular to one of the axes. A *boundary* can be specified by the axis to which it is perpendicular and the distance of the *boundary* from the origin. A *k-boundary* is at a distance of $i\frac{l}{2^k}$ $(0 \leq i \leq 2^k)$ from the origin. There are $2^k + 1$ *k-boundaries* perpendicular to each axis and spaced $\frac{l}{2^k}$ apart. Any *k-boundary* is also a *j-boundary* for every $j > k$. 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 3.3.

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 3.3). To determine this, we can examine boundaries perpendicular to each coordinate axis in turn and find the smallest such integers for each coordinate. The smallest over all coordinates is the number k which is to be used in calculation the size of the smallest subcell.

Without loss of generality, consider boundaries perpendicular to the x-axis. Let s be the smallest integer such that an *s-boundary* perpendicular to the x-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 where x_{min} and x_{max} are the maximum and minimum values of the interval on the x-axis.

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.*

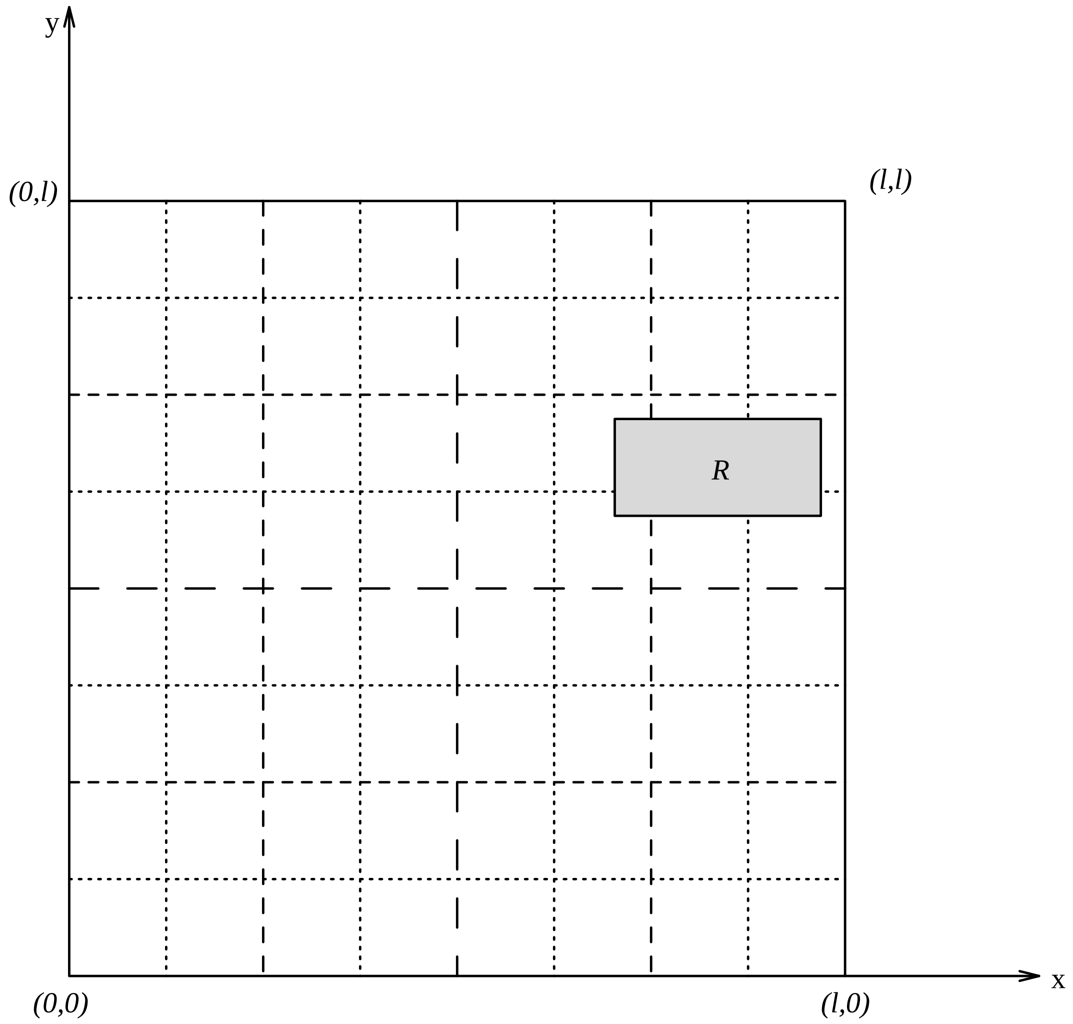

Figure 3.3: A *root cell* of length l and a rectangle R. The big dashed lines are 1-*boundaries*, the small dashed lines are 2-*boundaries* and the dotted lines are 3-*boundaries*. 2-*boundaries* are also 3-*boundaries* and 1-*boundaries* are also 2-*boundaries* and 3-*boundaries*.

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 than 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 $(i+2)\frac{l}{2^t} < x_{max}$. Thus, $(x_{max} - x_{min}) > (i+2)\frac{l}{2^t} - i\frac{l}{2^t} = 2\frac{l}{2^t} = \frac{l}{2^{t-1}}$, 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 the *s-boundary* passing through R is also a *t-boundary*. By Lemma 2, 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 $t - s = \log_2(1 + \{a' \oplus (a'-1)\}) - 1$.

The algorithm to find the smallest cell contained in the root cell enclosing the box R is given in Figure 3.4. Since all the operations in the algorithm take constant time, the smallest cell can be determined in $O(d)$ time.

Algorithm 1 Smallest_Cell (R)

for each dimension i

min = smallest coordinate of R along the dimension i

max = largest coordinate of R along the dimension i

$t = \lceil \log_2 \frac{l}{x_{max} - x_{min}} \rceil$

$a_1 = \lceil \frac{2^t x_{min}}{l} \rceil$

$a_2 = \lfloor \frac{2^t x_{max}}{l} \rfloor$

if $a_1 = a_2$

$a = a_1$

else

a = the even integer among a_1 and a_2

$s = t - \log_2(1 + \{a' \oplus (a' - 1)\}) - 1$

k = smallest of s values among all dimensions

size of the subcell $l_c = \frac{l}{2^{k-1}}$

for each dimension i

position of the subcell along dimension $i = \lfloor \frac{min}{l_c} \rfloor$

Figure 3.4: The algorithm to find the smallest cell enclosing the box R.

.

Chapter 4

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.

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 changes 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.

This chapter contains a simple and distribution-independent technique for ordering arbitrary and dynamic multidimensional points using space-filling curves. The algorithm is based on the smallest cell technique presented in Chapter 3. The technique is applied in two different areas: databases and parallel computing. Two distribution independent algorithms are presented for databases involving multiple keys: For storing database records and for answering range queries using space-filling curves. Application of the technique to parallel graph partitioning, domain decomposition and incremental load balancing are also presented.

4.1 Introduction

This section consists of a brief discussion on the notion of space-filling curves and description of some of the popularly used curves. First, let us restrict our attention to 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 [15]. An intuitive recursive description suffices for the rest of the chapter.

Z-Curve. The Z-curves [70] for $2{\times}2$, $4{\times}4$ and $8{\times}8$ grids are shown in Figure 4.1. 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 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

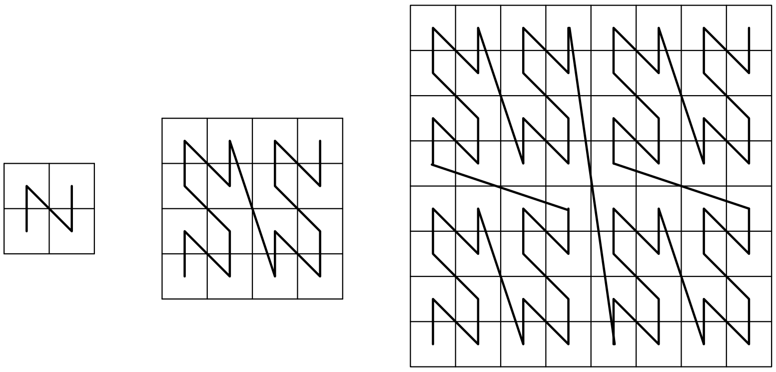

Figure 4.1: Z-curve for grids of size 2×2, 4×4 and 8×8.

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 [39] 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 4.2). The curve can be described recursively as follows: Place four curves of the previous resolution in the four quadrants. Flip the 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

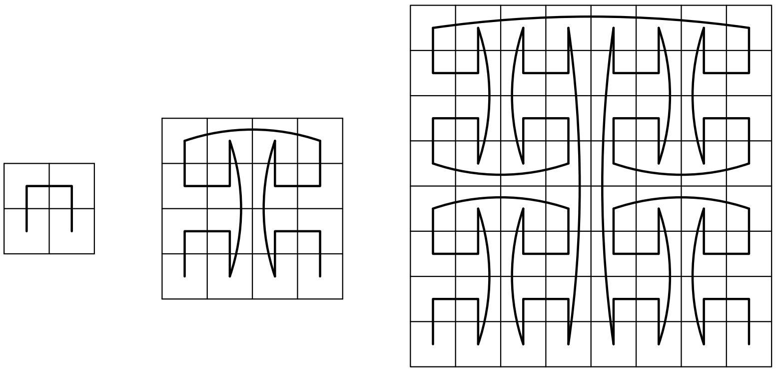

Figure 4.2: Graycode curve for grids of size 2×2, 4×4 and 8×8.

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 [51, 78] is a smooth curve that avoids the sudden jumps present in the Z and the Graycode curves (Figure 4.3). 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, a $2^{k-m+1} \times 2$ template can be defined and used 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

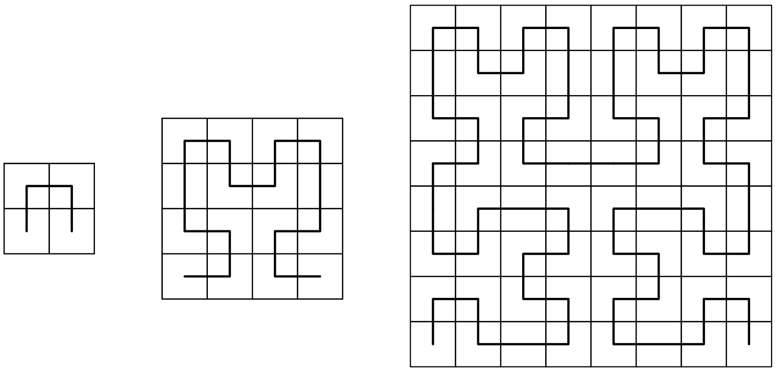

Figure 4.3: Hilbert curve for grids of size 2×2, 4×4 and 8×8.

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 [57].

4.2 Ordering Multidimensional Points

Consider the problem of ordering n points in the order in which a particular space-filling curve visits them. Square grids in two dimensions are used 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, the following approach is adopted: Start with the root cell. 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, let us assume 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 (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

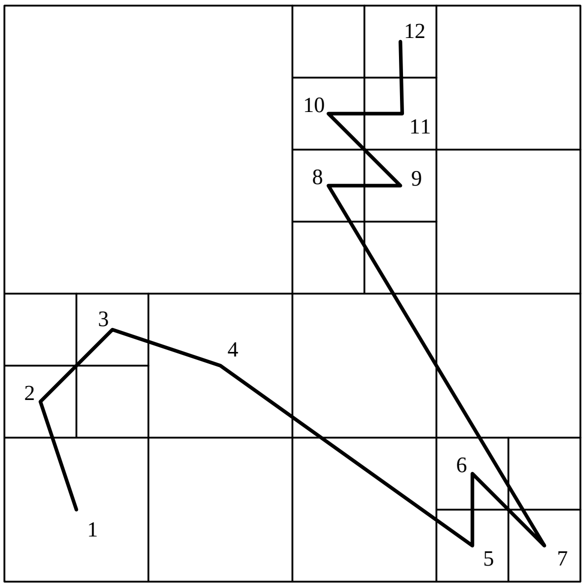

Figure 4.4: Z-curve for a collection of 12 points using the hierarchical subdivision approach.

one point, recursively subdivide it using the same method. This hierarchical subdivision is identical to the quadtree discussed in the next chapter. An example of this procedure is shown in Figure 4.4. This modification may significantly improve the run-time in practice but it does not change the distribution-dependent nature of the algorithm. In the rest of the section, an algorithm that is efficient in practice and has a run-time that is a function of the number of points alone is described.

Suppose two points are given with the goal of determining which of the two is first visited by the space-filling curve. Note that the order in which the space-filling curve visits the points is independent of the position and the presence of other points. First determine the smallest subcell of the root cell that contains these two points.

If the smallest subcell is divided 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 can be determined if the order in which the four immediate subcells of the smallest subcell is visited by the curve is known. 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 to order n points according to a space-filling curve is as follows: 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 such a comparison routine is available, any sorting algorithm orders the points. In the following, such a technique for the Z-curve and graycode curve is described.

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 4.1). Once the smallest subcell and the position of two points in that subcell is determined, the order of these points on the resulting Z-curve can be found immediately. 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. Let us refer to the former as $\cap$-order and the latter as $\cup$-order. Each row of cells in a grid has a unique order. Assume the rows are numbered 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, 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, 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 $O(dn \log n)$.

4.3 Applications of Space-filling Curves

4.3.1 Database Storage and Range Queries

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 [32]. 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. Each such record can be represented 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.

For storing database records with multiple keys, the technique developed in the previous section is employed; First map each key field to a coordinate and view the records as points in k-dimensional space. 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 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.

Algorithm 2 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 4.5: A simple algorithm for range queries.

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 the mapping mentioned above 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).

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, 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, separate range queries with the immediate subcells as search regions and associated query subregions are generated. Pseudo code for this algorithm is given in Figure 4.5.

