**A Multidimensional Access Method: the Interpolation-Based Grid File,
its Parallelization and Comparison to the MFP Tree**

by

Weiping Wang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario

January 2004

© 2004, Weiping Wang

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89869-5

Our file Notre référence

ISBN: 0-612-89869-5

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

In this work, we study a multi-dimensional access method, the Interpolation-Based Grid File (IBGF), and its parallelization. We compare it to the Multi-dimensional Fixed Partition (MFP) tree, a parallel multi-dimensional access method, with regards to its disk utilization and the performance of its maintenance operations and queries. We study their similarities and differences.

We improve and implement the IBGF and parallel IBGFs. We design and implement the dynamic parallel IBGFs. We also implement dynamic MFP tree. We run experiments in two different parallel environments, Sunfire system and Beowulf system, on four types of data distributions using four, eight, and sixteen processors.

According to our experimental results, the MFP tree has lower disk utilization than the IBGF. The maintenance of the MFP tree is faster and more efficient than that of the IBGF. The range query performance on the IBGF is better than that on the MFP tree.

Acknowledgements

I would like to thank everyone involved in my work directly or indirectly.

I want to thank my supervisor Jörg-Rüdiger Sack for his support for my work both financially and academically. He offered me the chance to do this study and always gave me important instructions and suggestions at the right time. I benefited greatly from his rich experience in GIS applications.

I want to thank my co-supervisor Doron Nussbaum for his guidance and advice. A major part of my work is based on his study of the MFP Tree. I got a lot of inspiration from his work. I also enjoyed the moments of discussion on the details of my study with him. He always showed lots of interest and gave me valuable academic advice.

I also want to thank David Bickle from the HPCVL Lab for his support in using the Beowulf and Sunfire systems. I really appreciated his patience and technical knowledge.

I would like to thank Robert Kane and Ji Chen in the Paradigm Lab for supporting me with the Sun Solaris System and PCs. I would not have been able to carry out my study as quickly and smoothly as I did without their help.

Finally, I would like to thank my family, my husband Xin, my mother-in-law Cuizhu, and my sister-in-law Hua. Because of them, I was able to devote most of my time to the study. Their spiritual support and encouragement was very important for me to keep working on the thesis.

Table of contents

ABSTRACT.....	I
ACKNOWLEDGEMENTS	II
CHAPTER 1 INTRODUCTION	1
CHAPTER 2 LITERATURE REVIEW	3
2.1 Requirements of Multi-Dimensional Access Methods.....	3
2.2 Grid File and its Variants.....	6
2.2.1 The Grid File	6
2.2.2 The IBGF	7
2.2.3 The BANG File	7
2.2.4 The Twin Grid File.....	8
2.2.5 The Multi-layer Grid File	8
2.3 Parallel Grid Files.....	9
2.4 MFP Tree.....	11
CHAPTER 3 METHODS AND ALGORITHMS.....	12
3.1 Conventions and Terminology.....	12
3.2 IBGF.....	14
3.2.1 Concepts	14
3.2.2 Maintenance of an IBGF	25
3.2.3 Queries on an IBGF	37
3.3 Parallel IBGFs.....	47
3.3.1 Creation of a Mapping Graph.....	48
3.3.2 Static Parallel IBGFs	50
3.3.3 Dynamic Parallel IBGFs.....	60

3.4 MFP Tree.....	63
3.4.1 Concepts	63
3.4.2 Maintenance of an MFP Tree	75
3.4.3 Queries on an MFP Tree.....	87
 CHAPTER 4 EXPERIMENTAL RESULTS	90
4.1 Data Sources	90
4.1.1 Uniform Data.....	90
4.1.2 Correlated Data.....	91
4.1.3 Hotspot Data.....	91
4.1.4 Clusters	92
 4.2 Parallel Environments	95
4.2.1 Beowulf System.....	95
4.2.2 Sunfire System.....	95
 4.3 Experimental Results and Analysis.....	96
4.3.1 Directory Size	96
4.3.2 Partitions in IBGF and MFP Trees	98
4.3.3 Disk Utilization	105
4.3.4 Creation Time.....	108
Set	112
4.3.5 Query Time.....	117
 CHAPTER 5 SUMMARY AND FUTURE WORK	126
 REFERENCES.....	127

List of Figures

Figure 3.2.1 A Running Example	15
Figure 3.2.2 The Creation of an IBGF	17
Figure 3.2.3 The Tree Corresponding to an IBGF	24
Figure 3.2.4 Transferring a Range Query into Multiple Point Queries	41
Figure 3.2.5 Proof of the Nearest Neighbor Query Algorithm	45
Figure 3.2.6 The Nearest Neighbor Query Algorithm	45
Figure 3.2.7 The Best Case and the Worst Case of Nearest Neighbor Query	46
Figure 3.3.1 How to Calculate the Proximity of Two IBGF Blocks	50
Figure 3.3.2 Improved Seeding Method	53
Figure 3.3.3 Improved Expanding Method.....	57
Figure 3.3.4 The First Example of Bonki et al.'s Partition Expanding Method	58
Figure 3.3.5 The Second Example of Bonki et al.'s Partition Expanding Method.....	59
Figure 3.3.6 Dynamic Insertion of Data into Parallel IBGFs	61
Figure 3.4.1 Create a 2×2 MFP Tree	65
Figure 3.4.2 Create a 2×4 MFP Tree	66
Figure 3.4.3 The Tree Corresponding to a 2×2 MFP, and the Ids, the Morton Areas, and the Levels in the MFP Tree	73
Figure 3.4.4 The Ids, the Morton Areas and the Levels in a 2×4 MFP Tree	74
Figure 3.4.5 The Mapping Scheme of an MFP Tree	77
Figure 4.1 Uniform Data Set.....	93
Figure 4.2 Correlated Data Set.....	93
Figure 4.3 Hotspot Data Set.....	94
Figure 4.4 Cluster Data Set.....	94
Figure 4.5 Uniform Data Set in an IBGF.....	99
Figure 4.6 Correlated Data Set in an IBGF.....	99
Figure 4.7 Hotspot Data Set in an IBGF	100
Figure 4.8 Cluster Data Set in an IBGF	100
Figure 4.9 Uniform Data Set in 4-Disk Parallel IBGFs.....	101

Figure 4.10 Uniform Data Set in 2×2 Disk MFP Tree	101
Figure 4.11 Correlated Data Set in 4-Disk Parallel IBGFs.....	102
Figure 4.12 Correlated Data Set in 2×2 Disk MFP Tree	102
Figure 4.13 Hotspot Data Set in 4-Disk Parallel IBGFs.....	103
Figure 4.14 Hotspot Data Set in 2×2 Disk MFP Tree.....	103
Figure 4.15 Cluster Data Set in 4-Disk Parallel IBGFs.....	104
Figure 4.16 Cluster Data Set in 2×2 Disk MFP Trees	104

List of Charts

Chart 4.1 Comparison of Disk Utilization between the IBGF and the MFP Tree.....	107
Chart 4.2 Comparison of Creation Time on Uniform Data Set	113
Chart 4.3 Comparison of Creation Time on Correlated Data Set	113
Chart 4.4 Comparison of Creation Time on Hotspot Data Set	114
Chart 4.5 Comparison of Creation Time on Cluster Data Set	114
Chart 4.8 Analysis of Creation Time on Hotspot Data Set.....	116
Chart 4.9 Analysis of Creation Time on Cluster Data Set.....	116
Chart 4.10 Comparison of Response Time on Uniform Data Set.....	120
Chart 4.11 Comparison of Response Time on Correlated Data Set	120
Chart 4.12 Comparison of Response Time on Hotspot Data Set.....	121
Chart 4.13 Comparison of Response Time on Cluster Data Set.....	121
Chart 4.14 Comparison of Running Time on Uniform Data Set	123
Chart 4.15 Comparison of Running Time on Correlated Data Set	123
Chart 4.16 Comparison of Running Time on Hotspot Data Set	124
Chart 4.17 Comparison of Running Time on Cluster Data Set	124

List of Tables

Table 4.1 Comparison of Directory Size in the Grid File and the IBGF	97
Table 4.2 Comparison of Disk Utilization in the IBGF and the MFP Tree.....	107
Table 4.3 Comparison of Static Creation Time (ms).....	111
Table 4.4 Comparison of Dynamic Creation Time (ms)	111
Table 4.5 Compare Dynamic Creation Time (ms) to Static Creation Time (ms).....	112
Table 4.6 Comparison of Query Response Time.....	122
Table 4.7 Comparison of Query Running Time	125

List of Algorithms

Algorithm 3.2.1: Creating an IBGF	27
Algorithm 3.2.2: ConstructIBGFDir()	28
Algorithm 3.2.3: CalculateIBGFBlock()	28
Algorithm 3.2.4: CalculateIBGFRegion()	30
Algorithm 3.2.5: SplitIBGFBlock()	31
Algorithm 3.2.6: Inserting Points into an IBGF	32
Algorithm 3.2.7: CalculateIBGFBuddyBlock()	33
Algorithm 3.2.8: Deleting Points from an IBGF	34
Algorithm 3.2.9: Point Query on an IBGF – Method 1	38
Algorithm 3.2.10: Point Query on an IBGF – Method 2	39
Algorithm 3.2.11: Range Query on an IBGF	41
Algorithm 3.2.12: Nearest Neighbor Query on an IBGF	43
Algorithm 3.3.1: Creating the Mapping Graph for an IBGF	49
Algorithm 3.3.2: Selecting Each Partition's Seed	51
Algorithm 3.3.3: Expanding Partitions in the Mapping Graph for an IBGF	54
Algorithm 3.3.4: Inserting an IBGF Block into Parallel IBGFs	60
Algorithm 3.4.1: Creating an MFP Tree	78
Algorithm 3.4.2: CalculateMFPDir()	78
Algorithm 3.4.3: SplitMFPBlock()	79
Algorithm 3.4.4: Inserting Points into an MFP Tree	80
Algorithm 3.4.5: CalculateMFPBuddyBlocks()	82
Algorithm 3.4.6: Deleting Points from an MFP Tree	83
Algorithm 3.4.7: CalculateBlockId()	86
Algorithm 3.4.8: CalculateMFPBlockRegion()	87

Chapter 1 Introduction

Multi-dimensional applications in GIS, CAD, medicine, molecular biology, and other disciplines are important areas of computer usage. One-dimensional access methods such as hash tables and B-Trees cannot deal with those applications adequately, so multi-dimensional access methods are required. Due to the size of data sets in multi-dimensional applications, they are stored in secondary memory. Furthermore, they should be able to conduct operations on multi-dimensional data, including creation, update, and query, correctly and efficiently.

As the size of applications became larger and parallel environments emerged, storing multi-dimensional data set on multiple disks became possible and necessary. Some applications, such as the NASA Earth Observing system, have large storage requirements measured in terabytes. The use of multiple disks also aims to improve the performance of operations on multi-dimensional data sets. In order to store data on multiple disks, some methods are necessary to distribute the data. If the number of the disks is M , the data should be scattered evenly such that parallel operations performed on the data should take about $1/M$ of the amount of time needed to run the same operations sequentially. Multi-dimensional applications are usually dynamic. Parallel multi-dimensional access methods should meet this dynamic requirement.

In this thesis, we study two multi-dimensional access methods, the IBGF and the MFP Tree. We then implement these methods, run experiments on them in parallel environments, and analyze the experimental results.

In Chapter 2, we review the related literature. We identify the requirements of multidimensional applications, introduce the characteristics of the grid file and its variants, explain its parallelization, and review the MFP Tree briefly.

In Chapter 3, we focus on the implementation of the IBGF and the MFP Tree, give a detailed description of concepts and the related algorithms and operations, including those for creation, update and query. We also explain the method used to parallelize the IBGFs and to update them dynamically.

In Chapter 4, we show our experimental results on the IBGF, the parallel IBGFs, and the MFP Tree. Disk utilization, creation time, update time, and query time are compared and analyzed.

In Chapter 5, we conclude and demonstrate possible future work.

Chapter 2 Literature Review

In this chapter, we briefly review some related literature. In Section 2.1, we state important requirements for multi-dimensional access methods. In Section 2.2, we describe a multi-dimensional access method, the grid file, and its variants. In Section 2.3, we review existing work on parallelizing grid files. In Section 2.4, we briefly introduce another multi-dimensional access method, the MFP Tree.

2.1 Requirements of Multi-Dimensional Access Methods

Based on the properties of multi-dimensional data and their applications, Volker Gaede and Oliver Günther summarize ten requirements of Multi-Dimensional Access Methods [6]. We cover the five most relevant of these in detail, while simply listing the other five.

1. Dynamic

This requirement results from the dynamic characteristics of multi-dimensional applications, in which data can be inserted or deleted at any time in any order. Dynamic data maintenance is important and necessary to keep up-to-date information in multi-dimensional applications.