To report the data points in a cell, the procedure is as follows: 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 the first point in the cell is found, report successive points until the curve 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

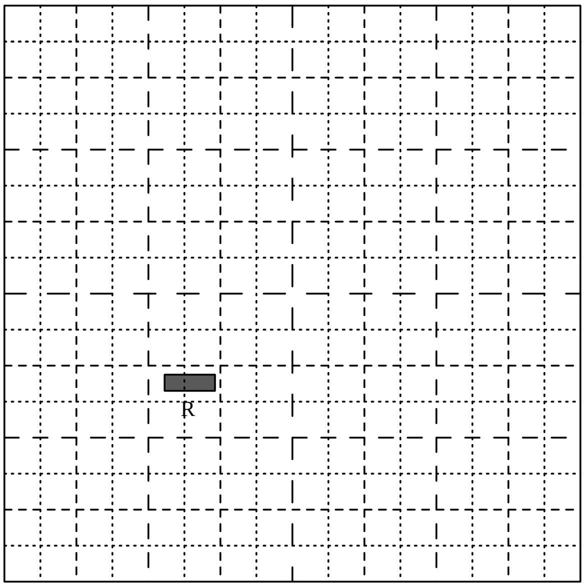

Figure 4.6: A two-dimensional domain and a query region R. The different numbered boundaries are shown using different styles of lines. The largest dashed lines are 1-boundaries and the smallest dashed lines are 4-boundaries.

query region. The effort spent until this point is wasted. In Figure 4.6, 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 find the 4-boundaries that divide the query region. This is a small example for illustration, 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. The query region is divided 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

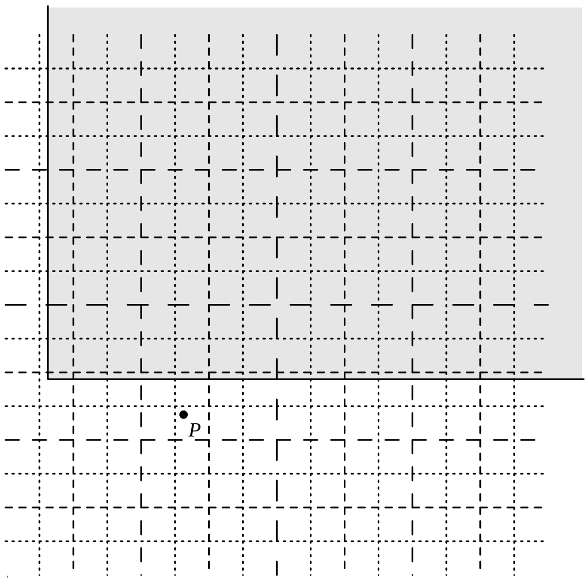

Figure 4.7: A query region and a point P close to its boundary. To determine that P is not in the query region, the search region should be broken into very small cells.

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. During the execution of the algorithm, the cell containing this point at the boundary is divided into finer and finer granularities to find subcells that do not intersect but are completely inside or outside of the query region. In Figure 4.7, the search region is subdivided until the query region and the point with the boundaries shown with dotted lines are separated.

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.

The following terminology is to be used in understanding the algorithm which eliminates large portions of the search region and at the same time guarantees bounded running time: 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 a *search segment* which is a continuous segment of the space-filling curve that contains points within this search cell is maintained. 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.

The first problem is solved 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 the smallest cell that contains the query region is divided, 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, it is necessary to ensure that at least one point which 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. This is accomplished by examining the search segment sequentially and reporting all the points that are in the query region until the first point that is not in the query region is found. Then the largest cell that contains this point but does not intersect the query region is found. 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 4.8. Let us call the **Range_Search** routine with three parameters: query region, search cell and search segment. During the execution of the algorithm, divide the search cell is divided into search subcells and the query region is split into query subregions, one for each search subcell as in the simple algorithm. However, before the recursive calls an effort is spent to eliminate as much of each search subcell and the associated search subsegment as possible. Firstly, 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

---

Algorithm 3 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
                         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 4.8: A distribution-independent algorithm for range queries.

search subcell. Secondly, traverse the points in the search subsegment and report them until the first point not in the query subregion is found. Thirdly, 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 $O(n)$ because it is possible to arrange the points and set up the query region so that the query region is empty and only one point, out of the query region, is eliminated at each iteration. 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. The algorithm presented 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 presented is expected to perform well in practice.

4.3.1.1 Experimental Results

To verify the distribution-independent nature of the algorithm and to show the algorithm performs well in practice, several experiments in two dimensions using the

Z-curve were performed. The algorithm was tested with both uniformly and non-uniformly distributed points. For uniform distribution sample, points are uniformly distributed in both dimensions. As a non-uniformly distributed sample, 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, are chosen. For both distributions, the sample size was $5,000,000$ data points in a unit square.

To test the algorithm, $10,000$ query regions were randomly generated. The lengths of the query regions along the x-axis were uniformly distributed between 0 and 0.05. 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, 5 different aspect ratios were tried: $1, 1/2, 1/4, 1/8, 1/16$. This ensures that many different sizes and shapes of query regions were generated. The locations of query regions determined using the same distribution as the data points.

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, the percentage of useful blocks among all the blocks accessed and the percentage of points in the query region among all the points accessed are used as performance measures.

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).

First two factors were chosen as variants in the experiments. The performance of the algorithm is shown as the number of points in query regions or area of query regions change. For each experiment, the information about performance measures, the fraction of useful points and blocks accessed, are collected. To eliminate minor fluctuations, the performance results are averaged in small ranges (e.g. interval of 100 points).

Figure 4.9 contains the resulting graphs for uniform and non-uniform distributions. The aspect ratio is 1 (i.e., square query regions) and block size is 10 for this experiment. There is a rapid increase in the performance with the increase in the number of points and the area for either distribution. The small fluctuations 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 100, the percentage of blocks that are actually needed is more than 75% and the percentage of points in the query region is more than 65%. 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.

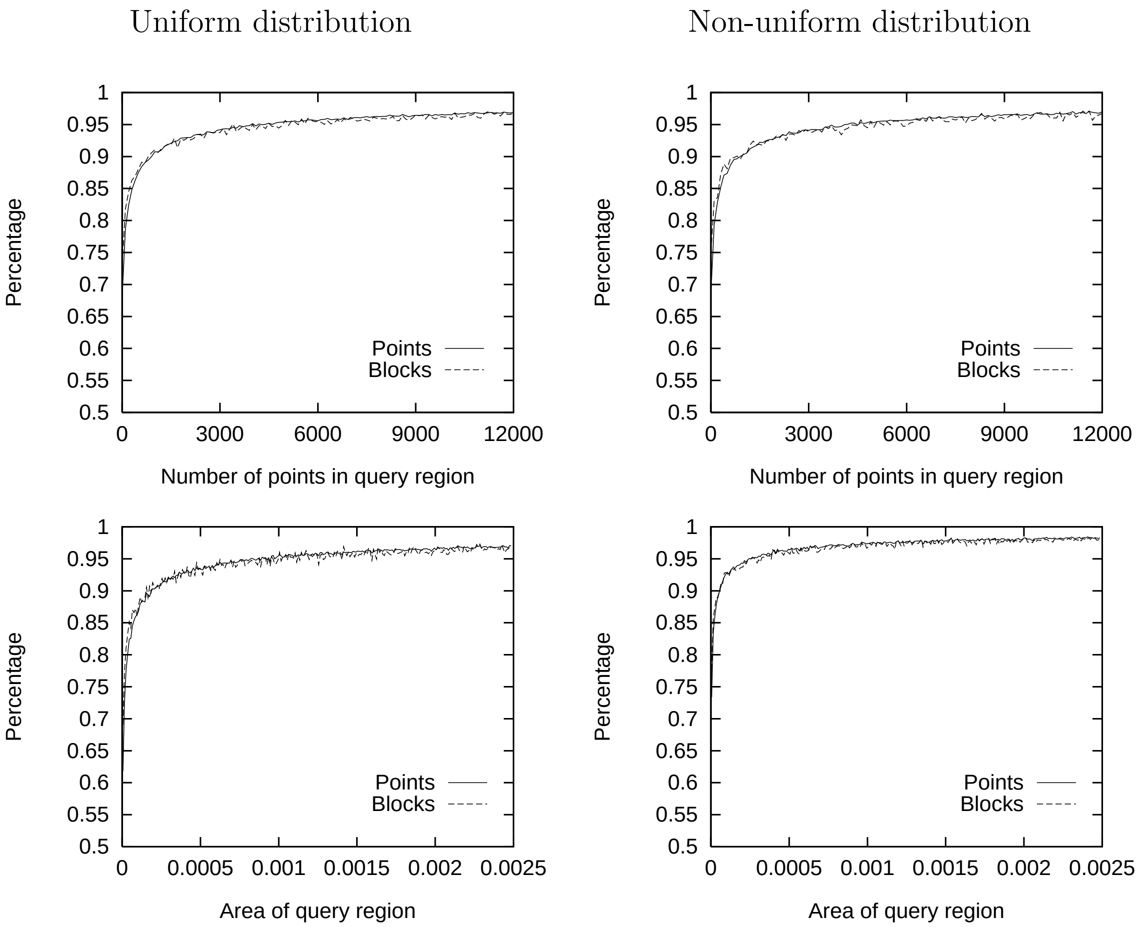

Figure 4.9: Number of points in the query region and area of the query region versus percentage of useful points and blocks for uniform and non-uniform distributions.

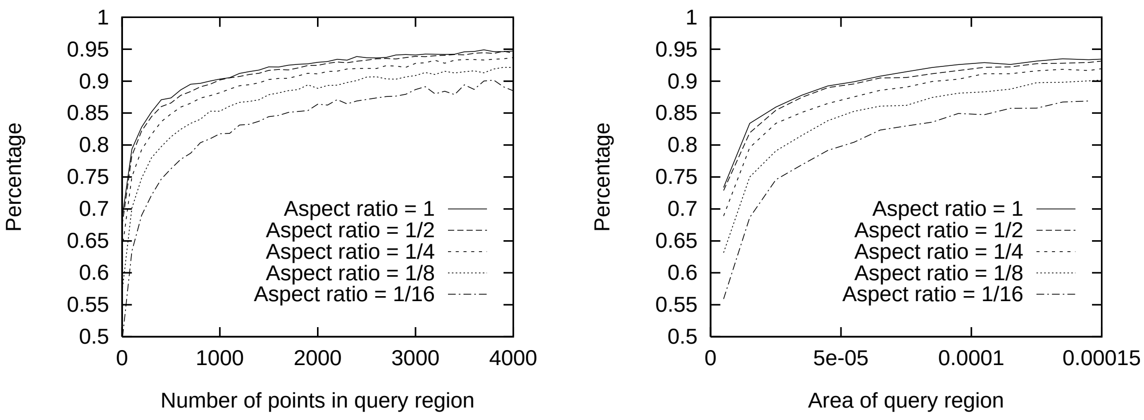

Figure 4.10: Number of points in the query region and size of the query region versus percentage of useful points and blocks for various aspect ratios.

When the graphs in Figure 4.9 are compared 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.

The graphs in Figure 4.10 are to illustrate the effect of different aspect ratios on the performance measures for non-uniform distribution. The results for uniform distribution are 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 1/16, more than 60% of the blocks accessed are useful to answer queries with 100 points. The largest decrease in the performance is about 15% from

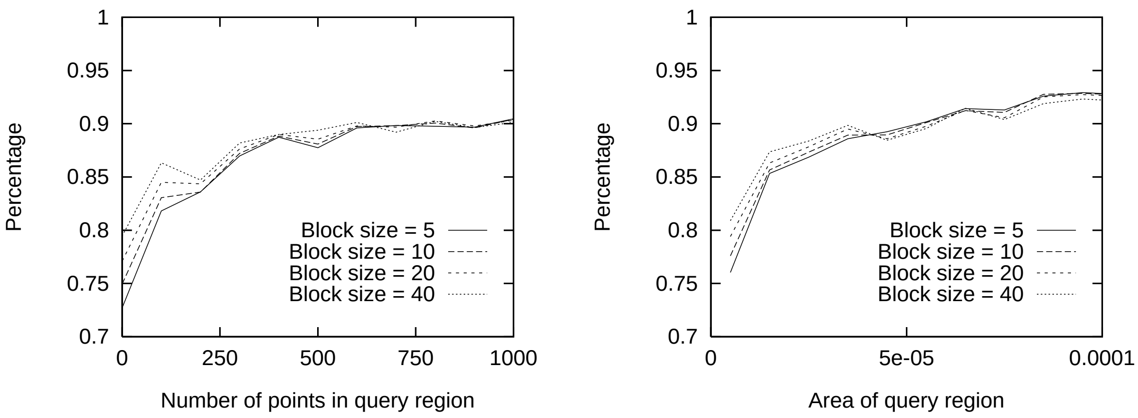

Figure 4.11: Number of points in the query region and size of the query region versus percentage of useful blocks and blocks for various block sizes (number of points in a block).

an aspect ratio of 1 to an aspect ratio of 1/16.

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, 20 and 40 for non-uniformly distributed points are shown in Figure 4.11. Similar results are also observed for the uniform distribution. For smaller queries with respect to both number of points enclosed in query region and query region size, percentage of useful blocks is higher for larger block sizes. However, the difference rapidly decreases while the size of the query increases. The curves becomes difficult to distinguish around 500 points and area of 0.00005. These results can not be converted to actual running times since the performance measures are not the only ingredient of the actual running time while the block size is changed.

4.3.2 Data Decomposition and Load Balancing

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 [74, 79, 106]. This section contains a distribution-independent domain decomposition algorithm using space-filling curves along with experimental results on the IBM SP-2.

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 [74]. 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.

To effect parallel domain decomposition on p processors, the following strategy is used: 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 aforementioned comparison routines. 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. An array of length p can be used to store the coordinates of the first point assigned to each processor, i.e., processor boundaries. By a simple binary search on this array 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.

The space-filling curve based partitioning algorithm described above was implemented on an IBM SP-2. The implementation was in C with calls to the MPI message-passing library functions. Parallel sample sort algorithm was employed because of its practical efficiency and scalability [61]. The Z-curve was chosen for the implementation. The algorithm was tested with three different graph sizes and two types of distributions for each size: uniform and non-uniform. The uniform distributions were

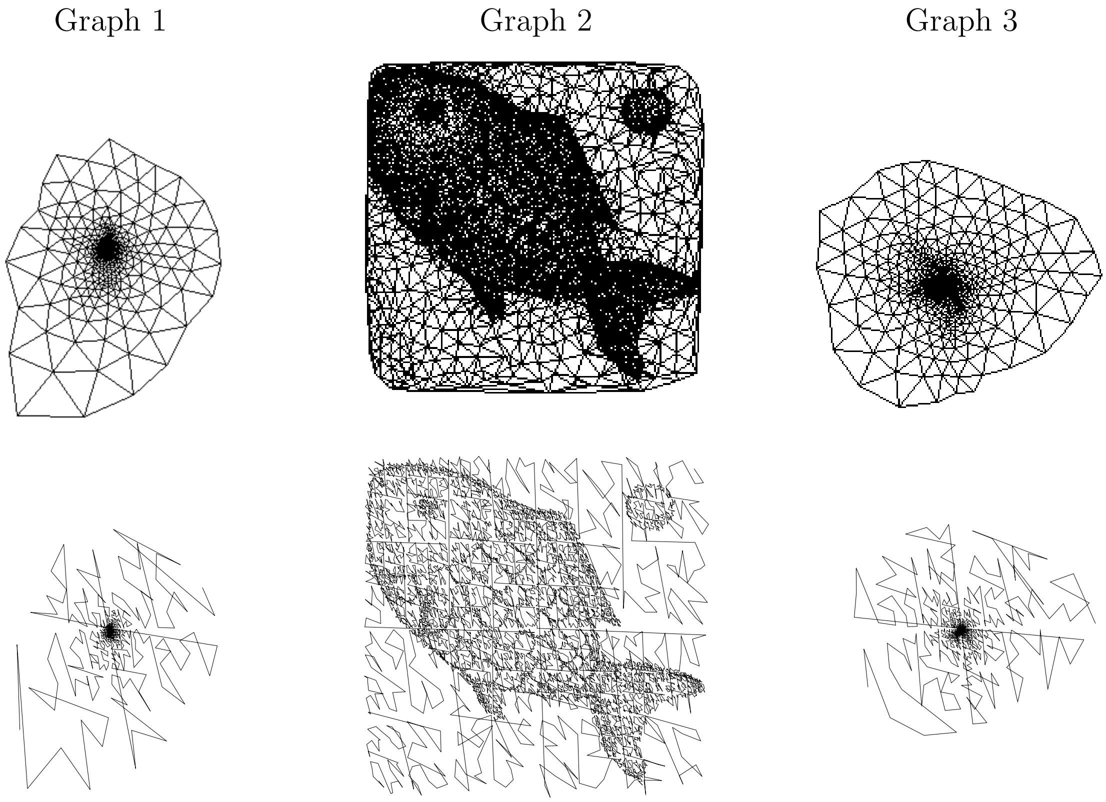

Figure 4.12: Non-uniform computational graphs and their ordering according to the Z-curve. Graph 1 has 6019 nodes, Graph 2 has 10166 nodes and Graph 3 has 15606 nodes. The graphs are taken from [74].

Number of Nodes	Graph size (# of vertices)					
	6019		10166		15606	
	non-uniform	uniform	non-uniform	uniform	non-uniform	uniform
1	0.0779	0.0755	0.1278	0.1270	0.2030	0.1962
2	0.0426	0.0441	0.0717	0.0721	0.1131	0.1056
4	0.0311	0.0223	0.0421	0.0426	0.0613	0.0591
8	0.0121	0.0107	0.0172	0.0180	0.0248	0.0249
16	0.0059	0.0062	0.0085	0.0098	0.0111	0.0132
32	0.0093	0.0084	0.0077	0.0103	0.0088	0.0123

Table 4.1: Parallel partitioning time (in seconds) for uniform and non-uniform graphs of different sizes.

generated using a uniform random number generator. The graphs chosen for the non-uniform distributions were taken from Ou's dissertation [74] and shown in Figure 4.12. The run times of parallel domain decomposition for different numbers of processors are shown in Table 4.1. 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 the algorithm is independent of the distribution of the data. Is is necessary to note the following three observation on the running times: Firstly, because of the overhead of parallel sort algorithm, the speedup is low from one to two processors. Secondly, The sizes of the graphs are small for large processor pools so, the gain of using more processors diminishes after 32 processors. Thirdly, there is a super-linear speedup because in same cases, the sample selected may not accurately reflect the characteristics of the data.

4.3.3 Incremental Load Balancing

Many scientific simulations require 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.

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, a technique similar to the one described in the previous subsection can be employed; 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, the proposed comparison routines is used. 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. Merge the vertices locally within the processors to achieve the global order. Once the multidimensional graph is sorted again, 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 5

Distribution Independent Adaptive Trees

Hierarchical tree data structures are a class of data structures that 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 recursively partition it 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

[90]. 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 [18].

Data driven tree 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. This chapter contains a detailed discussion of a new data structure based on hyperoctrees, Distribution Independent Adaptive Trees (DIAT). The DIAT tree data structure combines the advantages of both data driven and domain driven data structures.

5.1 Introduction

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

A hierarchy of cells can be built on a cell containing given n multidimensional points. A cell with the smallest possible length is chosen as the root cell of the hierarchy. This hierarchy is represented by a tree data structure called hyperoctree (quadtree in two-dimensions or octree in three-dimensions). Each node in the hyperoctree corresponds to a cell which encloses at least one point. The empty cells are not represented. The root node represents the root cell and the immediate subcells of the cell corresponding to a node are represented by its children. The quadtree for a cell hierarchy is shown in Figure 5.1.

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 hyperoctree, 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 points the cell contains. Compressed octrees are obtained by compressing each such path into a single node. Note that, each node in a compressed octree is either a leaf or has at least two children (see Figure 5.1 for a two-dimensional illustration). Therefore, the size of a compressed hyperoctree for n points is $O(n)$ and they are distribution-independent. However, straightforward algorithms for searching, insertion and deletion in compressed hyperoctrees are not

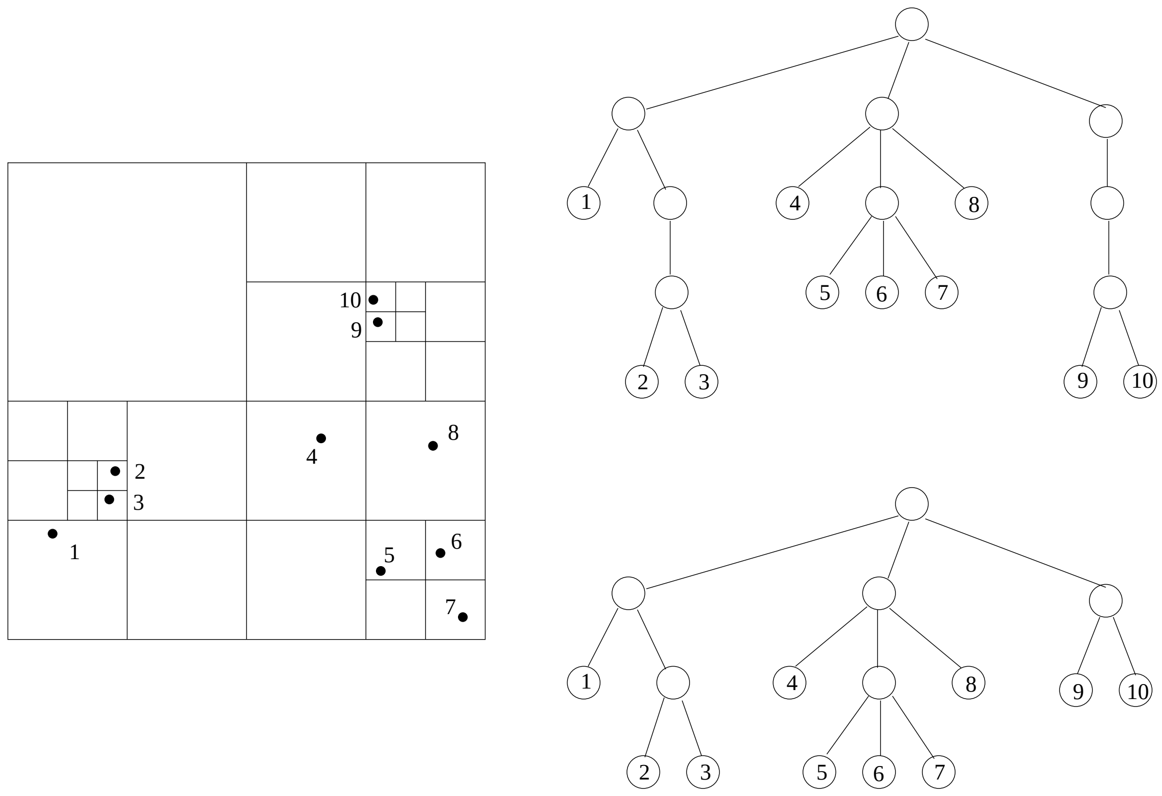

Figure 5.1: The hyperoctree and the compressed hyperoctree for 10 points in a two dimensional domain. The positions of the points in the domain are shown on the left. The square regions represents the subdivision of the domain in the construction of the hyperoctree. The resulting hyperoctree is shown at the top right. The corresponding compressed hyperoctree is at the bottom right.

efficient and require $\Omega(n)$ time in the worst case. This is because the height of the tree depends on the distribution of the points and is $\Omega(n)$ for some distributions.

5.2 The DIAT Tree

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

Each node v in the DIAT tree stores 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, 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.

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 $\Omega(n)$ in the worst-case. For speeding accesses and for efficient maintenance in the dynamic case, an ordering is imposed on the nodes of the DIAT tree and an auxiliary data structure is build on this ordering.

Let f be any bijective function that maps the 2^d immediate subcells of a cell to the set $\{1, 2, \ldots, 2^d\}$. Consider an internal node v and let v_1 and v_2 be two children of v. Then, $L(v_1)$ and $L(v_2)$ are immediate subcells of $S(v)$. Define an ordering as follows: v_1 appears earlier in the ordering than v_2 if and only if $f(L(v_1)) < f(L(v_2))$. If v_1 appears earlier in the ordering than v_2, then every node in the subtree of v_1 appears earlier in the ordering than any node in the subtree of v_2. The ordering is well-defined for any two nodes that do not have an ancestor-descendent relationship. If there is an ancestor-descendent relationship, the ancestor comes after the descendant. The ordering can be extended to sets of nodes and is a complete order. In particular, the ordering of all the nodes in this scheme is the same as the order of the nodes in postorder traversal of the tree. Furthermore, the leaves of a DIAT tree can be ordered. As each leaf contains a single point, we can also talk about ordering points according to the DIAT tree. Let p_1 and p_2 are two points represented by the leaf nodes u_1 and u_2 in the DIAT tree, p_1 is after p_2 in the DIAT tree order if $f(L(u_1)) > f(L(u_2))$. If the function f orders the immediate subcells of a cell in the same order as the Z-curve then the order of the points according to the DIAT tree and the order of the points according to Z-curve are equivalent.

The auxiliary data structure is a balanced binary search tree (BBST) built on the nodes of the DIAT tree. The BBST is used to access the nodes of the DIAT tree without navigating starting from the root. Each node in the BBST contains a pointer to the corresponding node in the DIAT tree. The order of the nodes in the DIAT tree determines the order of the corresponding nodes in the BBST.

5.3 Storing DIAT Trees

Unlike hyperoctrees, DIAT trees can be implemented using arrays because the number of nodes in a DIAT tree is bounded by $2n-1$ for n points. Each element in the array represents a node in the DIAT tree and the BBST. For internal nodes of the DIAT tree, only the small cell information, i.e., the length and the coordinates of a corner of the cell, is stored. The large cell information can be obtained by subdividing the small cell of the parent node.

For static points, the BBST can also be implicitly represented by using its inorder traversal, which corresponds to the sorted order. In the dynamic case, implementation 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, an element from the list is drawn and in the case of deletion, the deleted node is returned back to the list.

5.4 Constructing the Compressed Hyperoctree

The DIAT tree consists of two data structures: the compressed hyperoctree and the balanced binary search tree (BBST). In this section, three algorithms to build the compressed hyperoctree in $O(dn \log n)$ time are discussed. 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. The next section explains creation of the BBST for efficient searches and dynamic operations.

To make the discussion complete, the smallest cell technique can be used for a recursive construction of the compressed hyperoctree 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 $O(dn^2)$.

5.4.1 Divide-and-Conquer Algorithm

Let T_1 and T_2 be two compressed hyperoctrees 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 compressed hyperoctree T for $S_1 \cup S_2$ can be constructed in $O(d(|S_1| + |S_2|))$ time by merging T_1 and T_2.

To merge T_1 and T_2, start at their roots and merge the two trees recursively. Suppose that at some stage during the execution of the algorithm, a node u in T_1

Algorithm 4 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 5.2: Algorithm to merge two compressed hyperoctrees 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 $L(u) \cap L(v) \supseteq S(u) \cup S(v)$. Subscripts are used to represent the children of a node.

and a node v in T_2 are considered 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 points 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, i.e., if a node has less than 2^d children, there are empty nodes in place of the absent children. Four distinct cases arise as shown in Figure 5.2. Cases I and III admit symmetric cases with the roles of u and v interchanged and can be handled similarly.

- Case I: If there are no points 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 5.3), the corresponding children of u and v have the same large cells which are the immediate subcells of $S(u)$ (or equivalently, $S(v)$). In this case, simply merge these children one by one.