Most single disk spatial access methods take this requirement into consideration, but as we will see later, some parallel multi-dimensional access methods do not meet this requirement very well. Our work includes realizing a dynamic parallel multi-dimensional access method, the parallel IBGFs.

2. Secondary storage management

Because of the huge data size involved in most multi-dimensional applications, which could be in the gigabyte or even terabyte range, it is impossible to hold all the data in main memory. Secondary memory storage, usually the disk, is necessary.

Due to the mechanisms of secondary memory, access time is much slower than when using main memory. Therefore, a spatial access method should keep secondary memory accesses to a minimum.

3. Independence of the input data

A multi-dimensional access method should be efficient regardless of the distribution of the insertion sequence of the input data. This requirement is very important for data distributed differently along the various dimensions.

We will observe later that a grid file is not suited to a non-uniform situation, and several variants of it are designed to solve this problem.

4. Time efficiency

Queries on multi-dimensional data should be fast, and maintenance should be efficient. Volker Gaede et al. propose the logarithmic worst-case search performance as the criteria for time efficiency.

However, they do not consider the different access mechanisms of secondary memory from main memory. Since accessing secondary memory is far slower than accessing main memory, several multi-dimensional access methods aim to decrease the number of disk accesses.

Our work aims to decrease the time to access both secondary and main memory, and balances the dynamic requirement and time efficiency.

5. Space efficiency

This requirement refers to the disk utilization. A multi-dimensional access method should use the available storage efficiently to achieve a certain level of utilization. Low space efficiency also reduces time efficiency. To achieve higher space efficiency, the amount of memory used for indexing should be small compared to the data to be addressed, and the data should be stored intensively.

The other five requirements

The other five requirements of multi-dimensional access methods are as follows.

Broad range of supported operations: An access method should not support just one particular type of operation (such as a range query) at the expense of other tasks (such as deletion). **Simplicity:** An access method should be implemented simply so as to be sufficiently robust for large-scale applications. **Scalability:** An access method should adapt well to the growth of data size. **Concurrency and recovery:** When multiple users concurrently update, retrieve, and insert data, an access method should provide robust techniques for transaction management without significant performance penalties. **Minimum impact:** The integration of an access method into a database system should have minimum impact on existing parts of the system.

2.2 Grid File and its Variants

2.2.1 The Grid File

J. Nievergelt et al. first studied multidimensional application problems and introduced the grid file as a multi-dimensional access method in 1984 [1]. The grid file deals with all dimensions symmetrically by putting parallel hyper-planes, also called partitions, evenly in every dimension of the space. These partitions are orthogonal to each dimension's axis, and thus divide the multidimensional space into adjacent and non-intersected grid blocks, which form multi-dimensional orthogonal grids. A disk unit, or a page, corresponds to a grid block or a larger block formed by several adjacent grid blocks, and the data in the corresponding block is stored in this page.

The directory of a grid file has D one-dimensional arrays, also called “scales,” to store each dimension's partitions. It also has one D -dimensional array to store the page address of each grid block and to store the number of points in the page (D here is the number of dimensions). The grid file guarantees one disk access to retrieve data if the directory is kept in main memory, or two disk accesses if not.

The problem of the grid file is the super-linear growth of the directory size. As the data tends to be non-uniformly distributed, the directory size increases exponentially with the number of dimensions.

2.2.2 The IBGF

The grid file implicitly represents grid blocks by storing the partitions in each dimension into scale arrays. Thus each partition cuts through the whole space unnecessarily, which results in the super-linear growth of the number of grid blocks.

To achieve linear growth in the directory, M. Ouksel combines the grid file and the interpolation-based index [7] and created the IBGF [3]. Instead of using scale arrays, it represents each grid block explicitly so that there is only one directory entry, or one grid block, per page. In the IBGF, a partition only divides one over-loaded grid block, instead of the whole universe, into 2 blocks.

Our approach is inspired by Ouksel's work. Based on his paper, we develop the theory of a file system and implement it. We call it the IBGF, although it is not exactly the same as what Ouksel introduced. Ouksel represents each grid block with a multi-dimensional point, while we represent it with an integer to build a link between a multi-dimensional block and a one-dimensional integer. The details are presented in Chapter 3.

2.2.3 The BANG File

In both the grid file and the IBGF, the disk utilization is less than 70%. To obtain a better percentage, Michael Freeston proposed a new file structure, the BANG file (the Balanced and Nested Grid File) [10]. The main difference between the BANG file and the grid file is that the BANG file permits the overlay of grid blocks. It redistributes data among grid blocks to obtain higher disk utilization.

Although it increases disk utilization, the BANG file is unpredictable and difficult to maintain because grid block replacements are allowed.

2.2.4 The Twin Grid File

Andreas Hutflesz et al. introduced the twin grid file to increase the disk utilization [11]. As indicated by the name “twin,” two grid files are used and the relationship between them is equal and complementary. Both of them span the whole space. The distribution of the data among the two files is performed dynamically.

They reported an average disk utilization of 90% for the twin grid file (compared to 69% for the grid file) without substantial performance penalties. They claimed that the search efficiency of the twin grid file was competitive with the grid file, although the twin grid file was somewhat inferior to the grid file for smaller query ranges.

However, the dynamic distribution of data between the two files causes numerous reorganizations of the data pages and results in greater file maintenance complexity.

2.2.5 The Multi-layer Grid File

Grid files consider only point objects. However, in most multidimensional applications, the geometry of objects cannot be ignored. It is quite possible for a non-zero size object to be cut by partitions such that this object has to be saved in several pages, resulting in data redundancy and operational inefficiency.

Hans-Werner Six et al. designed the multi-layer grid file to handle extended objects [12]. The multi-layer grid file consists of an ordered sequence of grid layers. Each of these layers is a grid file that covers the whole universe. The first layer’s grid

file stores the objects that can fit into its blocks without clipping. If there is any object that does not fit in, a second layer grid file is created, and a third layer if necessary. A new object is inserted by traversing the sequence of layers until one is found where the object can be inserted without clipping. If in one of the grid files a new partition would split some objects, those objects have to be moved to another layer.

Possible disadvantages of the multi-layer grid file include low storage utilization and expensive directory maintenance.

2.3 Parallel Grid Files

As the size of data has become larger and larger, parallel file systems have emerged to improve the performance of multi-dimensional applications. The idea is to distribute the data set onto several disks such that any query on them accesses those disks evenly. Thus, if there are M disks, the parallel query time should be $1/M$ that obtained in sequential environments.

To achieve this, several parallel methods have been proposed. There are two kinds of methods to parallelize grid files. Index-based methods are extended from the methods used to de-cluster Cartesian Product files. Proximity-based methods calculate the proximity between grid blocks and transform parallel grid file problems into graph partition ones. The former is easier and faster, but disregards the possible non-uniform distribution and the dynamic characteristic of multi-dimensional data sets. The latter adapts to non-uniform data sets very well.

Bonki et al. have studied index-based de-clustering algorithms, disk modulo, fieldwise Xor, and Hilbert curve, and extended them to the grid file [4]. They then

propose a proximity-based method, known as the Minimax spanning tree algorithm, to parallelize grid files [4]. We study their work and develop improved methods based upon their algorithms. In Section 3.3, we discuss the proximity-based parallel method in detail.

2.4 MFP Tree

In 2001, Doron Nussbaum proposed, in his Ph.D. thesis, the Multi-dimensional Fixed Partition (MFP) Tree [2]. As with a grid file, in an MFP Tree hyper-planes are placed vertical to each dimension's axis and divide the whole space into blocks. The difference is that in an MFP Tree several hyper-planes form a fixed partitioning schema and are posed on an over-loaded block together.

The MFP Tree is mainly designed for a parallel environment. A particular schema has been developed to map pages onto processors. It has obvious advantages for dynamic applications in such environments. A detailed discussion concerning the concepts, algorithms and operations of the MFP Tree is given in Section 3.4.

Chapter 3 Methods and Algorithms

In this chapter, we discuss two multi-dimensional access methods, the IBGF and the MFP Tree, in detail.

Section 3.1 goes over the conventions used in this and the next chapter. Section 3.2 introduces the IBGF and its algorithms. Section 3.3 discusses how to parallelize IBGFs. Finally, Section 3.4 introduces the MFP Tree and its algorithms.

3.1 Conventions and Terminology

To make the explanation clear and concise, we outline the conventions that will be used. We denote by:

D : the number of dimensions;

M : the number of disks;

N : the number of blocks or pages; and

IBGF: the IBGF.

We also define several terms. Our work focuses on multi-dimensional objects without extension (i.e. points). A **point** is a D -dimensional point represented by $P(X_1, \dots, X_i, \dots, X_D)$, and X_i is the point's coordinate in the i -th dimension ($i = 1..D$). The **universe** is a D -dimensional space where all the input points are located. A **block** is a D -dimensional interval $I = I_1 \times \dots \times I_i \times \dots \times I_D$, in which $I_i = [l_i, u_i)$. l_i is the lower bound coordinate of the block in dimension i , and u_i is the upper bound coordinate of the block in dimension i . The **block of a point** $P(X_1, \dots, X_i, \dots, X_D)$ is a block I such that $l_i \leq X_i < u_i$ for $i = 1..D$. The **coordinate of a block** is the coordinate of the block's

Euclidean center. The **position of a block** is defined as the intersection of its lower bounds in all dimensions, which forms a point in the universe $P(X_1, \dots, X_i, \dots, X_D)$, where $X_i = l_i$. The **size of a block** is an array of D elements, whose i -th element is the block's width in the $(i+1)$ -th dimension. The **region of a block** is composed of its position and its size. The **capacity of a page** in a disk is the number of points a page can store. A block is called **overloaded** when the number of points in the block is more than a page's capacity. The **disk utilization** is the percentage of the disk space used to store data. It is calculated as follows.

$$\text{Disk Utilization} = \text{Number of Points in a File} / (\text{Page Capacity} * \text{Number of Pages})$$

3.2 IBGF

In this Section, we introduce the IBGF, discussing its concepts, its properties, ways of maintaining it, and queries that can be run on it.

M. Ouksel introduced the IBGF, the Interpolation-Based Grid File in [3]. He used a point to represent each block explicitly to fix the super-linear growth problem of the directory size in a grid file. The problem of using a point is that the size of each block undetermined. Thus the query on his IBGF is difficult to implement. He suggested using the same data structure for both the data and the directory, but he did not give clear idea about how to implement the IBGF. Based on his work, we made improvements. Instead of using a point, we use two integers to represent each block such that both the size and the position of a block can be determined. We simplified the implementation of IBGF. We also implement the point query, the range query, and the nearest neighbor query on the IBGF.

3.2.1 Concepts

The following contents are our theory about the improved IBGF. First, we need to know how the space is partitioned in an IBGF. Figure 3.1 is a set of two-dimensional points in the universe. We take it as an input example to explain the generation of an IBGF. There is, in the figure, an ID beside each point. The ID also indicates the sequence in which the points are inserted.

Figure 3.2.1 A Running Example

We divide the space into non-intersected multi-dimensional blocks, called **IBGF blocks**, in the following way. The first block is the whole universe. When the block representing the universe is overloaded, we put a division, called an **IBGF division**, vertical to its border in the first dimension partitioning it into two equal-sized sub-blocks. When either of these two sub-blocks is overloaded, we put another division vertical to its border in the second dimension to partition it into two equal sized smaller sub blocks. The next division for one of these two sub-blocks is put in the third dimension, or in the first dimension if only two dimensions are considered. The process continues until all the points are inserted into the IBGF. We state the following important properties of the IBGF based on the above description.

Property 3.2.1: The divisions in an IBGF are put in different dimensions circularly. If an IBGF block is created by a division in the i -th dimension during the last step, the next division in the block is put in dimension $(i \bmod D + 1)$.

Property 3.2.2: A division in an IBGF is put in the middle of a border of an overloaded block.

Figure 3.2.2, by using the input sample in Figure 3.2.1, illustrates how the blocks of an IBGF are generated. We assume the page capacity is two points. The points are inserted one by one in the order of increasing ids. When the number of points in a block exceeds the page capacity, the block is divided into two by a division in the middle of it. The divisions are placed in different dimensions in a circular way. The sub-figures show the steps in which a division is placed, and the double line in each sub-figure shows the new added divisions in each step.

The process to generate an IBGF resembles the generation of a K-D tree. However, the K-D tree is a data-driven data structure, where the division is decided by the points themselves, while the IBGF is a space-driven data structure, where the division is decided by the space in which the points are located.