- Case III: If the small cell of one node strictly contains the other, one of the immediate subcells of the small cell is equal to or contains the other (See Figure 5.4). Assume that $S(u)$ contains $S(v)$. The points in v are also in the large cell of one of the children of u. Thus, this child of u is merged with v.

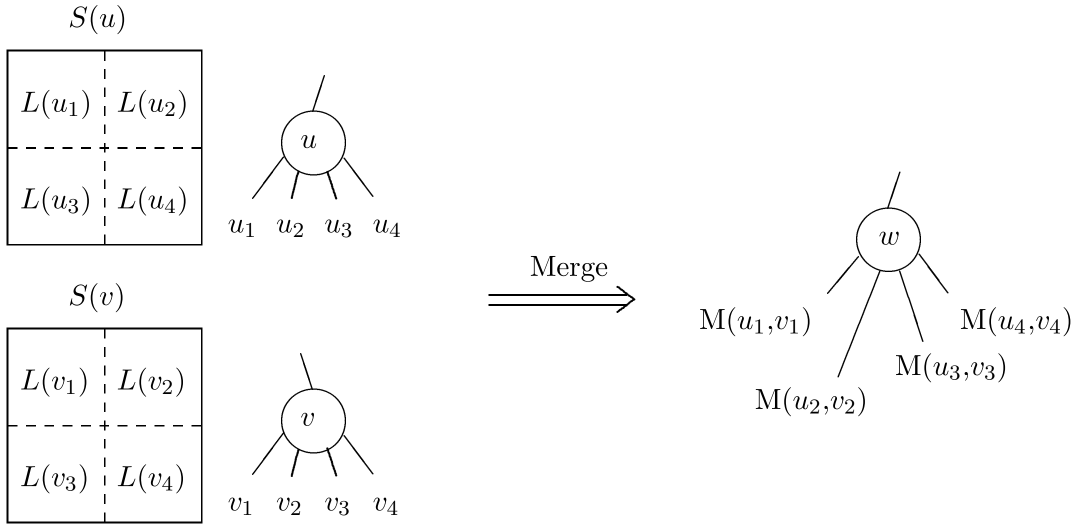

Figure 5.3: Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are equal. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The resulting tree is shown on the right. The node $\mathrm{M}(u_i,v_i)$ is the result of merging u_i and v_i.

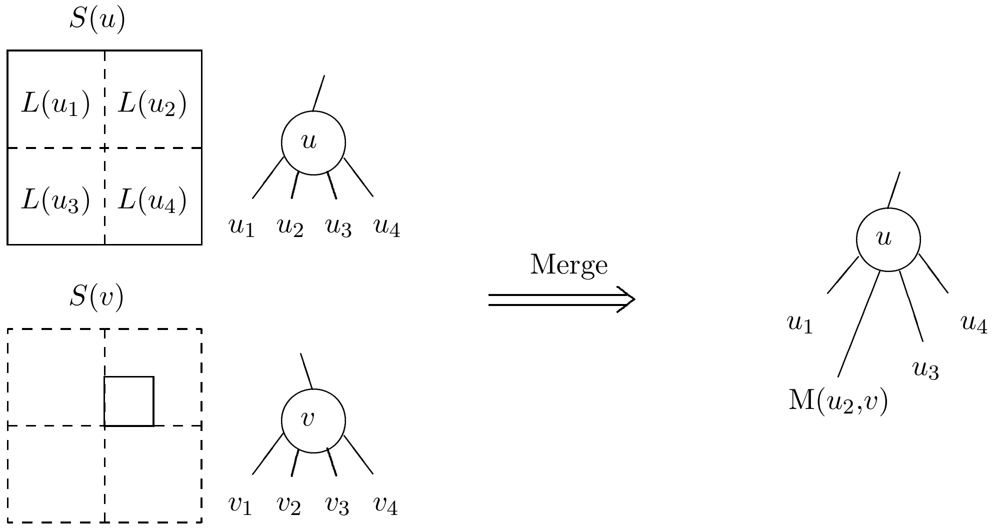

Figure 5.4: Merging subtrees rooted at u and v when $S(u)$ contains $S(v)$. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right. The node M(u_i,v_i) is the result of merging u_i and v_i.

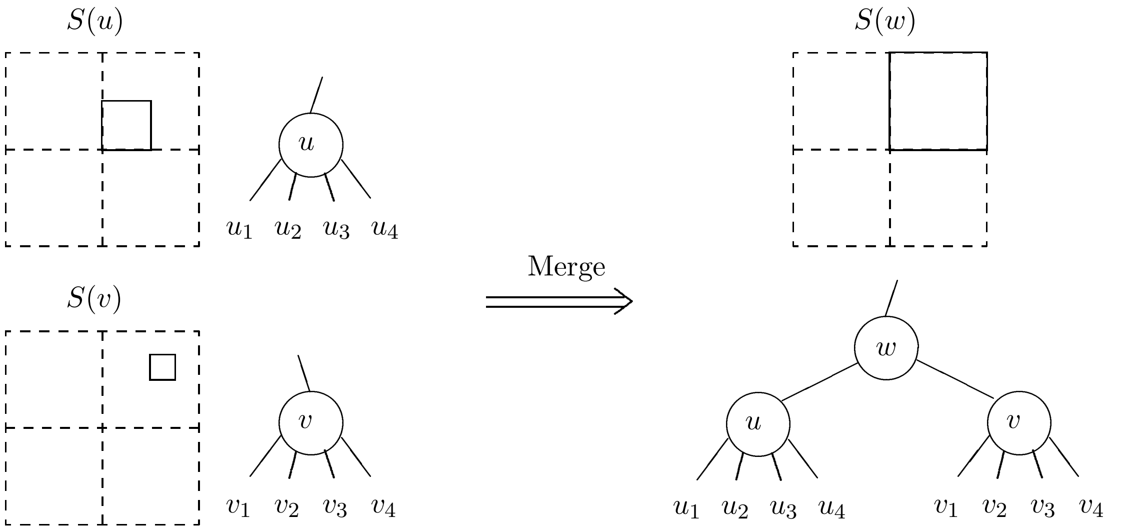

Figure 5.5: Merging subtrees rooted at u and v when the small cells $S(u)$ and $S(v)$ are disjoint. The subtrees rooted at u and v and their corresponding small cells are shown on the left. The small cells are shown using solid lines. The resulting tree is shown on the right.

- 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 5.5). If they both were to lie in the same immediate subcell, this immediate subcell, which is smaller, would have contained them, a contradiction. In this case, 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 4 *Two compressed hyperoctrees having sizes n_1 and n_2 with a common root cell can be merged in time $O(d(n_1 + n_2))$.*

Proof: The merging algorithm presented above performs a preorder traversal of each compressed hyperoctree. 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, at least one node is eliminated on one of the trees. Thus, the running time is proportional to the sum of the sizes of the trees to be merged. Since at each step takes only $O(d)$ time, total run-time is $O(d(n_1 + n_2))$. ■

To construct a compressed hyperoctree 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 compressed hyperoctree constructed in the process. Recursively construct compressed hyperoctrees for $\lfloor \frac{n}{2} \rfloor$ points and the remaining $\lceil \frac{n}{2} \rceil$ points and merge them in $O(dn)$ time. The compressed hyperoctree for a single point is a single node v with the root cell as $L(v)$. The algorithm is similar to mergesort, and takes $O(dn \log n)$ time.

5.4.2 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. The following procedure orders any two points: Given two points p_1 and p_2, compute the smallest subcell containing them. Subdividing this smallest subcell into its immediate subcells separates p_1 and p_2. The ordering of the two points is the same as the ordering of the immediate subcells they belong to according to the function f. Essentially, we are computing the lowest common ancestor of the leaf nodes representing p_1 and p_2 and ordering the large cells of two children of the lowest common ancestor containing p_1 and p_1. Given n points and a root cell containing them, the points can be sorted in $O(dn \log n)$ time. This can be done by using any optimal sorting algorithm.

The compressed hyperoctree is 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 5.6. Insertion process starts from 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 the first node v such that $p \in L(v)$ is found. Two possibilities arise (See Figure 5.7):

Algorithm 5 Bottom-up Construct (points)

Find the root cell
Sort the points according to the DIAT tree
Create one node compressed hyperoctree 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 5.6: Algorithm for the bottom-up construction of the compressed hyperoctree.

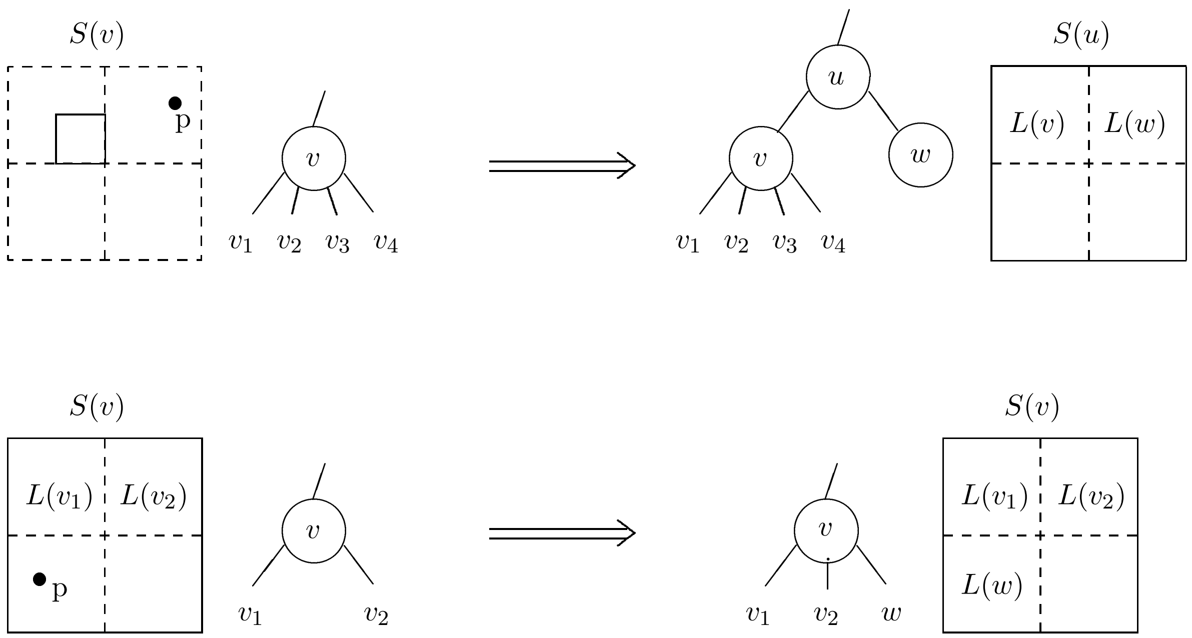

Figure 5.7: Illustration of the two cases of bottom-up compressed hyperoctree construction. The figures on the left shows the compressed hyperoctrees and the domains before inserting p and the resulting trees are on the right.

- 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, create a new node u between v and its parent and insert p as a child of u. $L(u)$ and $S(u)$ are set to the cell $L(v)$ and the smallest cell containing p and $S(v)$, respectively. The large cells of v and the node representing p are corresponding immediate subcells of $S(u)$.

- Case II: Otherwise, v is not a leaf node. Consider the current children of v. 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 compressed hyperoctree 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 compressed hyperoctree with $O(d)$ work per node. The time for a single insertion is not bounded by a constant but all n insertions together require only $O(dn)$ time. Combined with the initial sorting of the points, the tree can be constructed in $O(dn \log n)$ time.

5.4.3 Construction by Repeated Insertions

An algorithm to insert a point in an n-point DIAT tree in $O(d \log n)$ time is presented later. Using this algorithm, a compressed hyperoctree 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 $O(dn \log n)$ time.

5.5 Creating the Binary Search Tree

After constructing the compressed hyperoctree, it remains to build the BBST. Given the compressed hyperoctree, the balanced binary search tree can be constructed in $O(n)$ time. First, a postorder traversal of the compressed hyperoctree is performed to obtain the order of the nodes in $O(n)$ time. Then the BBST construction on this sorted data takes additional $O(n)$ time.

Theorem 1 *A DIAT tree for n points can be constructed in $O(dn \log n)$ time.*

5.6 Querying DIAT Trees

In this section, two types of searches on the DIAT trees are considered: point search and cell search. The result of a point search is the leaf node corresponding to the point if the point exists. Point search is not the main goal of DIAT trees. However, for cell searches and update operations, point searches are needed. 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 point can be considered to be a zero length cell. The node in the DIAT tree representing the given cell is located using the BBST by traversing 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 $O(d \log n)$.

5.7 Dynamic DIAT Trees

As points can be treated as cells of zero length, it is sufficient to describe insertions and deletions for cells only. Note that, although point insertions and deletions are always well-defined, there are complications on cell insertions and deletions. A cell insertion is meaningful only if the cell is inserted as a leaf node. Similarly, a cell deletion is meaningful only 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.

5.7.1 Insertions

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. To insert a given cell C, create a new node v that represent C and insert v in the BBST and the DIAT tree (i.e., set up the BBST and DIAT tree pointers). The algorithm is more formally described in Figure 5.8.

To insert v in the BBST, the standard insertion algorithm is used. 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 $O(d \log n)$ time.

Algorithm 6 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 successor.

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 5.8: Algorithm to insert the cell C in the DIAT tree.

5.7.2 Deletions

Before the deletion is actually performed, the first step is to check if C exists and can be deleted (i.e., whether it is represented by a leaf node in the DIAT tree). The cell C is searched in the DIAT tree for this purpose. If C is not represented in the DIAT tree, there is nothing to do. If the cell is not represented by a leaf node, the deletion is unsuccessful. Otherwise, as in insertion, the cell should be deleted from the BBST and 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.

.

Chapter 6

Neighbor Queries

Neighborhood queries are a class of queries where the neighborhood information for each point is gathered. Although the running time of performing the query on an individual point may be large, the queries for all the points can be answered more efficiently by intelligently combining the work required for individual queries. In this chapter, the following neighbor queries are considered:

1. **All-nearest neighbor queries** – Given n points, find the point that is closest to each point.