We now introduce the concepts and the lemmas necessary to demonstrate how an IBGF is maintained. We construct a **binary tree corresponding to an IBGF** as follows. Let its root correspond to the universe. When an IBGF block is overloaded and divided into two sub- blocks, we append two children to the **corresponding node of the original IBGF block**. The left child corresponds to the sub-block whose coordinate is smaller than the division's coordinate, and the right child is the other sub-block. The node corresponding to the original block becomes the **corresponding node of the division**. We repeat the process until there is no overloaded block. Figure 3.2.3 shows the tree corresponding to the IBGF in Figure 3.2.2

(1) After P3 is inserted, the 1st division is placed

(2) After P6 is inserted, the 3rd division is placed.

(3) After P10 is inserted, the 5th division is placed.

(4) P12 is inserted. The IBGF is created.

Figure 3.2.2 The Creation of an IBGF

Since we use terms such as the **ancestor**, the **descendant**, the **parent**, the **child**, and the **sibling** to represent the relations between two nodes in a binary tree, we adopt the same terms to represent the relations between IBGF blocks and divisions. For example, let division P divide block B into two sub-blocks B' and B'' , and the coordinate of B' be less than that of B'' . We say B is the parent block of B' and B'' ; P is the parent division of B' and B'' ; B' is the left child of B and P , and B'' is the right child of B and P . When we talk about IBGF blocks, we usually consider only the leaf blocks. The terms “parent blocks” or “ancestor blocks” are only used for the convenience of explanation.

We define the **level of an IBGF block** as the depth of the node corresponding to it, and similarly the **level of an IBGF division** as the depth of its corresponding node in the tree. Given the level of an IBGF division, we can calculate the dimension in which the division occurred and the width of the division’s child blocks in that dimension. The following lemmas give the details.

Lemma 3.2.1: A division at level L partitions an IBGF block in dimension $L \bmod D + 1$.

Proof: We prove the lemma by induction on L . First, we know the division at level 0 divides the universe in dimension 1, so the lemma is correct when $L = 0$. Then, we suppose all the divisions at level $L - 1$ are in dimension $(L - 1) \bmod D + 1$. Consider a division at level L ($L > 0$) dividing an IBGF block, B . The parent division of block B is at level $L - 1$, and thus is in dimension $(L - 1) \bmod D + 1$. Then the division at level L is put in dimension $((L - 1) \bmod D + 1) \bmod D + 1$ according to Property 3.2.1.

Let $L - 1 = nD + k$ ($0 \leq k < D, n \geq 0$), then $L = nD + k + 1$, and

$$((L - 1) \bmod D + 1) \bmod D + 1$$

$$\begin{aligned}
&= ((nD + k) \bmod D + 1) \bmod D + 1 \\
&= (k + 1) \bmod D + 1 \\
&= (nD + k + 1) \bmod D + 1 \\
&= L \bmod D + 1
\end{aligned}$$

Thus the Lemma is correct for any L , $L \geq 0$.

Q.E.D.

Lemma 3.2.2: If the level of a block is $L + 1$, the width of the block in dimension $L \bmod D + 1$ is $2^{-(L/D+1)}$ times the universe's width in that dimension.

Proof: Let B be a block at level $L + 1$. The parent division of B is then at level L , let it be P . Let A be the corresponding node of P in the tree. Consider each ancestor of node A , and let its depth be L' . All the ancestors whose depths meet the condition $L' \bmod D = L \bmod D$ correspond to the divisions in the same dimension as P . From the root, whose depth is 0, to node A , there are $L/D + 1$ such nodes. Since each division divides an IBGF block into two equal sized sub-blocks, the width of each child block is half that of its parent block. The bisections propagate from the root to node A for $L/D + 1$ times. Thus, the width of B in dimension $L \bmod D + 1$ is $2^{-(L/D+1)}$ times the width of the universe in that dimension. Q.E.D.

Based on the above lemmas and properties about an IBGF, if we know the level of an IBGF block, we can obtain its width in all dimensions. The details are as follows.

Lemma 3.2.3: If the level of an IBGF block is L , then the width of the block in dimension $(L-i) \bmod D+1, (i=1, \dots, D)$ is $2^{-((L-i)/D+1)}$ of the universe's width in that dimension.

Proof: Let the level of IBGF block B be L , then the level of its parent division, let it be P , is $L-1$. According to Lemma 3.2.1 and Lemma 3.2.2, the division divides an IBGF block, let it be B' , in dimension $(L-1) \bmod D+1$ and the width of block B in that dimension is $2^{-((L-1)/D+1)}$ of that of the universe. So, the lemma is correct when $i=1$. Now if we remove division P , block B is merged with another block into block B' . Since division P is in dimension $L-1$, the only difference between block B and block B' is their widths in dimension $L-1$. Their widths in all the other dimensions are the same. Thus, we can calculate the widths of block B' in the other dimensions to get those of block B . Using the same method to calculate the width of block B in dimension $(L-1) \bmod D+1$, we can obtain the width of block B' in dimension $(L-2) \bmod D+1$, which is $2^{-((L-2)/D+1)}$ of that of the universe. If we continue to remove the divisions, we will finally get the width of block B in each dimension, which is $2^{-((L-i)/D+1)}$ in dimension $2^{-((L-i)/D+1)}$ ($i=1 \dots D$). Q.E.D.

Until now, we have only proved that given the level of an IBGF block, its size can be calculated. We still need to know how to decide the position of an IBGF block. The following definitions and lemmas give the details.

We define the **id of a node** in the tree corresponding to an IBGF as follows. The root of the tree is given an id 0. Let the depth of a node P be L , and its id be $ID(P)$. Then

the id of its left child, let it be P_l , is $ID(P_l) = ID(P)$, and the id of its right child, let it be P_r , is $ID(P_r) = ID(P) + 2^L$. We use the id of a node to define the **id of the IBGF block** it corresponds to.

The id of an IBGF block at level L determines the position of the block, thus no two different IBGF blocks exist that have the same id. The following lemmas give the details.

Lemma 3.2.4: The id of the root of the tree corresponding to an IBGF is zero. The id of a node, P , at depth L ($L > 0$), can be represented as follows: $ID(P) = \sum_{i=1}^L A_i * 2^{i-1}$, ($A_i = 0$ or 1). If node P or its ancestor at depth i is the left child of its parent, $A_i = 0$, otherwise, $A_i = 1$.

Proof: It can be proved by induction on L . The id of the root is 0 by definition. Let P_l be the root's left child and P_r be its right child; then, the depths of P_l and P_r are all 1. By definition, we have $P_l = 0 = 0 * 2^0$ and $P_r = 1 = 1 * 2^0$. Thus we prove the lemma is correct when $L = 1$.

Assuming the lemma is correct for $L - 1$, we prove it is correct for L . Let the depth of node P be L , and let P' be its parent; then the depth of P' is $L - 1$. So we have $ID(P') = \sum_{i=1}^{L-1} A_i * 2^{i-1}$, ($A_i = 0$ or 1) according to the above assumption. If the ancestor of node P' at depth i is the left child of its parent, $A_i = 0$, otherwise, $A_i = 1$. By definition, if P is the left child of P' , $ID(P) = ID(P')$, and if P is the right child of P' , $ID(P) = ID(P') + 2^{L-1}$. So we have $ID(P) = ID(P') + A_L * 2^{L-1}$, $A_L = 0$ when P is the left

child of P' , and $A_L = 1$ when P is the right child of P' . Thus we get, for any L ($L > 0$),

$$ID(P) = \sum_{i=1}^L A_i * 2^{i-1}, (A_i = 0 \text{ or } 1). \text{ If the ancestor of node } P \text{ at depth } i \text{ is the left child of}$$

its parent, $A_i = 0$, otherwise, $A_i = 1$.

Q.E.D.

Lemma 3.2.5: The id of the block at level zero is zero. For any other block B , where L is

its level, we have the following formula: $ID(B) = \sum_{i=1}^L A_i * 2^{i-1}$, ($A_i = 0$ or 1 and $L > 0$).

If the coordinates of the block or its ancestors at level i are smaller than those of their parent divisions, $A_i = 0$, otherwise, $A_i = 1$.

Proof: By definition, the id of an IBGF block, let it be B , is equal to that of its

corresponding node, let it be P , so $ID(B) = ID(P) = \sum_{i=1}^L A_i * 2^{i-1}$, ($A_i = 0$ or 1 and $L > 0$).

Let B' be the ancestor of B at level i , and let P' be the parent division of B' . Let D be the node corresponding to B' . If the coordinate of B' is smaller than that of P' , and D is its parent's left child, then $A_i = 0$, otherwise D is its parent's right child, and $A_i = 1$.
Q.E.D.

The following lemma gives the formula to calculate the position of an IBGF block, i.e. its lower bound coordinates in each dimension.

Lemma 3.2.6: Let the id of an IBGF block B be $ID(B)$, its level be L , and

$ID(B) = \sum_{i=1}^L A_i * 2^{i-1}$, ($A_i = 0$ or 1 , and $L > 0$). Suppose the width of the whole universe in

dimension d is W_d , then the distance from the block's lower border to that of the universe in dimension d , X_d , can be computed using the formula:

$$X_d = \sum A_{i+1} 2^{-(i/D+1)}, (i = D * k + d - 1, k = 0, 1, \dots, (L-1)/D)$$

Proof: We consider all the ancestor divisions of block B in dimension d . According to Lemma 3.2.1, division at level i is in dimension $(i \bmod D + 1)$, so the divisions in dimension d are at levels $i = D * k + d - 1$, ($i < L$, and k is an integer).

We consider the block's ancestor blocks at level $i+1$ ($i = D * k + d - 1$, $i < L$ and k is an integer), let it be B' . According to Lemma 3.2.2 the width of block B' in dimension d is $2^{-(i/D+1)}$ of that of the universe. According to Lemma 3.2.5, if $A_{i+1} = 0$, then the coordinate of block B' is smaller than that of its parent division, thus the lower bound's coordinate of block B' in dimension d is the same as that of its parent block; otherwise if $A_{i+1} = 1$, it is increased by $2^{-(i/D+1)}$. Thus we know that the distance from the lower bound of a block at level L to that of the universe in the d -th dimension is calculated as follows:

$$X_d = \sum A_{i+1} 2^{-(i/D+1)}, (i = D * k + d - 1, i < L, k > 0, \text{ and } k \text{ is interger}). \quad \text{Q.E.D.}$$

(a) The tree corresponding to the IBGF in Figure 3.2.2. The number beside each node is its id, and the numbers on the right are the depths of the nodes. The letters are used to show the corresponding relations between those nodes and the IBGF blocks and divisions. Brackets mark the leaf nodes.

(b) Numbers are the levels of the IBGF blocks and divisions. Letters show the corresponding relations between the divisions and the nodes.

(c) Numbers are the ids of the IBGF blocks. Letters show the corresponding relations between the blocks and the nodes

Figure 3.2.3 The Tree Corresponding to an IBGF.

3.2.2 Maintenance of an IBGF

In this section, we discuss the maintenance of an IBGF, including creation, insertion and deletion, using our theory about the IBGF. We present the construction of the directory of an IBGF, give the variables used in the algorithms, and explain the various algorithms.

First, we introduce how the directory of an IBGF is constructed.

Directory of an IBGF

The directory of an IBGF is mainly composed of a one-dimensional array, each entry of which stores the information of one IBGF block. By Lemmas 3.2.3 and 3.2.6, we know that from the id of an IBGF block we can find its position, and from the level of an IBGF block we can find its size. So the entry of a block in the directory array stores id and level. It also stores the number of points in the block. There is a one-to-one relation between an IBGF block and a page on a disk, which means that all the points in the block are stored in the page. The entry of a block in the directory array thus stores the address of the page on the disk. In addition, the directory contains the number of pages and the maximum level of IBGF blocks in the IBGF.

During the usage of an IBGF, its directory is always loaded into main memory.

Variables used in Algorithms

Before explaining the IBGF algorithms, we need to define some variables for the sake of convenience. They are as follows:

IBGFDir: a dynamic one-dimensional array holding the information of IBGF blocks in an IBGF directory. Each element has 4 components. They are:

Id: the id of a block;

Level: the level of the block;

Number: the number of points in the block; and

Address: the address of the block's page on the disk;

NoBlock: the number of blocks in an IBGF, also the number of pages in the IBGF;

MaxLevel: the maximum level of blocks in an IBGF;

Corner: a static one-dimensional array of D elements, in which every element, $Corner[i]$, is the coordinate of the universe's lower bound in the $(i + 1)$ -th dimension;

Width: a static one-dimensional array of D elements, in which each element, $Width[i]$, is the universe's width in the $(i + 1)$ -th dimension;

InputBuf: a dynamic one-dimensional array to hold the input points, whose elements are input points;

BlockIdxs: a dynamic one-dimensional array to hold the index of each input point's block in array *IBGFDir*;

PageCapacity: the capacity of a page on a disk;

BlockRegion: a dynamic one-dimensional array holding the position and the size of each block. Each element consists of two arrays of D elements: one is Cn , composed of the coordinates of the block's lower boundary in each dimension, while the other is Wd , composed of the block's width in each dimension.