2. **All-k-Nearest neighbor queries** – Given n points, find the $k > 1$ points that are closest to each point.

3. **Spherical region queries** – Given n points, find all the points that lie within a distance of $r > 0$ from each point.

We present a unifying algorithm for all three types of queries using DIAT trees. The algorithm can be made suitable for various queries by changing there simple functions used in the algorithm. Because of this reason, although the algorithm is unique the running time is different for each type of query. Rigorous run-time analysis is presented for the first two types of queries for which the algorithm can be proved to be distribution-independent.

6.1 The Basic Algorithm

For all the queries we consider in this class, the algorithm presented in Figure 6.1 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 points in $S(v)$ or $S(u)$ is out of the result of the query for all points 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 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:

Algorithm 7 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 6.1: Unified algorithm for neighbor queries.

PROCESS	–	If $S(u)$ is in the result of the query for all points in $S(v)$
DISCARD	–	If $S(u)$ is not in the result of the query for all points 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 point 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.

6.2 All Nearest Neighbors

The invariant we would like to maintain for all-nearest neighbor queries is that nearest neighbor of each point in a node v is contained in some node in the active set of v. Initially, the root node has itself in its active set, so, the invariant is satisfied. For each node v, the algorithm keeps track of a value $d_{minmax}(v)$ which is the minimum of largest distances $d_{max}(v, w)$ between $S(v)$ and $S(w)$ for all w in active set of v. Note that, if there is more than one point in a node, the nearest neighbor of a point

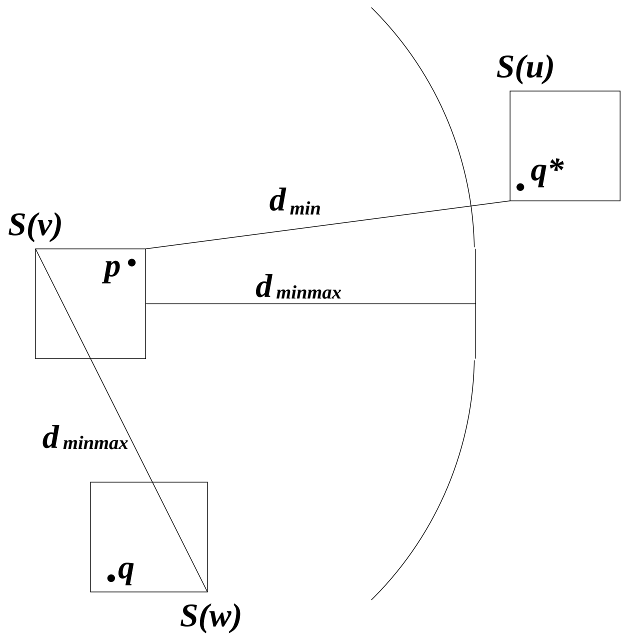

Figure 6.2: Illustration of when the function *Status* returns the value DISCARD for all-nearest neighbors queries.

in the node can be in the same node. Thus, in this case, the node is in the active set of itself. The largest distance from a cell to itself is the length of a main diagonal of the cell.

If the smallest distance $d_{min}(v, u)$ between $S(v)$ and $S(u)$ is greater than $d_{minmax}(v)$, 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.

Figure 6.2 and 6.3 shows the DISCARD and UNKNOWN cases respectively. In the figures, $d_{minmax}(v)$ is obtained by using the node w. Since there is at least one

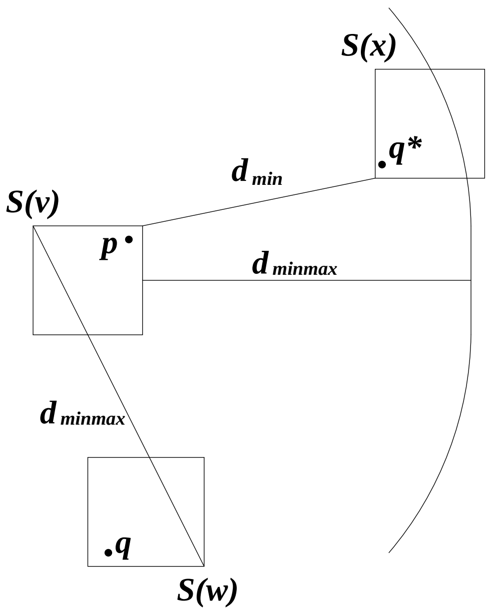

Figure 6.3: Illustration of when the function *Status* returns the value UNKNOWN for all-nearest neighbors queries.

point in $S(w)$ which is closer to the point p than $d_{minmax}(v)$, the distance from p to its nearest neighbor cannot be larger than $d_{minmax}(v)$. However, the point q^* in $S(u)$ cannot be the nearest neighbor of p because $d_{min}(v, u)$ is larger than $d_{minmax}(v)$, which means every point in $S(u)$ is farther from p than any point in $S(w)$. Thus, the node u can be discarded.

On the other hand, in Figure 6.3, some points represented by the node x can be the nearest neighbor of p, since $d_{min}(v, x)$ is smaller than $d_{min}(v, u)$. Actually, in the figure, the point q^* in $S(x)$ is closer to p than q. In this case, the information about the nodes is not enough for a final decision.

To be able to prove the running time of $O(n \log n)$, the nodes in the active set 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 having the largest small cell in the active set, breaking ties arbitrarily.

The computation performed by the algorithm can be related to Vaidya's algorithm [105], 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 active set 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 active set. On the other hand, the size of the active set can grow as large as $\Theta(n)$ for the leaf nodes. By using a heap for the small cells of the nodes in the active set, both the function Select and insertions can be accomplished in $O(\log n)$ time. Vaidya proves that any node w can be considered by only a constant number of nodes [105]. 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 all-nearest neighbor queries is $O(n \log n)$.

6.3 All k-Nearest Neighbors

For computing the k-nearest neighbors of each point, some modifications to the all-nearest neighbors algorithm are necessary. As in the all-nearest neighbors, for each node w in the active set of v, the algorithm keeps track of the largest distance $d_{max}(v,w)$ between $S(v)$ and $S(w)$. Assume the nodes in the active set of v are sorted according to corresponding largest distances and let w_i be the node in the active set, having the i^{th} smallest distance. Assume each node contains the information about the number of points it contains. If this information is not available, it can be computed by a simple, $O(n)$ time, bottom-up traversal of the tree. Let j be the smallest integer where the total number of points in the nodes $w_1, w_2, \ldots, w_j$ is greater than k. Note that, the distance between the points in the nodes $w_1, w_2, \ldots, w_j$ and any point in node v is less than or equal to $d_j = d_{max}(v, w_j)$. If the smallest distance between a point and the cell $S(v)$ is grater than d_j, the point cannot be a k-nearest neighbor of any point in v. Therefore, the function *Status(v,u)* returns DISCARD if the smallest distance between $S(v)$ and $S(u)$ is greater than d_j. The function *Select* still picks the largest cell in the active set.

In Figure 6.4, the smallest distance between $S(u)$ and $S(v)$ is greater than d_j. Since, for any point p in v, there are more than k points having distance to p smaller than d_j and the distance between p and any point q in u is larger than d_j, the points in u cannot contribute to the k-nearest neighbors of a point in v. Thus, the node u can be discarded. However, in Figure 6.5, the distance between the point q in w and

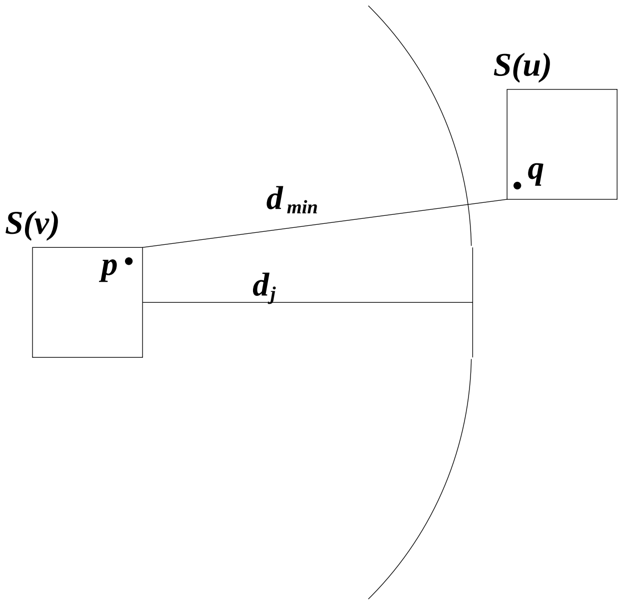

Figure 6.4: Illustration of when the function *Status* returns the value DISCARD for all-k-nearest neighbors queries.

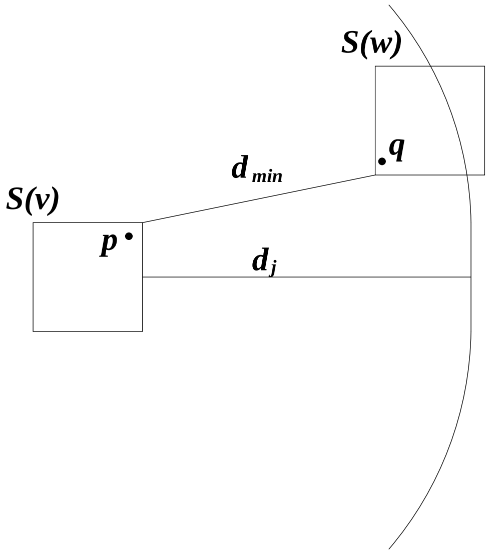

Figure 6.5: Illustration of when the function *Status* returns the value UNKNOWN for all-k-nearest neighbors queries.

p is smaller than d_j. In this case, q can be closer to p than some of the points in $w_1, w_2, \ldots, w_j$ such that the number of points closer to p than q is less than k and q is a k-nearest neighbor of p.

The running time of the algorithm is $O(kn \log n)$. To prove this bound, consider a slightly modified 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 active set 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 the active set. 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 $O(kn \log n)$.

6.4 Spherical Region Queries

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

For a node v, a node w may contain points in the spherical region corresponding to some point in v only if the smallest distance between $S(v)$ and $S(w)$ is less than r. If the largest distance between $S(v)$ and $S(w)$ is less than r, then all points in w are in the spherical region of every point in v. 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. In Figure 6.6, the return value of *Status(v,w)*, *Status(v,u)* and *Status(v,x)* are PROCESS, UNKNOWN and DISCARD, respectively. Processing u means including all the points in u in the query result for each point in v. For this query, we do not need any special selection.

The run-time analysis of the algorithm for spherical region queries is an open problem. Note that the size of the query result depends on the distribution of points and the running time should be a function of the size.

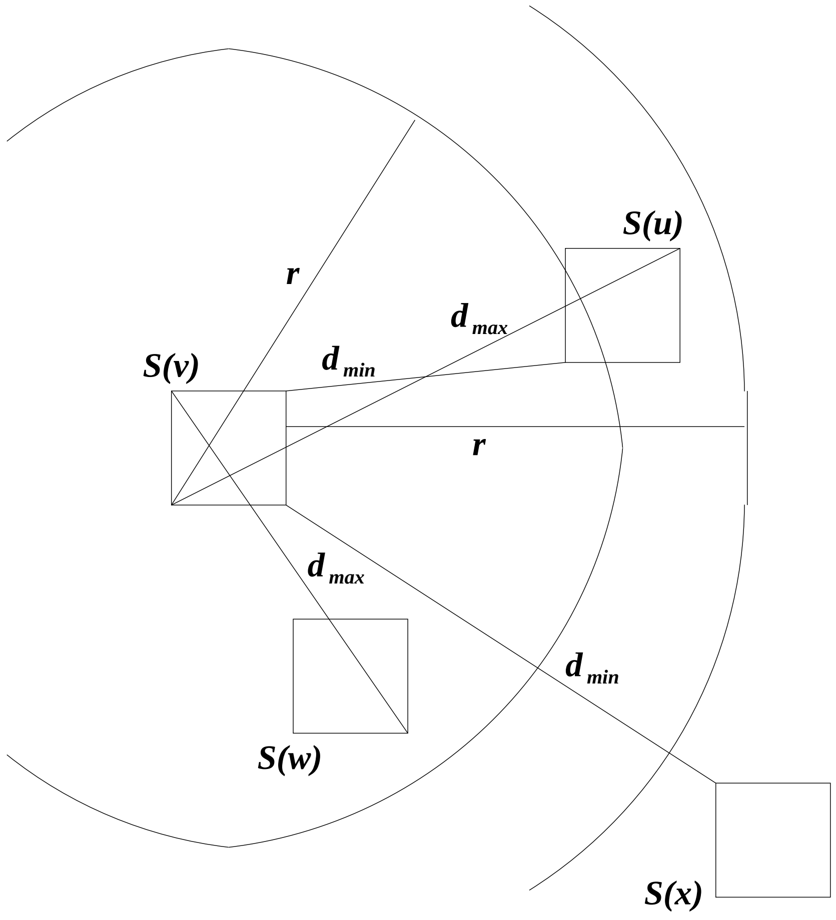

Figure 6.6: Illustration of various return values of function *Status* for spherical region queries. The return values are PROCESS, UNKNOWN and DISCARD for nodes w, u and x, respectively.

Chapter 7

Fast Multipole Method

The Fast Multipole Method (FMM) is a robust technique for the simulation of the evolution of n objects under the influence of pairwise interactions. A direct algorithm for computing all pairwise interactions requires $O(n^2)$ time. This complexity is reduced by approximating the interaction between clusters of objects instead of computing individual interactions. The FMM is popular because it allows the estimation of rigorous error bounds and the computation cost grows slowly with the problem size.

The most popular application of FMM is to physical system simulations involving particles. 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. In this chapter, the FMM is presented in the context of 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 the FMM. 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.