Creation of an IBGF

We have mentioned that accessing the disk is much more expensive than either accessing main memory or doing calculations in the CPU. So the strategy of our algorithms is to minimize the number of disk accesses. To achieve that, we divide the creation of an IBGF into two steps: The first step is to construct the directory, and the second step is to write the IBGF onto the disk. In all of our algorithms, we assume the input points are already loaded into the *InputBuf* buffer.

Algorithm 3.2.1: Creating an IBGF

Given a set of input points, create an IBGF

1. Construct the directory of the IBGF using `ConstructIBGFDir()`;
2. For each block, assign it to a page in the disk and record the page's address;
3. Write the data onto the disk page by page.

Lemma 3.2.7: In Algorithm 3.2.1, the number of disk accesses is equal to the number of pages in the IBGF.

Proof: In Algorithm 3.2.1, only Step 3 involves disk accesses to write the IBGF onto a disk. Step 1 determines the structure of the IBGF's directory, and Step 2 assigns a page to each block and gives an address to each page. Both these steps are done in main memory. In Step 3, the IBGF is written onto a disk page by page, so the total number of disk accesses is equal to the number of pages in the IBGF.

The following algorithm gives the details of constructing the directory of an IBGF.

Algorithm 3.2.2: ConstructIBGFDir()

1. For each input point
2. Calculate its block using CalculateIBGFBlock();
3. Record the index of that block in the *BlockIdxs* buffer;
4. Increase the number of points for that block in *IBGFDir*;
5. If the block is overloaded
6. Split the block using SplitIBGFBlock().

The above algorithm goes through each input point, calculates its block, increases the number of points for that block, and splits the block if it is overloaded.

The following algorithm explains how to calculate the block of an input point. The maximum level of blocks in an IBGF is used. We assume that the block of the point could be at any level from 0 to the maximum, and we calculate its ids for different levels. Then we compare them with the ids of existing blocks, and decide which IBGF block is the point's block.

Algorithm 3.2.3: CalculateIBGFBlock()

Given a point $P(X_1, X_2, \dots, X_D)$, calculate its IBGF block. Let $id[]$ be an array with $MaxLevel + 1$ entries, let $x[]$ be an array with D entries.

1. $id[0] = 0$
2. $x[0] = X_1 - Corner[0], x[1] = X_2 - Corner[1], \dots, x[D-1] = X_D - Corner[D-1]$
3. for ($i = 0 \dots MaxLevel - 1$)
4. $m = i \bmod D + 1$

```

5.       $k = 2^{-(i/D+1)} * Width[m-1]$ 
6.      if ( $x[m-1] > k$ ) then
7.           $id[i+1] = id[i] + 2^i$ 
8.           $x[m-1] = x[m-1] - k$ 
9.      else  $id[i+1] = id[i]$ 
10.   for ( $i = MaxLevel .. 0$ )
11.       for ( $j = 0 .. NoBlock - 1$ )
12.           if ( $id[i] = IBGFDir[j].Id$ )
13.               return  $j$ 

```

The above algorithm finds the block of a point. Steps 1 to 9 assume the point's block is at any level from 0 to the maximum, and calculate the id of each block at each level. Array x is used to store the distances from the point to its block's lower bounds in different dimensions, and its element $x[i]$ stores the distance in the $(i + 1)$ -th dimension. $x[i]$ changes as the level of the block under consideration increases. Array x is initialized in Step 2, where $x[i]$ is equal to the distance from the point to the universe's lower bound in dimension $(i + 1)$. Array id is used to store the ids of the blocks, and element $id[i]$ stores the id of the block at level i . Since the block of the point at level zero is the universe, zero is assigned to $id[0]$ in Step 1. Steps 3 to 9 simulate the process of adding divisions at levels 0 to $MaxLevel - 1$. Step 4 calculates the dimension of each division, and assigns the result to m . Step 5 calculates the width of the division's child blocks in dimension m . Step 6 decides whether the coordinate of the point in dimension m is larger

than that of the division. If it is, Steps 7 and 8 are executed to calculate the id of a higher level block and to change $x[m-1]$. Otherwise, Step 9 is executed to calculate the id of a higher level block. Steps 10 to 13 go from the highest possible id to the lowest possible id of the blocks, search through the directory to find a block with that id, and thus find the block of the point in the IBGF.

Before we give the algorithm to split a block, we need to know how to calculate the region of a block. The following algorithm gives the details.

Algorithm 3.2.4: CalculateIBGFRegion()

Given an IBGF block, B , whose level is L , and id is ID , calculate its region. Let x and w be two arrays, each having D entries. Array x is used to store the position of block B , and array w is used to store the size of block B .

1. For $i = 1..D$ // Steps 1 to 4 calculate the size of the block by Lemma 3.2.3
2. $k = L - i$
3. $m = k \bmod D + 1$
4. $w[m-1] = Width[m-1] * 2^{-(k/D+1)}$
5. For $i = 0..D-1$, $x[i] = Corner[i]$ // Initialize array x
6. While ($L > 0$) // Steps 6 to 11 backtrack the process to create block B , and calculate the distance from the lower bound of block B to that of its ancestor block at level $L - 1$. L decreases by one in each iteration, and the iteration stops when $L = 0$. Lemma 3.2.6 is used.
7. $m = (L - 1) \bmod D + 1$ // Calculate the dimension of the division at level $L - 1$.

8. If ($ID > 2^{L-1}$) // The ancestor block at level L is the right child of its parent.
9. $x[m-1] = x[m-1] + 2^{-((L-1)/D+1)} * Width[m-1]$ // Update the distance in dimension m .
10. $ID = ID - 2^{L-1}$ // Calculate the ancestor block's id.
11. $L = L - 1$ // Decrease the level.

The above algorithm uses Lemmas 3.2.3 and 3.2.6 to calculate the position and the size of an IBGF block by its id and its level.

Whenever a block is overloaded, it is divided into two, and all the points in this block are redistributed into two newly created blocks. The following recursive algorithm shows how to split a block.

Algorithm 3.2.5: SplitIBGFBlock()

Given a block whose level is L , and id is ID , split the block.

1. Create a new entry of block in *IBGFD*, and assign $L + 1$ to its level and $ID + 2^L$ to its id;
2. Increase the level of the original block by 1;
3. Increase the maximum level *MaxLevel* if necessary;
4. Increase the number of the blocks *NoBlock*;
5. Calculate the regions of the two resulting blocks;
6. For each point falling in the original block
7. Check which resulting block is its block, and update the number of points in that block;

8. If any resulting block is still full, split it by calling `SplitIBGFBlock()`.

With all of the previous algorithms, the creation of an IBGF can be implemented. Some of the algorithms are also used in the following operations.

Insertion of data into an IBGF

The operation is to insert data into an existing IBGF, which is unavoidable and important in practice. As with the creation of an IBGF, the strategy in the insertion is to reduce the number of disk accesses.

Algorithm 3.2.6: Inserting Points into an IBGF

1. For each input point
2. Calculate its block by calling `CalculateIBGFBlock()`;
3. Mark the block and record the number of points inserted into it.
4. For each block where there are points inserted
5. Read the page from the disk into a buffer *TempBuf*;
6. Append all the points inserted in this block into *TempBuf*;
7. If the number of points in the block is less than *PageCapacity*
8. Write the buffer *TempBuf* back into the original page;
9. else split the block using `SplitIBGFBlock( )`;
10. For the original block and each newly created block
11. Go through *TempBuf* and save the points in it into another buffer *Buf2*;
12. Write *Buf2* to the block's page in the disk.

Lemma 3.2.8: Algorithm 3.2.6 involves one disk read for each affected IBGF page and one disk write for each changed and newly created IBGF page.

Proof: In Algorithm 3.2.6, Step 5 reads all the pages where there are new points inserted. Step 7 writes back the non-overloaded pages, and Step 11 writes back the overloaded pages and the newly created pages. There is no other disk access. Q.E.D.

Deletion of data from an IBGF

With deletion, the load of some blocks may decrease and block merging may be necessary to raise the disk utilization of an IBGF. We use a **70% threshold** to decide when to merge blocks. That means we merge two blocks into one only if the number of points in each block after deletion is less than 70% of a page's capacity. This restriction helps avoid the repeated merges and splits of pages that occur when there are frequent updates in an IBGF.

Due to the special process used to generate an IBGF, not all adjacent grid blocks can be merged together. We use the term **buddy blocks** to refer to blocks which can be merged together. When a block is divided into two blocks in an IBGF, the resulting blocks are automatically **buddy blocks** of each other. In other words, sibling blocks represent buddy blocks. The following algorithm calculates a block's buddy block. It comes from the definition of the buddy block.

Algorithm 3.2.7: CalculateIBGFBuddyBlock()

Given a block in *IBGFDir*, whose level and id are L and ID, respectively, calculate its buddy block's index in *IBGFDir*.

1. If $ID \geq 2^{L-1}$ // Check if the given block is its parent's right child.
2. $BID = ID - 2^{L-1}$ // If yes, its buddy block is the parent's left child.
3. else $BID = ID + 2^{L-1}$ // Otherwise, its buddy block is the parent's right child.
4. For each entry i in $IBGFD\textit{ir}$
5. If $IBGFD\textit{ir}[i].Id = BID$ and $IBGFD\textit{ir}[i].Level = L$
6. return i
7. else there is no buddy block found.

The above algorithm applies the definitions of a block's id and the buddy block.

Since buddy blocks are in fact sibling blocks, if a block is its parent's left child, its buddy block must be its parent's right child, otherwise its buddy block is its parent's left child. Thus Steps 1 to 3 calculate the id of the buddy block. The level of the buddy block should be the same as that of the given block. Steps 4 to 7 go through each entry in $IBGFD\textit{ir}$ to look for the entry of the buddy block.

We have given the method to calculate the buddy block. Now we are ready to present the algorithm to delete points from an IBGF.

Algorithm 3.2.8: Deleting Points from an IBGF

1. For each input point
2. Calculate its block using $\text{CalculateIBGFBlock}()$;
3. Mark the block and record the number of points deleted from it;
4. For each marked block
5. Load the page from the disk into a buffer $\textit{TempBuf}$;
6. Delete any input point in this block from $\textit{TempBuf}$;

7. Write back *TempBuf* into the original page;
8. If the number of points in *TempBuf* is more than $70\% * PageCapacity$
9. Continue;
10. Calculate the block's buddy block using *CalculateIBGFBuddyBlock()*;
11. Unmark the buddy block so that it is not re-processed;
12. Load the page of the buddy block from disk into a buffer *TmpBuf2*;
13. Delete any input points in the buddy block from *TmpBuf2*;
14. If the number of points in *TmpBuf2* is more than $70\% * Page Capacity$ or the total number of points in *TmpBuf2* and *Tempbuf* is greater than *PageCapacity*
15. Write *TmpBuf2* back to the buddy block's page;
16. Continue;
17. Append *TmpBuf2* to the end of the original block's page;
18. Release the entry of the buddy block in *IBGFDir*;
19. Decrease the level of the original block and change its number of points;
20. Go to step 10.

Lemma 3.2.9: Algorithm 3.2.8 reads each affected block and buddy block from the disk once and writes the non-merged buddy block, the original block, or the merged block back to the disk once.

Proof: In Algorithm 3.2.8, Steps 5 and 12 involve reading blocks from the disk. In Step 5, the affected blocks are loaded. In Step 12 the buddy blocks are loaded. If a buddy block is also a block with points to be deleted, it is unmarked in Step 11 to avoid being

read again in Step 5. Steps 7, 15 and 17 involve writing blocks onto the disk. In Step 7, an affected block is written back to its page after the input points are removed from it. In Step 15, a buddy block is written back to its page since the merge condition is not satisfied. The block is unmarked such that it is not processed later. In Step 17, one of the buddy blocks is merged into another, and written to its page. All other steps do not involve disk accesses. Q.E.D.

We've discussed how to maintain an IBGF. In the next section, we discuss running queries on it.

3.2.3 Queries on an IBGF

In this section, we choose three popular multi-dimensional data queries, the point query, the range query, and the nearest neighbor query, and explain how to run them on an IBGF. As our work focuses on points, the following discussion only considers points, i.e. objects without extensions.

A point query is used to find the point overlapping a given point. For example, we are given the address of a shopping center, and we are required to locate it on a map. A range query finds points intersecting a given query window. For example, we are given an area in a city, and we are required to find all the shopping centers in this area. A nearest neighbor query finds the nearest point to a given point. For example, we are given the address of a shopping center, and we are required to find the intersection nearest to it. The formal definitions of these queries are given as follows:

Definition: Point Query

Given a point $P(X_1, X_2, \dots, X_D)$, find every point $Q(Y_1, Y_2, \dots, Y_D)$ in the universe, such that $(X_1 = Y_1)$ and $(X_2 = Y_2)$ and ... and $(X_D = Y_D)$

Definition: Window Query (or Range Query)

Given a D -dimensional range $[L_1, U_1] * [L_2, U_2] * \dots * [L_D, U_D]$, find every point $P(X_1, X_2, \dots, X_D)$ in the universe such that

$$(L_1 \leq X_1 \leq U_1) \text{ and } (L_2 \leq X_2 \leq U_2) \dots \text{ and } (L_D \leq X_D \leq U_D)$$

Definition: Nearest-Neighbor Query

Given a point P , find every point Q in the universe, such that for any other point O in the universe the following holds: $|PQ| \leq |PO|$, ($|AB|$ refers to the distance from point A to point B. The Euclidean distance is used here.)

The algorithms corresponding to these queries on an IBGF are now introduced.

Point Query on an IBGF

We introduce two methods of implementing point queries. One method calculates the point's block using its id and its level. This method is the same as the point query algorithm suggested by M. Ouksel [3]. This method is not efficient. Let the maximum level of blocks are ML , it needs to search through the directory of the IBGF ML times in the worst case. We suggest a more efficient method, which search through the information of each block in the main memory at most once. In both methods, only one disk access is necessary for a point query.

Algorithm 3.2.9: Point Query on an IBGF – Method 1

1. Calculate the block of the given point using CalculateIBGFBlock();
2. Load the page of the block into a buffer;
3. Search the buffer to find the points overlapping the input point.

Lemma 3.2.10: Algorithm 3.2.9 reads from the disk once to achieve a point query.

Proof: Only Step 2 involves one disk access, which reads one page from the disk.

Q.E.D.

To implement the second method, the region of each block must be calculated in advance, which is implemented in Algorithm 3.2.4.

Algorithm 3.2.10: Point Query on an IBGF – Method 2

Given a point $P(X_1, X_2, \dots, X_D)$, find the point Q overlapping P .

1. For $i = 0..NoBlock - 1$
2. For $j = 0..D - 1$
3. If $BlockRegion[i].Cn[j] > X_{j+1}$
4. Goto Step 1
5. For $j = 0..D - 1$
6. If $BlockRegion[i].Cn[j] + BlockRegion[i].Wd[j] \leq X_{j+1}$
7. Goto Step 1
8. Load the points in the block from the disk into a buffer $TmpBuf$;
9. Search $TmpBuf$ for any point overlaying the given point, and output it;
10. Break.

The above algorithm implements a point query in an IBGF. If a block overlaps the query point P , then its lower bound coordinates in all dimensions should be smaller than or equal to that of the point, and its upper bound coordinates in all dimensions should be larger than those of the point. Steps 2 to 7 exclude the blocks that do not meet the above conditions. Step 8 loads the block overlapping the query point P into a buffer. Then the buffer is searched for points overlapping P . Due to the fact that the leaf blocks in an

IBGF do not intersect with each other, there is not any other block overlapping point P , so the algorithm stops after the block overlapping the given point is processed.

Lemma 3.2.11: Algorithm 3.2.10 accesses the disk once to achieve a point query

Proof: Only Step 6 accesses the disk to read the block overlapping the given point into a buffer. All the other steps do not involve any disk access. Q.E.D.

Range Query on an IBGF

There are two ways to implement a range query. One is to calculate the ids and the levels of IBGF blocks intersecting a given range, which is suggested by M. Ouksel [3]. His method is very complicated and not efficient. Another method is to simply search all the blocks to find those intersecting the query range. For this method, the region of each IBGF block needs to be calculated in advance.

M. Ouksel's method is as follows. To calculate the ids and the levels of the blocks intersecting a range, we create several points in the range in the following way. We put several divisions in each dimension such that the range is divided into grid cells and the size of each cell is equal to that of the smallest IBGF block. The sizes of the cells near the upper bound of the range area are allowed to be smaller than that of the smallest IBGF block. We create a point at each intersection of those divisions. Since each grid cell is equal to or smaller than the size of the smallest IBGF block, every block intersecting the range must overlap at least one intersection of the divisions. Thus by calculating the block of those points, we can find all the IBGF blocks intersecting the range. Figure 3.2.4 shows an example.

Figure 3.2.4 Transferring a Range Query into Multiple Point Queries

Although the above method can be used to solve a range query in an IBGF, its performance is somewhat poor due to its complexity. We adopt the following simple algorithm to implement a range query.

Algorithm 3.2.11: Range Query on an IBGF

Given a D -dimensional range $[L_1, U_1] * [L_2, U_2] * \dots * [L_D, U_D]$, find all the points enclosed by the range.

1. For $i = 0..NoBlock - 1$
2. For $j = 0..D - 1$
3. If $BlockRegion[i].Cn[j] > U_{j+1}$
4. Goto Step 1
5. For $j = 0..D - 1$
6. If $BlockRegion[i].Cn[j] + BlockRegion[i].Wd[j] \leq L_{j+1}$
7. Goto Step 1
8. Load the block from the disk into buffer *TempBuf*;
9. $Inside = True$ // Determine if the block is enclosed in the given range.
10. For $j = 0..D - 1$

11.	If $BlockRegion[i].Cn[j] < L_{j+1}$ or $BlockRegion[i].Wd[j] +$
$BlockRegion[i].Cn[j] > U_{j+1}$
12.	$Inside = False$
13.	If $Inside = True$
14.	Output all the points in buffer $TempBuf$.
15.	Else for each point $P(X_1, X_2, \dots, X_D)$ in the block
16.	If the point is located in the range, output the point.

The above algorithm implements a range query on an IBGF by going through each IBGF block and checking whether it intersects with the given range. Steps 2 to 7 exclude all the blocks not intersecting the query range. Step 8 loads the block intersecting the query range from the disk. Steps 9 to 14 output all the points in a block enclosed by the range. Steps 15 and 16 search the blocks intersecting the range but not enclosed by it and output the points enclosed by the range. Since more than one block might intersect a given range, the process searches through all the IBGF blocks.

Lemma 3.2.12: Suppose there are K blocks intersecting the query range. Algorithm 3.2.11 reads the disk K times.

Proof: Only Step 6 accesses the disk to load the block intersecting the given range and it is executed once for each of these blocks. Q.E.D.

Nearest Neighbor Query on an IBGF

The following algorithm gives a method to find the nearest neighbor of a query point in an IBGF by converting the nearest neighbor query to a group of range queries.

Algorithm 3.2.12: Nearest Neighbor Query on an IBGF.

Given a query point $P(X_1, X_2, \dots, X_D)$, find its nearest neighbor in an IBGF. Let L be $1/20$ of the largest one among the universe's widths in all dimensions.

1. $l = L$;
2. $L_1 = X_1 - L, U_1 = X_1 + L, L_2 = X_2 - L, U_2 = X_2 + L, \dots, L_D = X_D - L, U_D = X_D + L$;
3. Run a range query on the IBGF with the range $R: [L_1, U_1] \times [L_2, U_2] \times \dots \times [L_D, U_D]$, and mark each loaded block such that it is not reloaded later;
4. Among the retrieved points, find the point Q nearest to query point P ;
5. Let D be the Euclidean distance between P and Q ;
6. If $D \leq L$, then Q is the nearest neighbor of P , stop.
7. $L = L + l$, go to step 2.

Lemma 3.2.13: Algorithm 3.2.12 implements a nearest neighbor query on an IBGF.

Proof: First we prove there is one point found. Unless the IBGF is empty, there is at least one point in the IBGF found in Step 3, as the query range R becomes larger and larger until the whole universe is covered.

Then we show that the point we found is the nearest neighbor of the query point P . Step 4 finds point Q , which is the nearest to the query point P in the query range R . If point Q meets the condition of Step 5, the distance from P to Q is less than the distance from P to the bound of the query range in any dimension. Suppose there is another point Q' which is nearer to P than Q , then $|Q'P| < |QP|$. We draw a circle with P as the center

and $|Q'P|$ as the radius. The circle must be inside the query range R and Q must be out of the circle. This contradicts the fact that Q is the nearest neighbor of P in range R . Thus Q is proved to be the nearest neighbor of P in the IBGF. Q.E.D.

Figure 3.2.5 illustrates the proof of Lemma 3.2.13. Figure 3.2.6 demonstrates Algorithm 3.2.12. P is the query point, and $P4$ is the nearest neighbor of P .

Figure 3.2.5 Proof of the Nearest Neighbor Query Algorithm

Figure 3.2.6 The Nearest Neighbor Query Algorithm

Lemma 3.2.14: The number of disk accesses in Algorithm 3.2.12 is one in the best case and N in the worst case.

Proof: We prove the lemma by giving examples for both the best and the worst cases. Step 3 makes sure that a page will not be loaded more than once. The following figure shows the best case and worst case examples. Point P is the query point, and point Q is its nearest neighbor. Figure 3.2.7(a) is an example of the best case, in which point Q is in the same block as point P , so only one disk access is needed to find it. Figure 3.2.7(b) is an example of the worst case, in which point Q is in a block adjacent to the block of point P . As the query range enlarges, all the blocks have been retrieved when point Q is finally found. Q.E.D.

Figure 3.2.7 The Best Case and the Worst Case of Nearest Neighbor Query

We've introduced the concept of an IBGF, explained the algorithms to create and update it, and presented three queries that can be run on an IBGF in this section. In the next section, we discuss the implementation of parallel IBGFs.

3.3 Parallel IBGFs

To parallelize an IBGF is to allocate an IBGF from a single disk to multiple disks. The purpose is to reduce the time wasted due to disk accesses. Simply dividing the file into several parts and distributing each part onto a disk does not necessarily achieve the objective. To reduce the disk access time, we have to make sure that the data accessed simultaneously are distributed evenly on different disks.

There are two types of methods to allocate files onto several disks. One is the index-based method, which is mainly designed for Cartesian Product Files and is not well-suited to IBGFs, especially when the data are not uniformly distributed. The other is the proximity-based method, which makes use of the proximity between blocks and transforms a disk allocation problem into a graph partition one.

Bongki et al. propose a proximity-based method, the mini-max spanning tree method, to parallelize grid files [4]. The mini-max spanning tree method works well for non-uniformly distributed data but not for uniformly distributed data. We introduce an algorithm which improves on theirs and performs well in all distributions of data.

To be able to evolve dynamically is an important factor for IBGFs. Our work also includes the algorithms needed to maintain the dynamic parallel IBGFs developed from the proximity-based idea.

3.3.1 Creation of a Mapping Graph

The first step in a proximity-based method is to create a complete weighted graph. A node is created for each block in the IBGF. An edge is created for every two blocks, and the proximity between those blocks is assigned as the weight of the edge. We call such a graph the **mapping graph for an IBGF**.

Ibrahim Karna et al. propose the proximity of two rectangles R and S as the proportion of queries that involve both rectangles [5]. First they worked on one-dimensional range queries. They converted a segment to a two-dimensional point using its start as the x coordinate and its end as the y coordinate. Thus, both the query range and the two segments R and S become two-dimensional points. They considered the two situations, where R and S intersect and where they are disjoint. For both situations, they calculated the proportion of queries that retrieve R and S together. Then they generalized the one-dimensional range queries to multi-dimensional ones by getting the projections of R and S in each dimension, calculating the proportion of queries that retrieve both projections in each dimension, and multiply the proportions in all dimensions. Please refer to [5] for more details. Bongki Moon et al extended the concept to the proximity of two grid blocks [4]. We extend the concept further to the proximity of two IBGF blocks as follows.

Definition: The **proximity of two IBGF blocks** is the proportion of queries that retrieve both blocks. The proximity of two IBGF blocks R and S can be calculated as follows:

$$\text{Proximity}(R, S) = \prod_{i=1}^D \text{Proximity}(R_i, S_i)$$

$$\text{Proximity}(R_i, S_i) = \begin{cases} (1 + 2 * U_i / L_i) / 3 & \text{if } (R_i \text{ intersects } S_i) \\ ((1 - V_i / L_i)^2) / 3 & \text{if } (R_i \text{ and } S_i \text{ are disjoint}) \end{cases}$$

R_i is the projection of R along the i -th dimension axis, and S_i the projection of S . L_i is the width of the universe in the i -th dimension. U_i is the intersection's length if R_i and S_i intersect, and V_i is the distance between them if they are disjoint.

Figure 3.3.1 shows how to calculate the proximity between IBGF blocks R and S .

The following algorithm describes how to create the mapping graph for an IBGF.

Algorithm 3.3.1: Creating the Mapping Graph for an IBGF

1. Create one node for each block in an IBGF;
2. Create one edge for each pair of nodes;
3. Calculate the region of each block;
4. Calculate the proximity of each pair of blocks, and assign it as the weight of the corresponding edge in the graph.