7.1 Introduction

The essential idea of the FMM is to approximate interactions between distant clusters of particles instead of computing individual pairwise interactions. To enable a recursive application of this idea, the method uses a hierarchical subdivision of the space containing the particles. The hierarchical subdivision is usually represented by an octree data structure. In practical FMM implementations, leaf cells are allowed to hold up to a constant number of particles, where the constant is determined by experimental evaluation.

In FMM, each cell in the octree is associated with a multipole expansion and a local expansion. The multipole expansion at a cell C, denoted $\phi(C)$, is a truncated infinite series that describes the effect of the particles within the cell C at a distant point. It converges at points that are sufficiently far off from cell C. The local expansion at a cell C, denoted $\psi(C)$, is a truncated infinite series that describes the effect of all distant particles on the points within the cell C.

The multipole expansions are computed by a bottom-up traversal of the octree. At the leaf cells, they are directly computed. For an internal cell, its multipole expansion is computed by aggregating the multipole expansions of its immediate subcells. The time for computing the multipole expansions is proportional to the number of nodes in the octree.

The local expansion at a cell C is obtained by appropriately combining (referred to as 'addition', for simplicity) the multipole expansions of the cells that converge at every point in cell C. In Figure 7.1, the local expansion at cell C should include the effect of all particles outside region R. However, the effect of all particles outside region R' are included in the local expansion of C's parent C'. When considering cell C, we only compute the effect of particles that are outside region R but within region R'. We call this the *partial local expansion* at cell C. This is added to the local expansion at cell C' to compute the local expansion at cell C.

To compute local expansions for all cells, a top-down traversal is used. Two cells C and D of equal length are said to be *well-separated* if C's multipole expansion converges at every point in D and vice versa. The local expansion at a cell is borrowed from its parent. However, some cells are not accounted for in the parent's local expansion because they are not well-separated from it. These cells are subdivided into their immediate subcells and the subcells are separated into two sets: the set of subcells that are well-separated from the cell and the set of subcells that are not well-separated. The first set is called as *interaction set* and the second set is called

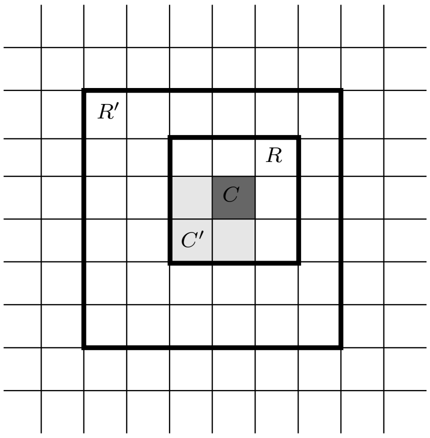

Figure 7.1: Illustration of proximity set and interaction set in two dimensions.

as the *proximity set*. The multipole expansions of the well-separated cells are added to the parent's local expansion. Finally, the total effect of all particles at a particle is obtained by evaluating the local expansion at its leaf. Because the size of any interaction set is bounded by a constant, computing the partial local expansion at each cell takes constant time. Thus, the time for computing all the local expansions is proportional to the size of the octree.

As it is mentioned in Chapter 5, the octree data structure is distribution-dependent. Thus, the FMM has unbounded running time for arbitrary distribution as there is no upper bound on the size of the tree. This problem is a result of chains on an octree such as $v_1, v_2, ..., v_k$ where v_i is the only child of v_{i-1} $(1 < i \leq k)$. Note that each node represents the same particles on the chain so, the multipole expansions at these nodes are merely translations of one another. They are useful only to compute the multipole expansion of v_1's parent which can be directly obtained by using the multipole expansion of v_k. In the next section, how this computation can be avoided to achieve distribution-independence is presented.

7.2 Fast Multipole Method Using DIAT Trees

To achieve a provably good run-time and distribution-independence, the DIAT tree data structure can be used in the FMM. In a DIAT tree, each chain $v_1, v_2, ..., v_{k-1}$ is replaced by the last node on the chain, i.e., v_k. Thus, the aforementioned problem in computing multipole expansions can be solved trivially. The multipole expansion of v_1's parent is computed using the multipole expansion of v_k.

On the other hand, in the algorithm presented in previous section, the local expansion of a node is not directly calculated from the local expansion of its parent. The nodes in the interaction set also contribute to the local expansion. While calculating the local expansions of the nodes $v_1, v_2, ..., v_k$ in a chain, 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-separated*(C, D) to be true if D's multipole expansion converges at any point in C, and false otherwise. If two cells are not well-separated, they are *proximate*. Similarly, two nodes v_1 and v_2 in the DIAT tree are said to be *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 *proximate*.

The DIAT tree based FMM 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 aggregating the multipole expansions of the children of v.

The algorithm to compute the local expansions is given in Figure 7.2. The computations are done using a top-down traversal of the tree. To compute local expansion at node v, the set of nodes that are proximate to its parent needs to be considered, which is called the *proximity set*, $P(parent(v))$. The proximity set of the root node contains only itself. Recursively decompose these nodes until each node is either 1) well-separated from v or 2) proximate to v and the length of the small cell of the node is smaller than the small cell of v. The nodes satisfying the first condition form the *interaction set* of v, $I(v)$ and the nodes satisfying the second condition are in the proximity set of v, $P(v)$. In the algorithm, the set $E(v)$ contains the nodes that are yet to be processed. Local expansions are computed by combining parent's local expansion and the multipole expansions of the nodes in $I(v)$.

Before analyzing the run-time of this algorithm, it is necessary to precisely define

---

Algorithm 8 Compute-Local-Exp (v)

I. Find the proximity set $P(v)$ and interaction set $I(v)$ for v

  $E(v) = P(parent(v))$

  $I(v) = \emptyset$; $P(v) = \emptyset$

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

    Pick some $u \in E(v)$

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

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

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

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

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

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

the sets used in the algorithm. The set of cells that are proximate to the cell C and having same length as C is called *the proximity set of* C and is defined by $P^{=}(C) = \{D \mid \mathit{length}(C) = \mathit{length}(D), \neg \mathit{well\text{-}separated}(C, D)\}$. The superscript "=" is used to indicate that cells of the same length are being considered. For a node v in the DIAT tree, define the *proximity set P(v)* as the set of all nodes proximate to v and having the small cell smaller and large cell larger than $S(v)$. More precisely, $P(v) = \{w \mid \neg \mathit{well\text{-}separated}(S(v), S(w)),\ \mathit{length}(S(w)) \leq \mathit{length}(S(v)) \leq \mathit{length}(L(w))\}$. The *interaction set I(v)* of v is defined as

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

where $\mathit{parent}(w)$ denotes the parent of the node w.

In the rest of the section, it is proved that the algorithm achieves the running-time of $O(n)$ for any predicate *well-separated* that satisfies the following there conditions:

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

C2. For any cell C, $|P^{=}(C)|$ 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 various applications using the FMM and in particular, the N-body problem. Well-separatedness criteria is used to decide if the approximate interaction on one cluster due to another should be directly computed. In these applications, this decision is solely based on the geometry of the cells and their relative distance and is oblivious to the number of particles or their distribution within the cells. Given two cells C and D of the same length, 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, 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 proximity set of its parent cell must be considered. It follows that, the size of the proximity sets of cells of the same length should be $O(1)$ as prescribed by Condition C2 in order that an $O(n)$ algorithm is possible. Otherwise, an input that requires processing the proximity sets of $O(n)$ such cells can be constructed, 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'.

Lemma 5 *For any node v in the DIAT tree, $|P(v)| = O(1)$.*

Proof: Consider any node v. Each $u \in P(v)$ can be associated with a unique cell $C \in P^{=}(S(v))$ such that $S(u) \subseteq C$. This is because any subcell of C which is not a subcell of $S(u)$ is not represented in the DIAT tree. It follows that $|P(v)| \leq |P^{=}(S(v))| = O(1)$ (by Condition C2). ■

Lemma 6 *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 $w \in P(parent(v))$ or w is in the subtree rooted by a node $u \in P(parent(v))$. Thus,

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

Consider these two summations separately. The bound for the first summation is easy; From Lemma 5, $|P(parent(v))| = O(1)$. So,

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

The second summation should be explored more carefully.

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

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

Since $w \notin P(parent(v))$, there exists a node $u \in P(parent(v))$ 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 $S(parent(w)) \in P^{=}(C)$ and $C \in P^{=}(S(parent(w)))$ by Condition C1. By Condition C2, we know that $|P^{=}(S(parent(w)))| = O(1)$. Moreover, for each cell $C \in P^{=}(S(parent(w)))$, there are at most 2^d 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, $\sum_v |I(v)| = \sum_w |\{v \mid w \in I(v),\ w \notin P(parent(v))\}| = \sum_w O(1) = O(n)$. ■

Theorem 2 *Given a DIAT tree for n particles, the FMM calculation can be completed in $O(n)$ time.*

Proof: Computing the multipole expansion at a node takes constant time and the number of nodes in the DIAT tree is $O(n)$. Thus, total time required for the multipole expansion calculation is $O(n)$. The nodes explored during the local expansion calculation at a node v are either in $P(v)$ or $I(v)$. In both cases, it takes constant time to process a node. By Lemma 5 and 6, the total size of both sets for all nodes in the DIAT tree is bounded by $O(n)$. Thus, local expansion calculation takes $O(n)$ time. As a conclusion, the running time of the fast multipole algorithm on the DIAT tree takes $O(n)$ time irrespective of the distribution of the particles. ■

7.3 Incremental Updates

As noted before, a popular application of the FMM is the N-body problem. In the N-body problem, the FMM is used to calculate the force on each particle due to every other particle. The force 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 force. The position of each particle after an arbitrary length of time is calculated by many iterations of this method. A single iteration of the N-body problem consists of two phases:

1. Constructing the DIAT tree
2. Computing the multipole and local expansions for each node of the tree.

The running time for the first phase is $O(n \log n)$ as proved in Chapter 5 and the second phase is proved to require $O(n)$ time. So, the running time of the overall algorithm is $O(n \log n)$. 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. 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 the N-body algorithm to $O(n + k \log k)$ with $O(n \log n)$ pre-processing time.

.

Chapter 8

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. To parallelize applications that use accumulations, the accumulation procedures should be parallelized as well. In this chapter, first, parallel accumulation algorithms on the PRAM model are described briefly. 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. Later, a more realistic model for parallel computers, distributed memory permutation network model, is defined and parallel accumulation algorithms on this model are presented.

8.1 Tree Accumulations on PRAM Model

This section contains description of two EREW PRAM algorithms for upward tree accumulation. The first algorithm is based on the tree contraction algorithm by Kosaraju and Delcher [60]. The second algorithm is by Miller and Reif [65]. Both algorithms consist of two phases, *contraction* and *expansion*. The downward accumulation counterparts of both algorithms are presented by explaining the differences between upward and downward accumulations.

The first algorithm assumes 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.

In the second algorithm, a tree with constant degree is assumed. 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 [43], Gazit et. al. propose some techniques to reduce the memory requirement for each node to a constant.

8.2 Permutation Network Model

In this section, the parallel model of computation assumed while designing the algorithms in the subsequent chapters is presented.

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, the model of communication cost in the presence of multiple messages is 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, the run-time analysis for only the distributed-memory model is presented in this thesis.

8.3 Primitive Parallel Operations

Although the permutation network model is assumed throughout this thesis, it should be stressed that the algorithms presented are applicable to other models of computation as well, and often equally efficiently. This is because the algorithms are presented in terms of a few simple communication primitives that can be implemented in efficiently on many architectures, for example hypercubes and multistage interconnection networks. Also, under the assumption that the distance between communicating processors (in the absence of network contention) can be ignored due to the large set-up costs and the moderate size of parallel computers, the same efficiency could be achieved for meshes as well. In the following, these primitive operations are described. As these operations are well known, they are briefly described and their run-times are stated, omitting the run-time analysis. For a detailed description and run-time analysis, see [61]. 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 $\Sigma_{i=1}^{m} n_i = n$. The segmented prefix sum is $s_{i,j}$, 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 $\frac{t}{p}$ [82].

- **Sorting:** Sorting n elements takes $O\left(\frac{n \log n}{p}\tau p + \mu\left(\frac{n}{p} + p\log^2 p\right)\right)$ time (for $n > p^2 \log p$) by using sample sort [61] in conjunction with bitonic sort for sorting splitters identified during sample sort.

8.4 Distributed Memory Tree Accumulations

For the tree accumulation algorithm presented in this section, the following is assumed: The accumulation function is associative. The tree has a constant degree and the nodes of the tree are stored in an array T according to its postorder traversal. The array T 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.

If the tree is not available readily on an array, the following operations can be used to create the array: 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 [103] 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 [58].

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 all the repetitions except the last occurrence are removed and the remaining nodes are numbered, the postorder traversal of the tree is obtained.

Consider the list of edges in the Euler tour starting from and ending by the root node. Assign 0 to each edge from a node to one of its children and 1 to each edge from a child to its parent. The postorder number of a node is according to the prefix sum for the directed edge from the node to its parent. The parallel prefix operation required to calculate these prefix sums is different from the one discussed in previous section because the calculation should be performed on a list not on an array. The methods described in [98, 99] for the list ranking problem can be used for an efficient implementation of such a parallel prefix. Now, it is sufficient to move all the nodes to respective processors and adjust pointers to the parents. This sequence of operations uses $O(\log n)$ transportation primitives and the computation cost for it is $O\left(\frac{n}{p}\log n\right)$.

8.4.1 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 run-time 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 is $\Omega(n)$, such an algorithm takes $\Omega(n)$ 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. 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 8.2(B) illustrates an internal chain and a leaf chain. If the chains can be compressed before starting the next iteration, this will lead to an algorithm that runs in $O(\log n)$ iterations. In the following, it will be shown 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 the 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

Algorithm 9 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 8.1: Algorithm for parallel upward accumulations.