Figure 3.3.1 How to Calculate the Proximity of Two IBGF Blocks

3.3.2 Static Parallel IBGFs

By static parallel IBGFs, we mean that the data in the IBGFs are loaded only during the creation of the files and are not frequently updated. In the next section, we discuss the algorithms used to maintain dynamic parallel IBGFs. Whether the IBGFs are static or dynamic, their creation can use the same algorithms. The construction of the

directory in the parallel IBGFs is the same as with the single disk IBGF. The main concern of parallelization is how to assign the IBGF blocks onto different disks.

With the mapping graph created, the problem of assigning disks is now converted into a graph partition problem. We need to give more definitions to facilitate our explanation. If both nodes of an edge, let it be E , are in one partition, let it be P , we call edge E an **edge inside partition P** . We define the sum of the weights of all the edges inside partition P as **the weight of partition P** . Thus our graph partitioning problem is to divide a complete weighted graph into several partitions such that the total of the weights of all the partitions is minimized. This is an NP-complete problem. Bongki Moon et al. propose an approximate solution [4]. We noticed some shortcomings in their algorithms and make some improvements. Following are the details of our approach.

The partitioning problem is solved in two steps. The first step selects a seed for each partition, and the next step expands each partition. For the first step, Moon's algorithm randomly selects N distinct nodes from the complete graph as the seeds for the N partitions. In randomly selecting the seeds, their results become unpredictable, and their method does not consider the weights between the seeds. We take the weight into consideration and try to make the weights between seeds as large as possible. Our improved algorithm is as follows.

Algorithm 3.3.2: Selecting Each Partition's Seed

1. Randomly select a node V_1 , and let it be the seed of the first partition;
2. While the number of seeds is less than M
3. For each unselected node

4.	Calculate the total weights of the edges from it to every selected node;
5.	Find the largest total weight, and let the first node with the weight be the next partition's seed.

We use a complete graph with eight nodes in Figure 3.3.2 to demonstrate the difference between Moon's partition method and ours. Each node is given a name. To make it easier to understand, we assign the length of each edge to its weight. We want to get three partitions from the eight points. In Figure 3.3.2(a), we use Algorithm 3.3.2 to find three seeds for the partitions. At first, we randomly select the seed of the first partition, which happens to be node *A* in our example. Then in Step 5, we select node *E*, which is farthest from node *A*, as the seed of the second partition. Then in Step 4, we sum the weights from unselected nodes to all the seeds, and in Step 5, point *C* is selected as the seed of the third partition. In Figure 3.3.2(b), two sub-figures show two possible collections of seeds randomly selected by Bongki Moon et al.'s method. Obviously, the right one is not good. With the method described above, we increase the weights among the seeds such that the minimum total weight of partitions is further improved in the first step.

(a) The seeds of partitions chosen in our method

(b) The possible cases of randomly selected seeds of partitions

Figure 3.3.2 Improved Seeding Method

The next step is to expand the partitions. To explain the methods, we give one more definition concerning the weight. We consider a node, let it be A , and a partition, let it be P , such that A is outside partition P . For all the edges from A to any nodes in partition P , we compare their weights, and we define the largest one as the **weight of node A to partition P** , and denote it by $PW(AP)$.

Bongki Moon et al.'s algorithm goes through all the partitions circularly. For a partition P , it selects node A , such that for any other node B , $PW(AP) \leq PW(BP)$. Then it appends node A to partition P , and updates the weight of all the unselected nodes to partition P . The algorithm ends when all the nodes are assigned to different partitions. Our improvement, however, finds unselected node A and partition P , such that for any unselected node B and any partition Q , $PW(AP) \geq PW(BQ)$. Then, to avoid A being assigned to P , our method assigns A to another partition R , such that $PW(AR) \leq PW(AQ)$. Following is the details of our improved expansion algorithm.

Algorithm 3.3.3: Expanding Partitions in the Mapping Graph for an IBGF

1. Go through the weight from each unselected node to each partition, and find the largest one;
2. For each node whose weight to a partition is equal to the largest
3. Assign it to a partition, A , such that the weight of the node to A is the least;
4. Recalculate the weights of all unselected nodes to partition A ;
5. If all the nodes are now partitioned, stop.
6. Else go to Step 1.

In the above algorithm, Steps 1 to 6 are iterated until all the nodes are assigned to different partitions. Step 1 finds the largest of the weights of all the unselected nodes to each partition. Steps 2 to 4 is a **for** loop that processes each node whose maximum weight to a partition is equal to the largest one. In Step 3, the node is assigned to a partition to which it has the smallest weight. In Step 4, the weight from unselected nodes to this partition is updated. Steps 5 and 6 check whether all the nodes are assigned; if so, the algorithm stops, otherwise Step 1 is executed again. Figure 3.3.3 explains the above algorithm using the example shown in Figure 3.3.2(a). In the first iteration of Steps 1 to 6, node *G* is processed because its weight to partition 3 is the largest. We assign node *G* to partition 1 because its weights to partitions 1 and 2 are equal. In the second iteration of Steps 1 to 6, nodes *B*, *D*, *F*, and *H* are processed, since their weights to partitions are the same. We begin with node *B*, whose weight to partition 3 is the least, and we assign it to partition 3 in Step 4. Then in Step 5, the weights from nodes *D* and *F* to partition 3 are increased. The next time we execute Steps 3 to 5, and process node *D*. The weight of *D* to partition 2 is the least, so *D* is assigned to partition 2. Thus the weights of *F* and *H* to partition 2 are increased. In the following 2 iterations of Steps 3 to 5, node *F* is assigned to partition 2, and node *H* is assigned to partition 1.

Figure 3.3.4 and Figure 3.3.5 shows the expansion method by Bongki Moon et al. on the two examples shown in figure 3.3.2(b). Each partition is processed circularly to find the node whose weight to the partition is the least among the possible weights, and the partition is expanded to include the node. From Figures 3.3.2, 3.3.3, and 3.3.4, we can see the advantages of the improved expanding method. The improved method is

predictable and gets a better partition result. The original method is unpredictable and gets a worse partition result. Furthermore, our method proves to be faster and more efficient than original method.

(a) The weight of node G to partition 3 is the largest one among all the weights of unselected nodes to all partitions. Node G is assigned to partition 1 to which the weight of G is the smallest

(b) The weights of nodes B , D , F , and H to all partitions are equal and also the largest. We assign node B to partition 3 first, which causes the weights of other nodes to partition 3 to change. Then D is assigned to partition 2 because its weight to partition 3 has increased. F is assigned to partition 2, and H is assigned to partition 1.

Figure 3.3.3 Improved Expanding Method

(a) Each partition is assigned a node whose weight to the partition is the least. The weights of unselected nodes to other partitions are updated.

(b) Each partition is assigned nodes whose weights to the partition is the least

Figure 3.3.4 The First Example of Bonki et al.'s Partition Expanding Method

(a) Each partition is assigned a node whose weight to the partition is the least. The weights from the unselected nodes to all the partitions are updated.

(b) Each partition is assigned the node whose weight to the partition is the least

Figure 3.3.5 The Second Example of Bonki et al.'s Partition Expanding Method

3.3.3 Dynamic Parallel IBGFs

The existing parallel methods seldom consider the dynamic characteristics of a grid file, which is very important in practice. We propose a practical method to support the dynamic feature and at the same time minimize the number of disk reorganizations. The idea is based on the proximity-based parallelizing algorithm proposed by Bongki Moon et al. [4].

Insertion of data into parallel IBGFs

When inserting data into parallel IBGFs, there are two possible situations depending on whether or not there are over-loaded blocks. If there are no over-loaded block, we simply insert data into the pages of the blocks in which they are located. The other situation is much more complicated, since the over-loaded blocks must be split, which brings up the problem of how to assign these newly created pages to different disks. We solve the problem in the following way. When a new IBGF block is created, a new node is created in the mapping graph for the IBGF, and the edges from the node to all the other nodes are created, including their weights, which, as we recall, are determined by the proximity between blocks. Then the weights of this node to all the partitions are calculated, and this node is assigned to the partition with the minimum weight. The following algorithm shows all the required steps.

Algorithm 3.3.4: Inserting an IBGF Block into Parallel IBGFs

1. Update the mapping graph by adding a node, adding edges from this node to all the existing nodes, and calculating the weights of the edges;
2. Calculate the weight from the node to all partitions;

3. Find the partition to which the weight from the node is the least;
4. Assign the node to the found partition.

The following figure illustrates what happens when a new node is inserted into a partitioned graph.

A node, I , is inserted into the former example in Figure 3.3.3. We add edges from I to all the existing nodes, and calculate their weights and the weights from I to all the partitions. We assign node I to partition 2, to which the weight from I is the least among all the weights from I to the other partitions.

Figure 3.3.6 Dynamic Insertion of Data into Parallel IBGFs

Deletion of data from parallel IBGFs

Analogous to insertion, deletion has two scenarios depending on whether pages need to be merged or not. If no merge is necessary, we simply delete that data from the corresponding page. Otherwise, it is more complicated. Strictly speaking, it should cause the reorganization of the whole grid file because in its mapping graph, two nodes are combined into one node, and the weights of all the edges adjacent to them have to be changed, thus changing the partitioning of the whole graph. To make things easier, we only combine the data in the merged blocks, transfer them from one disk onto the other, and delete the first page from its disk. In this way, the performance of queries on parallel IBGFs might deteriorate, but the reorganization of entire files is avoided so that the performances of update and query are balanced. To improve the performance of queries on dynamic parallel IBGFs, a periodic reorganization of the files is recommended.

We have discussed the parallelization of IBGF in this section. In the next section, we introduce another multi-dimensional access method, the MFP Tree, which is primarily designed for parallel environments.

3.4 MFP Tree

The MFP Tree was invented by Doron Nussbaum [2]. It is especially designed for a parallel environment. In this section, we describe in detail how to maintain an MFP Tree, and at the same time compare it to the IBGF.

3.4.1 Concepts

Before discussing the maintenance of the MFP Tree, we describe what an MFP Tree is and explain the concepts behind it. As with the creation of an IBGF, the creation of an MFP Tree is also a space-driven process. Contrary to an IBGF, where an overloaded block is divided into two equal-sized blocks by a division put in different dimensions circularly, in an MFP Tree, an overloaded block is divided into equal-sized blocks by a fixed partitioning scheme. In an IBGF, the universe is not divided until it is overloaded, while in an MFP Tree, the universe is divided at the very beginning.

We call the block in an MFP Tree an **MFP block**. We call the partitioning scheme “Multidimensional Fixed Partitions,” which is abbreviated as “MFP.” A **Multidimensional Fixed Partition** of an MFP Tree is composed of a certain number of partitions in each dimension, which evenly divide the projection of an MFP block in each dimension into $P_1, P_2, \dots, \text{ and } P_D$ segments, and thus partition the block into $\prod_{i=1}^D P_i$ equal-sized sub-blocks. We abbreviate the Multidimensional Fixed Partition as a “ $\prod_{i=1}^D P_i$ MFP”.

An MFP A is said to be larger than another MFP B , when $\prod_{i=1}^{D^A} P_i^A \geq \prod_{i=1}^{D^B} P_i^B$.

We take the example of Section 3.2.1 to illustrate the partitions of an MFP Tree after each point has been inserted. Figure 3.4.1 shows a 2×2 MFP Tree, and Figure 3.4.2 shows a 2×4 MFP Tree. The capacity of a page is set at 2 points. According to those 2 figures and Figure 3.2.2, which illustrates the partition of an IBGF using the same example, we have the following observations.

Observation 3.4.1: Given the same input point set, there are fewer overloaded blocks in an MFP Tree than in an IBGF.

Observation 3.4.2: Given the same input point set, there are more blocks in an MFP Tree than in an IBGF.

Observation 3.4.3: Given the same input point set, a larger MFP in an MFP Tree involves fewer overloaded blocks and results in more blocks in the end.

(1) After P8 is inserted, the first overloaded block is partitioned.

(2) After P11 is inserted, the third overloaded block is partitioned

(3)The Final partition of a 2×2 MFP Tree.

Figure 3.4.1 Create a 2×2 MFP Tree

Figure 3.4.2 Create a 2 x 4 MFP Tree

One reason for the name “MFP Tree” is that it refers to the “Multidimensional Fixed Partitioning” scheme (MFP). Another reason is that it can be represented by a tree structure. Before we introduce the tree corresponding to an MFP Tree, we need to explain the concept of “the id of a block.” When an MFP block is divided by a $\prod_{i=1}^D P_i$ MFP,

$\prod_{i=1}^D P_i$ sub-blocks are created. Their projections in the i -th dimension are P_i continuous intervals, and we define the **id of a projection** as follows. Each projection is given an id from 0 to $(P_i - 1)$, 0 to the projection nearest to the lower bound, and $(P_i - 1)$ to the

projection nearest to the upper bound. Then, a sub-block can be identified by a group of integers $(A_1, A_2, \dots, A_i, \dots, A_D)$, where A_i is the id of the block's projection in the i -th dimension. The **id of the MFP sub-block**, let it be $ID(B)$, is calculated as follows.

$$ID(B) = \sum_{i=1}^D A_i \prod_{j=0}^{i-1} P_j \quad (P_0 = 1, P_j > 0, \text{ and } 0 \leq A_i \leq P_i - 1)$$

Lemma 3.4.1: The ids of MFP sub-blocks form continuous integers from 0 to $\prod_{i=1}^D P_i - 1$

Proof: First we know that there are $\prod_{i=1}^D P_i$ integers altogether, because an MFP block is always divided into $\prod_{i=1}^D P_i$ sub-blocks. Since $0 \leq A_i \leq P_i - 1$ and $P_j > 0$, the smallest block id is 0, when $A_i = 0$ ($i = 1..D$), and the largest block id is $\prod_{i=1}^D P_i - 1$, when $A_i = P_i - 1$ ($i = 1..D$).

Now, we prove that there cannot be two sub-blocks with the same the same id.

For any two different sub-blocks A and B , $ID(A) = \sum_{i=1}^D A_i \prod_{j=0}^{i-1} P_j$ and

$ID(B) = \sum_{i=1}^D A'_i \prod_{j=0}^{i-1} P_j$, we have the following formulas.

$$\begin{aligned} & ID(A) - ID(B) \\ &= \sum_{i=1}^D a_i \prod_{j=0}^{i-1} P_j, (a_i = A_i - A'_i) \\ &= \sum_{i=1}^{D-1} a_i \prod_{j=0}^{i-1} P_j + a_D \prod_{j=0}^{D-1} P_j \\ &= C + a_D \prod_{j=0}^{D-1} P_j, (C = \sum_{i=1}^{D-1} a_i \prod_{j=0}^{i-1} P_j) \end{aligned}$$

Since $0 \leq A_i, A'_i \leq P_i - 1$, we get $(1 - P_i) \leq a_i \leq (P_i - 1)$.

So we have $\sum_{i=1}^{D-1} (1 - P_i) \prod_{j=0}^{i-1} P_j \leq C \leq \sum_{i=1}^{D-1} (P_i - 1) \prod_{j=0}^{i-1} P_j$, thus

$$0 \leq |C| \leq \left| \prod_{i=0}^{D-1} P_i - 1 \right| = |X - 1|, (X = \prod_{i=0}^{D-1} P_i).$$

If $ID(A) - ID(B) = 0$, then $C = -a_D \prod_{j=0}^{D-1} P_j = -a_D X$, thus $0 \leq |C| = |a_D X| \leq |X - 1|$

Because P_i and a_D are integers, the equation $ID(A) - ID(B) = 0$ is possible if and only if

$a_D = 0$ and $C = 0$. In the same way, we can prove $C = 0$ if and only if $a_i = 0$ ($i = 1..D-1$)

Thus, we prove that $ID(A) = ID(B)$ if and only if $a_i = 0$ ($i = 1..D$). That means there are no two different sub-blocks having the same id. So the ids of MFP blocks form continuous

integers from 0 to $\prod_{i=1}^D P_i - 1$. Q.E.D.

Now, we give the description of the tree corresponding to an MFP. Let the root of the tree represent the universe. When a block, B , is divided by a $\prod_{i=1}^D P_i$ MFP, we add

$\prod_{i=1}^D P_i$ children to the node corresponding to block B . These children from left to right correspond to the sub-blocks in the sequence of their ids from small to large. The process continues until all the overloaded MFP blocks have been partitioned. A tree built in the above way is called the **tree corresponding to the MFP**. The node representing an MFP block is called the **corresponding node of the MFP block**. Figure 3.4.3 illustrates the tree corresponding to an MFP Tree. We define the **level of an MFP block** as the depth of

the node corresponding to it. As we use terms including **parent**, **child**, and **sibling** to represent the relations between two nodes in a tree, we adopt the same terms to represent the relations between two MFP blocks based on the relations between the corresponding nodes.

We use a special concept, “**Morton area**,” to maintain an MFP Tree. It is the identification of a MFP block, and can be used to calculate both the position and the size of a MFP block. The Morton area of an MFP block can uniquely identify it, and this property can be used to construct the directory of an MFP Tree.

Before explaining it, we need to give the definition of area first. Let U_i and L_i be the upper and the lower bound coordinates of an MFP block, B , in dimension i , respectively. The **area of the MFP Block**, let it be $Area(B)$, is calculated as follows:

$$Area(B) = \prod_{i=1}^D (U_i - L_i). \text{ Because an overloaded MFP block is divided into } \prod_{i=1}^D P_i \text{ equal-}$$

sized sub-blocks, the area of an MFP block has the following property.

Property 3.4.1: All MFP blocks with the same parent block have the same area, which is

equal to $1 / \prod_{i=1}^D P_i$ of that of their parent block.

Now we define the **Morton area of an MFP block** as follows: The Morton area of the first MFP block, the universe, is zero. Let the Morton area of an MFP block, B , be $MA(B)$, and the block be divided by a $\prod_{i=1}^D P_i$ MFP. The Morton area of any resulting block, B' , is calculated as follows: $MA(B') = MA(B) + ID(B') * Area(B')$. Figure 3.4.3 shows the ids, the Morton areas, and the levels of blocks in a 2×2 MFP Tree and its corresponding tree. The universe's size is $100 * 100$.

Lemmas 3.4.2 and 3.4.3 define some properties of the Morton area and show that the Morton area can be used to uniquely identify an IBGF block.

Lemma 3.4.2: The Morton area of a block is equal to the sum of its left sibling's Morton area and its own area. A leftmost child's Morton area is equal to its parent's Morton area.

Proof: Let block B' be a leftmost child, whose id is thus zero. Let its parent block be B . By the definition of the Morton area, the Morton area of block B' is calculated as follows:
 $MA(B') = MA(B) + ID(B') * Area(B') = MA(B)$. So the Morton area of a leftmost child block is equal to that of its parent.

Now we suppose that block B' is not a leftmost child, and block B'' is its left sibling. Let block B be their parent. Then

$$\begin{aligned}
 ID(B'') &= ID(B') + 1 \quad \text{and} \quad Area(B'') = Area(B'). \quad \text{Thus we have the following} \\
 \text{equation. } MA(B') & \\
 &= MA(B) + ID(B') * Area(B') \\
 &= MA(B) + (ID(B'') + 1) * Area(B') \\
 &= MA(B) + ID(B'') * Area(B'') + Area(B') \\
 &= MA(B'') + Area(B')
 \end{aligned}$$

Thus, it is proved that a block's Morton area is equal to its left sibling's Morton area plus its own area. Q.E.D.

Lemma 3.4.3: In an MFP Tree, there are no two MFP blocks with identical Morton areas.

Proof: First we prove that the Morton area of a block is either larger than or equal to that of any of its ancestors. We know that the Morton area of a block is either larger than that

of its left sibling or equal to that of its parent by Lemma 3.4.2. Then, the Morton area of a block is either larger than or equal to that of its parent. Thus, the Morton area of a block is either larger than or equal to that of its ancestor.

Second, we prove that the Morton area of a block is always smaller than that of its ancestor's right sibling. Let B' be any MFP block. Let B be the parent of block B' . Let C be the right sibling of block B . Then we have the following equations.

$$MA(B') = MA(B) + ID(B') * Area(B'),$$

$$Area(B') = Area(B) / \prod_{i=1}^D P_i,$$

$$Area(B) = Area(C),$$

$$\text{and } MA(C) = MA(B) + Area(C) = MA(B) + Area(B).$$

$$\text{We also know } ID(B') \leq \prod_{i=1}^D P_i - 1.$$

So we get: $MA(B')$

$$\begin{aligned} &= MA(B) + ID(B') * Area(B) / \prod_{i=1}^D P_i \\ &\leq MA(B) + (\prod_{i=1}^D P_i - 1) * Area(B) / \prod_{i=1}^D P_i \\ &= MA(B) + Area(B) - Area(B) / \prod_{i=1}^D P_i \\ &= MA(C) - Area(B) / \prod_{i=1}^D P_i \end{aligned}$$

Thus we know $MA(B') < MA(C)$, which proves that the Morton area of a block is always smaller than that of its parent's right sibling. In the same way, we can show that the Morton area of block C is smaller than that of the right sibling of block C 's parent,

the ancestor of block B' . If we continue, we can prove that the Morton area of block B' is smaller than that of the right sibling of any of its ancestors. Since B' is selected randomly, the second part is proved.

For any two MFP blocks A and B , either they are siblings, or their ancestors are siblings. If they are siblings, let B be the right sibling of A . Then by Lemma 3.4.2., we have: $MA(A) < MA(B)$. If their ancestors are siblings, let A' be the ancestor of block A , let B' be the ancestor of block B , and B' be the right sibling of A' , then we have the following inequations: $MA(A') \leq MA(A) < MA(B') \leq MA(B) \Rightarrow MA(A) < MA(B)$.

Since blocks A and B are randomly selected, we have proved that there are no two MFP blocks having identical Morton areas. Q.E.D.

2, 6250, 2, H	3, 6875, 2, I	2, 8750, 2, L	3, 9375, 2, M
0, 5000, 2, F	1, 5625, 2, G	0, 7500, 2, J	1, 8125, 2, K
2, 1250, 2, C	3, 1875, 2, D	1, 2500, 1, E	
0, 0, 2, A	1, 625, 2, B		

(a) The ids, the Morton areas, and the levels of MFP blocks. The letters show the corresponding relations between MFP blocks and the nodes in the tree below corresponding to this MFP.

(b) The tree corresponding to the MFP. The letters show the relations between MFP blocks and the nodes in the tree.

Figure 3.4.3 The Tree Corresponding to a 2×2 MFP, and the Ids, the Morton Areas, and the Levels in the MFP Tree

6, 8437.5, 2	7, 8593.75, 2	7, 8750, 1
4, 8125, 2	5, 8281.25, 2	
2, 7812.5, 2	3, 7968.75, 2	
0, 7500, 2	1, 7656.25, 2	
4, 5000, 1		5, 6250, 1
2, 2500, 1		3, 3750, 1
0, 0, 1		1, 1250, 1

Figure 3.4.4 The Ids, the Morton Areas and the Levels in a 2×4 MFP Tree

3.4.2 Maintenance of an MFP Tree

This Section includes a discussion on the creation and the update of an MFP Tree. We present the construction of an MFP tree's directory, introduce the mapping scheme of an MFP Tree to the Torus topology of parallel processors, and then explain the algorithms for the creation, insertion, and deletion operations in an MFP Tree.

First, we need to know how to construct the directory of an MFP Tree.

Directory of an MFP Tree

The directory of an MFP Tree is mainly composed of a one-dimensional array, where there is one entry for each MFP block. The entry stores the level and the Morton area of a block, the number of points in it, and the address of the page in the disk where the points in the block are stored. In the MFP Tree, each MFP block is stored in one page of a disk. The directory also contains the number of MFP blocks and the maximum level of blocks in an MFP Tree. It is loaded into main memory while the MFP Tree is processed.

Mapping Scheme of an MFP Tree

As we mentioned before, the MFP Tree is primarily designed for parallel environments. The Multidimensional Fixed Partition is decided by the topology of processors. The topology of the processors defines how the processors are connected. There exist several commonly-used topologies. For example, there is the Ring and Linear Array, the Mesh and Torus, the Hypercube, and the Tree [2]. The Torus topology is the most popular one, and the MFP tree is well-suited for it. We extend the definition of the

two-dimensional Mesh and Torus to the multidimensional one as follows. A **mesh** is an interconnected network of processors organized in a $\prod_{i=1}^D P_i$ array, and a **torus** is a mesh where processors in each dimension are interconnected in a ring. Figure 3.4.5(b) shows a 2 x 4 torus topology of processors. For a $\prod_{i=1}^D P_i$ torus of processors, we use a $\prod_{i=1}^D P_i$ MFP such that the number of projections in dimension i is equal to P_i , the number of processors in dimension i . Thus each overloaded MFP block is divided into $\prod_{i=1}^D P_i$ sub-blocks. Each sub-block is assigned to a disk attached to a processor. There are several methods to map the block to a processor. We choose the trivial mapping method [2], which matches a block to a processor at the same topological position. This means that if we give unique ids to the processors in the same way we give ids to MFP sub-blocks, then we can map an MFP block to the processor which has the same id as the MFP block. Figure 3.4.5 shows an example. Sub-figure (a) shows the blocks and their ids in an MFP Tree. Sub-figure (b) shows a processor's topology in a parallel environment. The block in sub-figure (a) is mapped to the processor with the same id in sub-figure (b). To simplify the expressions, in the following we define the id of the processor from which a disk is accessed as the **id of the disk**. We call the disk accessed directly by a processor the **processor's disk**, and the processor itself the **processor responsible for the disk**. We use the **current disk** to refer to the current processor's disk.