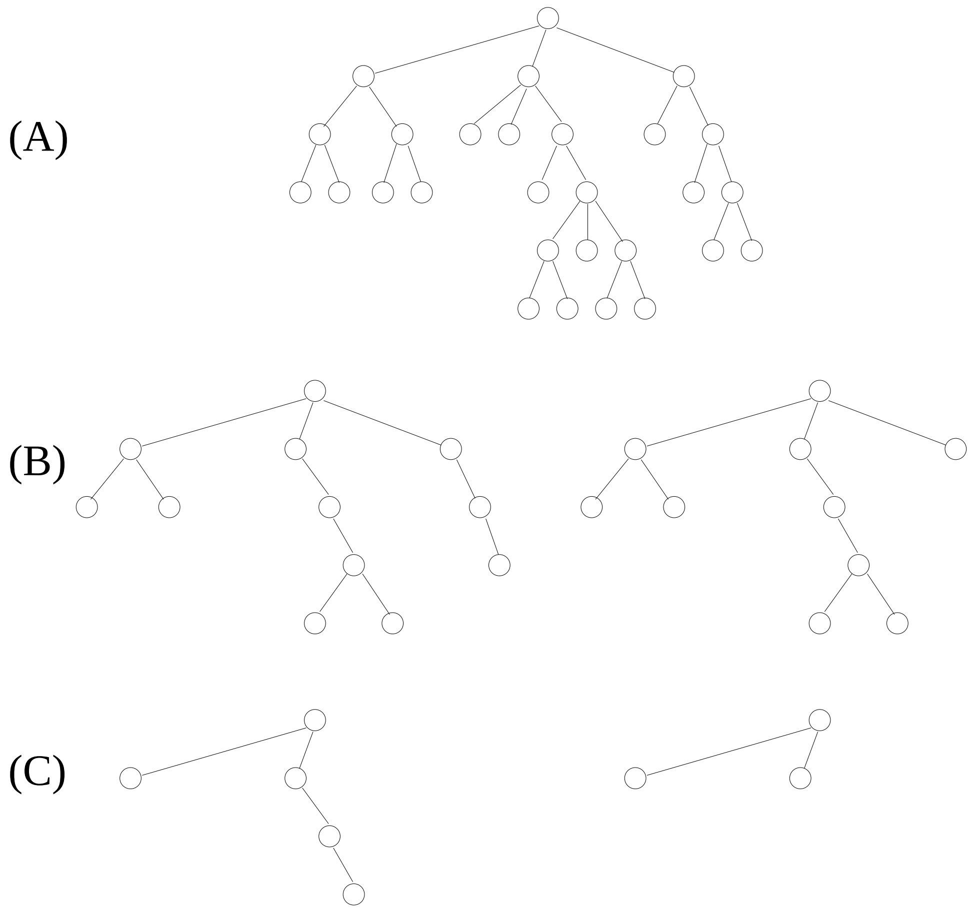

Figure 8.2: Execution of the parallel upward accumulation algorithm. The initial tree is depicted in (A). The first two iterations of the algorithm are shown in (B) and (C). The left tree is after step 1 and the right tree is after step 2 of the corresponding iteration.

a single node. An example of the execution of the algorithm is shown in Figure 8.2. Figure 8.2(A) illustrates the initial tree and Figures 8.2(B) and 8.2(C) illustrate the first two iterations of the algorithm.

Lemma 7 *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 the 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. To show this, use 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, such elements are skipped over. The upward accumulations of all the children of a node may not be added to the node in the same iteration. Thus, for each node, the number of children whose upward accumulations are yet to be added are maintained.

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 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 postorder traversal is used. The following variant of segmented parallel prefix is used 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, $O\left(\frac{n}{p}\right)$ work is performed on each processor for the purpose of identifying the leaves, setting up markers etc.. There are also 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 $n > p \log p$.

8.4.2 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, 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. This is achieved by collecting information during the execution of tree contraction which is used in computing upward accumulation so that it can exactly be reversed. 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: 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 upward accumulation computation itself is not needed for the contraction algorithm. So segmented parallel prefix is not performed while compressing leaf chains in step 2. Instead, the chains are just deleted. Downward accumulations are computed while activating the deleted nodes in the reverse order of contraction. Repeat the 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. A counter (say i) that starts with the number of iterations in the contraction phase is maintained and decremented 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.

Algorithm 10 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 8.3: Algorithm for calculation of downward accumulations.

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 $n > p \log p$.

Chapter 9

Parallel Fast Multipole Method

Parallel Fast Multipole Method is of interest because 1) the FMM 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. There are many algorithms developed for parallel implementations of the FMM by various research groups.

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 interaction computation phase. The data decomposition required for efficient tree construction may conflict with the data decomposition required for interaction computation. In many applications, many iterations of FMM computation is required and the data distribution changes from iteration to iteration. This 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. Because of all these, the efficiency of the algorithms is often demonstrated by experimental evaluation alone.

A parallel FMM algorithm that is distribution-independent and rigorously analyzable is presented in this chapter. The algorithm is a parallelization of the sequential framework introduced in Chapter 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 given in Chapter 8. For the simplicity of the presentation, the application of the FMM to the N-body problem is assumed.

9.1 Introduction

Any FMM algorithm should consist of the following three basic steps:

1. Constructing the hierarchical subdivision for the particles.
2. Calculating multipole expansions.
3. Calculating local expansions.

Once the local expansions at the leaf nodes are calculated, they are used to compute the combined effect of all other particles on each particle.

In the following, algorithms for three basic steps are presented. The local expansion calculation is subdivided into two steps for simplicity of presentation. Recall that, the local expansion at a node is calculated using the local expansion at the parent and multipole expansions at the nodes in the interaction set. Let us call the contribution of the nodes in the interaction set as *partial local expansion*. To calculate the local expansions, first, the partial local expansions are calculated. Then, the partial local expansions are accumulated from the root to the each node.

The data decomposition assumed by the algorithm is extremely simple: Initially, the n particles are distributed across the processors such that each processor has approximately the same number of particles. The algorithm employs the compressed octree data structure. Throughout the algorithm, the compressed octree is stored in an array according to its postorder traversal. This array is uniformly distributed across the processors. No load estimation or load balancing is used either within a single iteration, or in between multiple iterations. Instead, it is proved that each processor does the same amount of work during each step of the algorithm.

9.2 Parallel Construction of Compressed Octrees

The algorithm for constructing compressed octrees is based on Bern et. al.'s algorithm [19] for constructing quadtrees on the PRAM model.

Algorithm 11 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 9.1: Algorithm for parallel construction of compressed octree.

To construct the compressed octree, first, sort the particles in parallel based on their positions as in the sequential bottom-up construction algorithm. 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 lowest 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 lowest common ancestor of the rightmost leaf in v_i's subtree and the leftmost leaf in v_{i+1}'s subtree. If the lowest common ancestors of every consecutive pair of leaf nodes is generated, every internal node (i.e., the small cell of the internal node) in the compressed octree is guaranteed to be generated. However, there may be duplicates − each internal node is generated at least once but at most seven times.

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 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 chapter.

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, it is necessary to determine its parent cell, search for and locate it and set up the pointers. This is accomplished as follows: For each node, 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*. Virtual parent cells of all nodes can be generated in parallel. To this, 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 $n > p^2 \log p$.

9.3 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 the same 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 multipole expansion 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 $n > p \log p$.

9.4 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, a mechanism to independently generate the interaction sets is needed. 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 $O(n)$, the size of a single interaction set is only bounded by $O(n)$. 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.

An alternative way of computation is to reverse the perspective: Instead of computing the interaction set of each node v, 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 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 $O(n)$.

In the following, it is shown that by employing a combination of the two strategies, constant set sizes can be achieved. Recall that the interaction set of a node v is defined as

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

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

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

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

Note that the interaction set for a node v satisfies

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

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 the cells in $P^{=}(S(parent(v)))$ are searched in the compressed octree the nodes in $I_f(v)$ can be found and if the immediate subcells of $P^{=}(S(parent(v)))$ are searched the nodes in $I_b(v)$ can be found.

Lemma 8 *Let v and w be two nodes in a compressed octree. Then*

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

Proof: Let $w \in I_f(v)$. By definition, $\textit{well-sep}(v,w)$ and $w \in P(parent(v))$. By the definition of proximity set, $length(S(w)) \leq length(S(parent(v))) \leq length(L(w))$ and $\neg\textit{well-sep}(parent(v),w)$. Thus, $\exists C$ such that $length(S(parent(v))) = length(C)$ and $S(w) \subseteq C \subseteq L(w)$. Since $\neg\textit{well-sep}(parent(v),w)$ and $S(w) \subseteq C$, C and $S(parent(v))$ are not well-separated. Then, $C \in P^{=}(S(parent(v)))$.

Now, let $\textit{well-sep}(v,w)$, $\neg\textit{well-sep}(parent(v),w)$ 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 3 *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 9 *Let v and w be two nodes in a compressed octree. Then,*

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

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

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

Corollary 4 *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

Algorithm 12 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 9.2: Algorithm for parallel calculation of partial local expansions.

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 9.2. Consider the computation of forward interaction sets and the corresponding accumulation of multipole expansions. Lemma 3 is used 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. 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, perform searches in the following way: For each node v in T, put $S(v)$ and the cells in $P^{=}(S(parent(v)))$ together into an array T' and sort T' in parallel. Along with each $S(v)$, its multipole expansion is also carried. 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, 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. The multipole expansion at each search cell is sent 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.

9.5 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 calculated using the parallel downward accumulation algorithm described previously. Further, the contraction phase of the downward accumulation is not necessary since the information gathered during the contraction phase can be collected during the upward accumulation used for multipole expansion calculation. Note that shifting a local expansion 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 $n > p \log p$.

9.6 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

$$O\left(\frac{n\log n}{p} + \tau p \log n + \mu\left(\frac{n\log n}{p} + p\log^2 p\right)\right).$$

In the algorithm, the work done by each processor is proved to be of the same order. The actual work done in an execution can differ by a constant factor. For example, in assigning pointers during tree construction, some nodes in the tree may have 8 children and some nodes may have none or 2. Thus, the number of array entries per processor can vary by a constant factor. While load balancing can be used to distribute the load exactly, the overhead in performing load balancing is likely to outweigh any resulting benefit. This is because in the algorithm, the work done on the data elements received by a processor with less than average load is linear in the number of elements. Due to the larger constant associated with communication, more time will be spent in communicating the extra elements than is required to operate on the elements.

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.

Several experimental studies have shown that the time spent in force calculation dominates the total running time, despite its lower order complexity ($O(n)$ vs. $O(n \log n)$ sequentially). Here, force calculation includes computing multipole expansions, adding multipole expansions to multipole expansions, shifting multipole expansions to local expansions, adding local expansions to local expansions and evaluating local expansions. It is important to design parallel algorithms that have optimal speedup for force calculation part of the algorithm. While the total running time of our parallel algorithm is optimal $O\left(\frac{n \log n}{p}\right)$, the same complexity applies to force calculation as well.

.

Chapter 10

Conclusions and Open Problems

10.1 Conclusions

In this dissertation, we discussed the significance of distribution-independence for both data structures and algorithms and developed a suite of distribution independent techniques. Dependence on the distribution is disregarded in many studies where the analysis is given only for specific distributions. However, if distribution-independence is not achieved, an unpredictable behavior for arbitrary distributions is possible. In the worst-case the run-time of the algorithms and the size of data structures could be unbounded.

Space-filling curves have been widely used in a number of areas such as databases, scientific computing and parallel computing as an efficient tool to perform proximity preserving mappings from multidimensions to one dimension. We presented optimal and practically efficient algorithms to linearly order points in a multidimensional space using some space-filling curves, without making any assumptions on the precision or distribution of the points. Our technique is based on a comparison routine that can compare two points and decide which point precedes the other in the order induced by the space-filling curve. This allows us to use well-developed algorithms for sorting and good data structures such as balanced binary search trees to represent and process mappings of multidimensional points. It also allows us to efficiently process dynamic point sets. We applied our technique to applications drawn from databases and parallel computing. We presented an algorithm for storing database records involving multiple keys and a distribution-independent algorithm for answering range queries using space-filling curves. Application of our technique to parallel graph partitioning, domain decomposition and incremental load balancing are also presented. We demonstrate the practical utility of the algorithms through experimental evaluation.

Hierarchical data structures are used in many applications ranging from scientific computing to computer graphics. However, some of them (i.e., domain driven data structures) are not distribution-independent and some others (i.e., point driven data structures) do not provide efficient insertions and deletions. The DIAT tree data structure presented in this dissertation is distribution-independent and admits efficient dynamic operations. The size of a DIAT tree is $O(n)$ for n points. We presented three construction algorithms for DIAT trees in $O(n \log n)$ time. We also presented $O(\log n)$ time basic search and insertion/deletion algorithms. The DIAT trees and algorithms presented improve or match the results presented in [19], [30] and [93] and provide dynamic handling of the data points.

Note that the algorithms on DIAT trees use logarithm and floor operations. In the analysis of the algorithms an extended model of computation in which these operations can be performed in constant time is assumed. A lower bound of $\Omega(n \log n)$ exists for compressed hyperoctree construction in the standard algebraic model of computation [3]. Our construction algorithms can not be claimed to be optimal only because of the use of logarithm, floor and bitwise exclusive-or operations. We see no practical difficulty in assuming that these operations take constant time.

An important application area of hierarchical data structures is computational science. To use the DIAT trees in these applications, access patterns required by them should be satisfied. In this regard, we presented a unifying algorithm for all-nearest-neighbors, all-k-nearest-neighbors and spherical region queries which are common to scientific applications involving particle methods. By only changing simple functions, the same algorithm provides solutions to all three queries. The resulting algorithm is proved to be distribution-independent for the first two type of queries.

With these properties, the DIAT trees can potentially be used to solve any application that currently uses hyperoctrees. Furthermore, we believe that employing DIAT trees can eliminate the dependence on the distribution and result in faster algorithms. As evidence, we presented a distribution-independent algorithm for the FMM using DIAT trees. We extend the notion of well-separatedness by proposing three generalizing conditions. The conditions are respected by many applications using FMM and they provide a basis for checking the success of the algorithm.

The only other FMM algorithm comparable to ours is presented by Callahan *et al.* [25], using the more complicated fair split tree data structure. There are some important differences between our algorithm and theirs: Almost all the FMM algorithms used by practitioners involve hyperoctrees. DIAT trees contain same type of cells. The regularity in the shapes and locations of the cells makes it easy to perform error calculations.

Even though the computational power of sequential computers increases exponentially, it is still not adequate to satisfy the demands of the scientific community. Moreover, the limits to microprocessor technology justifies the need for parallel computing. Distribution-independence is even more crucial for parallel algorithms. If the run-time of an algorithm depends upon the distribution, deterministic distribution of the load between processors evenly is not an easy job. Thus, the algorithms usually rely on heuristic methods to perform data decomposition and load balancing which in turn entail unstructured communication. Rigorous analysis of the algorithms using heuristics is not possible in most cases.

We have depicted the problem of distribution-dependence for parallel algorithms by using the FMM as example. We argued that there are drawbacks of the parallel FMM algorithms because of the dependence of the run-time on the distribution of particles. Then, we presented the first provably efficient parallel FMM algorithm having all the following properties:

- 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.

We show that the problem can be solved in such a fashion on a realistic model of parallel computers, taking into detailed account the physically distributed nature of memory. Using such well-understood and simple primitives as parallel prefix, all-to-all communication and sorting is an important aspect of our algorithm. This feature enabled us to analyze the algorithm easily and should make it easy to implement.

Our parallel FMM algorithm employs parallel tree accumulations for multipole expansion calculation and local expansion calculation using partial local expansions. Algorithms for parallel tree accumulations are of interest by themselves because they are useful in many other applications. The parallel algorithms for upward and downward accumulations presented in this dissertation are both rigorously analyzable, taking into detailed account the distributed nature of memory in parallel computers. Our algorithm requires $O(\log n)$ iterations like the two EREW PRAM algorithms discussed. 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 our 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.

10.2 Open Problems

There are a number of areas in which the research presented in this dissertation can be extended. First of all, the algorithms and data structures presented are theoretically proved to be good in many ways. However, experimental evaluation of these results needs to be performed to show the practicality of the presented tools. A comparative study of the DIAT tree with similar data structures is an example possible extension in this regard. Similarly, the question of how an implementation of the parallel FMM algorithm presented compares with the best existing implementations that employ dynamic load distribution remains to be answered.

Secondly, in the run-time analysis of some of the algorithms presented, we assume an extended model of computation in which the logarithm and floor and bitwise-or operations can be performed in constant time. Although we verified the validity of the assumption through experiments, it is still of interest to obtain similar results for standard algebraic model of computation.

Another possibility of future work is to analytically investigate the clustering properties of space-filling curves. Studies on this topic are mostly experimental and analytical studies are usually limited to cases where points occupy every cell in a fixed grid. It is important to study the clustering properties of space-filling curves vs. traditional data structures such as multidimensional binary search trees for arbitrary point sets. In practice, it is computationally more efficient, especially in parallel, to use space-filling curves for ordering multidimensional data. Therefore, it is important to study the quality of the orderings obtainable by space-filling curves.

In Chapter 5, we devised comparison routines for multidimensional points using the Z-curve and the grey-code curve. However, several studies have shown that the Hilbert curve has better locality preserving properties. It would be of much interest to find if such a routine can be designed for the Hilbert space-filling curve as well.

The DIAT tree is an alternative to both domain driven and data driven hierarchical data structures. Since it efficiently supports many popular access patterns, it can replace the alternatives in many applications. This replacement usually can be done directly but sometimes it is necessary to design new algorithms to benefit from the DIAT tree as in the case of the FMM. Such applications still need to be investigated.

We presented algorithms for some access patterns common in many applications in Chapter 6. We proved that the algorithms for nearest neighbor queries are distribution-independent and efficient but for the spherical region query algorithm, we do not have a proof. Moreover, there are many other access patterns used in various applications which may benefit from DIAT trees provided that the algorithms are designed for these access patterns. For example, in some SPH applications, different size spherical regions should be used for each particle and the stability of the solution requires the interactions to be mutual in some others.

A number of applications require more computational power than a single processor can supply. We have efficiently parallelized the construction of DIAT trees. To achieve wide use of DIAT trees for parallel algorithms, parallel algorithms for various access patterns should be studied.

The presented parallel FMM algorithm is efficient for the whole process of tree construction and force calculation phases. However, the force calculation is the most time consuming phase. Thus, it is important to achieve optimal speedup for this phase as well. The analysis of our algorithm in this regard and possible extensions to achieve optimality are open questions. Our parallel FMM algorithm is the first algorithm to use a static decomposition to satisfactorily solve the problem. This raises the questions: can we expect to find even simpler algorithms without the need for dynamic load distribution? Are there any other problems for which regular algorithms using static decomposition are yet to be designed?

Bibliography

[1] 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.

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

[3] S. Aluru, *Distribution-independent hierarchical N-body methods*, Ph.D. thesis, Iowa State University, 1994.

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

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

[6] S. Aluru, F.E. Sevilgen, Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves, *Proc. International Conference on High-Performance Computing, HiPC '97* (1997) 230-235.

[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] S. Aluru and F. Sevilgen, Points Need Not Lie on a Grid: Space-Filling Curves for Arbitrary and Dynamic Point Sets with Applications, Technical Report No. NMSU-CSTR-9704, New Mexico State University Computer Science Department, March 1997.

[9] S. Aluru and F. Sevilgen, Dynamic Compressed Hyperoctrees with Application to the N-body Problem, *Proc. Foundations of Software Technology and Theoretical Computer Science* (1999).

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

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

[12] Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.Y., An optimal algorithm for approximate nearest neighbor searching in fixed dimensions, *Journal of the ACM, 45* (1998) 891-923.

[13] Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.Y., An optimal algorithm for approximate nearest neighbor searching, *Proc. ACM-SIAM Symposium on Discrete Algorithms* (1994) 573-582.

[14] 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.

[15] 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.

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

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

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

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

[20] Bern, M., Approximate closest-point queries in high dimensions, *Information Processing Letters, 45* (1993) 95-99.

[21] Bespamyatnikh, S.N., An optimal algorithm for closest-pair maintenance, *Discrete Comput. Geom., 19* (1998) 175-195.

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

[23] 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.

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

[25] 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.

[26] Callahan, P.B., Kosaraju, S.R., Algorithms for dynamic closest pair and n-body potential fields, *Proc. ACM-SIAM Symposium on Discrete Algorithms* (1995) 263–272.

[27] T.M. Chan, Approximate nearest neighbor queries revisited, *Proc. ACM Symposium on Computational Geometry* (1997) 352-358.

[28] Chazelle, B., A theorem on polygon cutting with applications, *Proc. Foundations of Computer Science* (1982) 339-349.

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

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

[31] K. L. Clarkson, An algorithm for approximate closest-point queries, *Proc. ACM Symposium on Computational Geometry* (1994) 160-164.

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

[33] J.G. Cleary and G. Wyvill, Analysis of an algorithm for fast ray tracing using uniform space subdivision, *Visual Computer, 4(2)* (1988) 65-83.

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

[35] 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.

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

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

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

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

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

[41] R.A. Finkel and J.L. Bentley, Quad trees: A data structure for retrieval on composite keys, *Acta Informatica, 4(1)* (1974) 1-9.

[42] Frederickson, G.N., A data structure for dynamically maintaining rooted trees, *Proc. ACM-SIAM Symposium on Discrete Algorithms* (1993) 175-194.

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

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

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

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

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

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

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

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

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

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

[53] T. Hsu and V. Ramachandran, Efficient massively parallel implementation of some combinatorial algorithms, *Theoretical Computing Science, 162* (1996) 297-322.

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

[55] P. Indyk and R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensioality, *Proc. ACM Symposium on Theory of Computation* (1998) 604-612.

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

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

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

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

[60] 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.

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

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

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

[64] X. Liu and G.F. Schrack, An algorithm for encoding and decoding the 3-D Hilbert ordering, *IEEE Transaction on Image Processing, 6(9)* (1997) 1333-1337.

[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)* (1991) 1128-1147.

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

[72] 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.

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

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

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

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

[77] 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.

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

[79] 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.

[80] 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.

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

[82] 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.

[83] 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).

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

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

[86] 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)* (1990) 83-90.

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

[88] 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).

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

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

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

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

[93] Schwarz, C., Smid, M., Snoeyink, J., An optimal algorithm for the on-line closest-pair problem, *Algorithmica, 12* (1994) 18-29.

[94] T. Sellis, N. Roussopoulos and C. Faloutsos, Multidomensional access methods: Trees have grown everywhere, *Proc. International Conference on Very Large Data Bases* (1997) 13-15.

[95] F. Sevilgen, S. Aluru, and N. Futamura, A provably optimal, distribution-independent parallel fast multipole method, *Proc. International Parallel and Distributed Computing Symposium* (2000), to appear.

[96] F. Sevilgen, S. Aluru, and N. Futamura, Distributed Memory Tree Accumulations, *Proc. Parallel and Distributed Computing Systems* (1999) 389-395.

[97] F. Sevilgen and S. Aluru, A Unifying Data Structure for Hierarchical Methods, *Proc. Supercomputing'99* (1999).

[98] J. Sibeyn, F. Guillaume, and T. Seidel, Practical parallel list ranking, *Journal of Parallel and Distributed Computing, 56* (1999) 156-180.

[99] J. Sibeyn and I. Stadtwald, List ranking on meshes and networks. *Proc. Computing Science in Netherlands* (1994) 271-280.

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

[101] Sleator, D.D., Tarjan, R.E., A data structure for dynamic trees, *Journal of Computer and System Sciences 26* (1983) 362-391.

[102] 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.

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

[104] 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.

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

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

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

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

VITAE

Education

- Ph.D. in Computer Science at Syracuse University.
- M.S. in Computer Science at Syracuse University.
- M.S. in Systems and Control Engineering at Bogazici University.

  Thesis: Open-Loop and Closed-Loop Observer Design for Generalized Linear Time-Invariant Systems
- B.S. in Computer Engineering at Bogazici University.

  Thesis: Design & Implementation of a bar-code system.

Honors and Awards

- Syracuse University Graduate Fellowship, 1999-2000.
- Scholarship for graduate study in USA from Gebze Institute for Advance Technologies, Gebze, Turkey, 1993.
- Graduate School Scholarship from Bogazici University, Istanbul, Turkey, 1991-1993.
- Ranked in first 250 (out of more than 600,000 student) in the Central University Entrance Exam and admitted to the Bogazici University, 1985.
- Ranked in first 96 (out of about 300,000 students) in the Entrance Exam for admission to the Ankara Science High School, 1982.

Work Experience

- June 1998-Aug 1999, Research Assistant, New Mexico State University.
- Spring 1998, Teaching Assistant (CIS-675 Intr. to Algorithms), Syracuse University.
- May 1997-Dec 1997, Research Assistant, New Mexico State University.
- Spring 1996, Teaching Assistant (CIS-600 P. of Databases Systems), Syracuse University.
- Spring 1995, Grader (CIS-675 Intr. to Algorithms), Syracuse University.
- Fall 1994, Tutor (CIS-665 Computer Architecture), Syracuse University.
- Feb 1991-Feb 1993, Research Assistant, Computer Center, Bogazici University, Turkey.

Research Interests

- Distribution Independent Methods for Scientific Applications

  Spatial Data Structures

  Algorithms
- Parallel Computing

  Parallel Algorithms

  Parallel Computation Models
- N-body Methods

  Efficient Construction and Incremental Updates of N-body Trees

  Parallel Algorithms for N-body Methods

Refereed Publications

• Fatih Sevilgen and Srinivas Aluru, A Provably Optimal, Distribution-Independent Parallel Fast Multipole Method, IPDPS (2000).

• Fatih Sevilgen and Srinivas Aluru, A Unifying Data Structure for Hierarchical Methods, Supercomputing (1999).

• Fatih Sevilgen, Srinivas Aluru and Natsuhiko Futamura, Distributed Memory Tree Accumulations, Parallel and Distributed Computing Systems (1999), 389-395.

• Fatih Sevilgen, Srinivas Aluru and Natsuhiko Futamura, A Provably Optimal Distribution Independent Parallel N-body Method, SIAM Annual Meeting (1999).

• Srinivas Aluru and Fatih Sevilgen, Dynamic Compressed Hyperoctrees with Application to the N-body Problem, Foundations of Software Technology and Theoretical Computer Science (1999).

• S. Aluru, G.M. Prabhu, J. Gustafson, and F.E. Sevilgen, Distribution-Independent Hierarchical Algorithms for the N-body Problem, Journal of Supercomputing, 12 (1998) 303-323.

• Srinivas Aluru and Fatih E. Sevilgen, Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves, High-Performance Computing (1997).

• K. Dincer, H.R. Topcuoglu, F.E. Sevilgen, P.C. Coddington, and S. Ranka, Parallel Simulation of Dynamically Triangulated Random Surfaces, IASTED Eighth International Conference on Parallel and Distributed Computing and Systems (1996).

Manuscripts under Review

• Srinivas Aluru and Fatih Sevilgen, Space-Filling Curves For Arbitrary And Dynamic Points With Applications, submitted to The Computer Journal.

• Fatih Sevilgen, Srinivas Aluru and Natsuhiko Futamura, Distributed Memory Tree Accumulations, submitted to IEEE TPDS.

• Fatih E. Sevilgen and Srinivas Aluru, A Provably Efficient, Distribution-Independent Parallel Fast Multipole Method, submitted to IEEE TPDS.