6	7	7
4	5	
2	3	
0	1	
4		5
2		3
0		1

(a) Block Id and the Processor the Block is Mapped to

(b) Torus topology and Processor Id

Figure 3.4.5 The Mapping Scheme of an MFP Tree

Variables used in Algorithms

Most variables given in Section 3.2.2 are also used in the algorithms of the MFP Tree. Besides that, we define the following variables:

MfpDir: a dynamic one-dimensional array holding the blocks' MFP directory information. Each element has four components:

Level: the level of a block;

MArea: the Morton area of the block;

Number: the number of points in the block; and

Address: if the block is located on the current disk, this variable gives the page's address on the disk, otherwise, it holds the id of the disk.

Now we are ready to explain the maintenance algorithms of the MFP Tree. In our implementation, all the processors run the same program at the same time. Each

processor accesses a certain disk directly, and data or information can be transferred among processors if necessary.

Creation of an MFP Tree

The creation of an MFP Tree is implemented in two steps, which are the same as the ones used to create an IBGF. In the first step, the directory is constructed, and in the second step, the data in the MFP Tree are written to disks page by page. This is done to decrease the number of disk accesses. In the following algorithms, we assume that the input data are already loaded into a buffer in the main memory.

Algorithm 3.4.1: Creating an MFP Tree

1. CalculateMFPDir(); //construct the directory of an MFP Tree
2. For each block, if it is stored in the current disk, calculate its page address;
3. For each block, if it is stored in the current disk, write it to the disk.

In the above algorithm, only Step 3 involves a disk access.

The next algorithm shows how to construct the directory of an MFP Tree.

Algorithm 3.4.2: CalculateMFPDir()

1. For each input point
2. Calculate its block;
3. For each block, if it is full, SplitMFPBlock().

Step 2 calculates an input point's block. The details are discussed in Section 3.4.3.

The following algorithm is used to split an overloaded block.

Algorithm 3.4.3: SplitMFPBlock()

Let the index of the block to be split be i .

1. $L = MfpDir[i].Level$, $MA = MfpDir[i].MArea$
2. $L = L+1$;
3. $Area = \prod_{i=0}^{D-1} Width[i] / \prod_{i=1}^D P_i^L$
4. $MfpDir[i].Level = L$, $MfpDir[i].Address = 0$, $MfpDir[i].Number = 0$;
5. Calculate the region of block i , and update $BlockRegion[i]$;
6. Create $\prod_{i=1}^D P_i - 1$ entries in arrays $MfpDir$ and $BlockRegion$;
7. For $j = 0$ to $\prod_{i=1}^D P_i - 2$
8. $MfpDir[j + NoBlock].Level = L$;
9. $MfpDir[j + NoBlock].MArea = MA + (j+1)*Area$;
10. $MfpDir[j + NoBlock].Number = 0$;
11. $MfpDir[j + NoBlock].Address = -(j + 1)$; // Map the block to a disk
12. Calculate the region of block $(j + NoBlock)$ and update $BlockRegion[j + NoBlock]$;
13. $NoBlock = NoBlock + P_1 * P_2 * \dots * P_D - 1$
14. For each point that has been in block i
15. Find its block and increase the number of points in that block;
16. If any newly created block is overloaded, split it by calling SplitMFPBlock().

The above algorithm describes the process of dividing an MFP block into $\prod_{i=1}^D P_i$ sub-blocks. Step 3 calculates the area of the sub-blocks. Steps 4 and 5 update the original block information, including its level, address, the number of points in it, and its region. Steps 6 to 12 create $\prod_{i=1}^D P_i - 1$ new block entries and fill the information for each entry, including its level, Morton area, address, the number of points in it, and its region. Step 13 updates the amount of blocks in the MFP Tree. Steps 14 and 15 search through each point falling in the original block, calculate its block, and update the number of points in it. Step 16 checks for any overloaded sub-blocks, and splits them recursively. Steps 5 and 12 calculate the region of the MFP blocks. The detailed algorithm for this is given in Section 3.4.3, as is the algorithm to calculate the block containing a point in Step 15.

Insertion of data into an MFP Tree

Inserting data into an existing MFP Tree is quite similar to the insertion operation in an IBGF. It may cause some pages to be overloaded, requiring the splitting of pages.

Algorithm 3.4.4: Inserting Points into an MFP Tree

1. For each input point
2. Calculate its block;
3. Mark the block and increase the number of points in it;
4. For each block where there are points inserted
5. If the number of points in this block does not exceed the page capacity
6. If the block is in the current disk
7. Load the page into a buffer and append the inserted points;

8.	Write back the page;
9.	Else if the block is not in the current disk
10.	Wait for message from the processor responsible for the disk
11.	go to step 16;
12.	Else load the page into a buffer <i>TempBuf</i> ;
13.	Append the inserted points in this block into <i>TempBuf</i> ;
14.	Split the block and create new block entries with <i>SplitPage()</i> ;
15.	Broadcast buffer <i>TempBuf</i> and the new blocks' information;
16.	For each new block and the original block
17.	If the block is in the current disk
18.	Write that block's points from <i>TempBuf</i> to buffer <i>Buf2</i> ;
19.	Write <i>Buf2</i> to that block's page in the disk.

The above algorithm describes the insertion of points into an existing MFP Tree. Steps 1 to 3 calculate each input point's block, mark the block, and increase the number of points in that block. Steps 4 to 19 process each affected block. Steps 5 to 8 deal with the situation in which the block is not overloaded. Only the disk on which the block is stored is affected, and the inserted points are appended to the affected page. Steps 9 to 19 deal with the situation in which the block is overloaded. In Steps 9 to 11, the processors, which are not responsible for the disk on which the block is stored, wait until they get a message from the processor responsible for the disk. In Steps 12 to 15, the processor responsible for the disk loads the block into a temporary buffer, appends the inserted points in that buffer, splits the block, creates new block entries, updates their information,

and broadcasts the temporary buffer and the new blocks' information. In Steps 16 to 19, each processor, whose disk must store a newly created block, extracts the points in that block from the temporary buffer, and writes the points onto its disk.

Deletion of data from an MFP Tree

During deletion, the number of points in some blocks of an MFP Tree will decrease, as will the disk utilization of the MFP tree. To increase the disk utilization, we may have to merge adjacent blocks. Due to the special properties of the MFP Tree, not all adjacent blocks can be merged together. We give a name, **buddy blocks**, to those blocks which can be merged. We also used the name “buddy blocks” in the IBGF for the blocks which could be merged together. In the MFP Tree, we borrow the same concept. When a block is divided into $\prod_{i=1}^D P_i$ blocks in an MFP Tree, the resulting blocks are called **buddy blocks**. In other words, buddy blocks are sibling blocks in an MFP Tree. Before we present the deletion algorithm, we give the algorithm to calculate the buddy blocks.

Algorithm 3.4.5: CalculateMFPBuddyBlocks()

Given a block, B , whose Morton area and level are MA and L , respectively, find all of its buddy blocks. Let $MortonA$ be an array of $\prod_{i=1}^D P_i$ elements.

1. $Area = \prod_{i=0}^{D-1} Width[i] / \prod_{i=1}^D P_i^{L-1}$
2. $l = (\text{int}) \text{ floor}(MA / Area)$
3. $MortonA[0] = l * Area$

4. $Area = Area / \prod_{i=1}^D P_i$
5. for ($i = 1.. \prod_{i=1}^D P_i - 1$)
6. $MortonA[i] = MortonA[i-1] + Area;$
7. for ($j = 0 .. \prod_{i=1}^D P_i - 1$)
8. Search the *MfpDir* for the page whose Morton area and level are $MortonA[j]$ and L , respectively.

The above algorithm finds the buddy blocks of the MFP block B . Steps 1 to 3 calculate the area of the block's parent and use the *floor*() function to get the smallest Morton area of all the buddy blocks. Step 4 calculates the area of the buddy blocks. Steps 5 and 6 calculate the Morton areas of all the other buddy blocks, including the input block. Steps 7 and 8 search for the buddy block using the calculated Morton area and the level L .

We do not define a 70% threshold to merge blocks in the MFP Tree as we do in the IBGF. The reason is due to the low disk utilization in the MFP Tree, as seen in our experiments. The detailed experimental results are given in Section 4.3.3.

We are now ready to introduce the deletion algorithm.

Algorithm 3.4.6: Deleting Points from an MFP Tree

1. Calculate each point's block and mark it as an affected block;
2. For each affected block

3. Calculate its buddy blocks using CalculateMFPBuddyBlocks();
4. Load the buddy block in the current disk into a buffer;
5. Delete the input points from the buffer;
6. Calculate the number of remaining points and broadcast the number;
7. If the total number of points in the buddy blocks is less than a page's capacity
 8. Calculate in which disk to store the resulting block;
 9. If the current disk does not store the resulting block
 10. Send the buffer of remaining points in the buddy block;
 11. If the current disk does store the resulting block
 12. Wait for the remaining points' buffers sent from the other processors;
 13. Write all the remaining points in the merged block onto the disk
 14. Eliminate all the other blocks' entries and release those pages;
 15. Update the merged block's entry and decrease the number of blocks.
16. Else write back the buffer of the buddy block to the current disk;
17. Update each buddy block's entry;
18. Change the status of all the buddy blocks' pages to unaffected.

The above algorithm deletes points from an MFP Tree and merges the buddy blocks when it is possible. Step 1 calculates the block of each input point and marks the block as an affected block. Steps 2 to 18 process each affected block. Step 3 calculates the buddy blocks of the block being processed. According to the mapping scheme and the

definition of buddy blocks, each processor will process one buddy block. In Steps 4 to 6, a processor loads the buddy block in the current disk, removes the input points, calculates the number of remaining points, and broadcasts the number to all the other processors. Steps 7 to 15 deal with the situation where the number of remaining points in all the buddy blocks are less than a page's capacity, and the buddy blocks can therefore be merged into one block. Step 8 calculates the disk on which the merged block will be stored. This step is in fact calculating the number of the block according to the mapping scheme we are using. We give these details in the next algorithm. In Steps 9 and 10, the processor which is not responsible for the disk that stores the resulting block sends the buffer of remaining points in its buddy block to the processor which is responsible for the disk to store the resulting block. In Steps 11 to 13, the processor which is responsible for the disk which stores the resulting block receives the buffers of the remaining points from all the other processors, combines them into one buffer, and writes the buffer onto its disk. Steps 14 and 15 release the other buddy blocks and update the resulting block information. Steps 16 to 18 deal with the situation in which the total number of the remaining points in the buddy blocks exceeds a page's capacity. Each buddy block is then written back to the original page without the deleted input points. Step 17 updates each buddy block's information. Step 18 changes the status of each buddy block to unaffected so that it will not be re-processed.

The calculation of a block's id in order to decide which disk it is stored on is as follows.

Algorithm 3.4.7: CalculateBlockId()

Given a block, whose Morton area and level are MA and L , respectively, calculate

its id. Let $MortonA$ be an array of $\prod_{i=1}^D P_i$ elements.

1. $Area = \prod_{i=0}^{D-1} Width[i] / \prod_{i=1}^D P_i^{L-1}$
2. $l = (\text{int}) \text{ floor}(MA / Area)$
3. $MortonA[0] = l * Area$
4. $Area = Area / \prod_{i=1}^D P_i$
5. for $(i = 1.. \prod_{i=1}^D P_i - 1)$
6. $MortonA[i] = MortonA[i-1] + Area;$
7. for $(i = 0 .. \prod_{i=1}^D P_i - 1)$
8. If $(MA = MortonA[i])$
9. return i .

The above algorithm calculates a block's id. Steps 1 to 6 calculate the Morton areas of the block's siblings. These Morton areas are stored in an array in the sequence of the sibling blocks' ids. Steps 7 to 9 get the id of the block and return it.

3.4.3 Queries on an MFP Tree

We implement the point query, the range query, and the nearest-neighbor query on the MFP Tree using the same methods we used on the IBGF. To run the queries, the region of each MFP block must be calculated first. The following algorithm gives the steps.

Algorithm 3.4.8: CalculateMFPBlockRegion()

Given a block's index i in $MfpDir$, calculate its region. Let W be an array of D elements.

1. For ($j = 1..D$)
2. $W[j-1] = Width[j-1]/P_j$
3. $MA = MfpDir[i].MArea$
4. For ($j = 1..D$)
5. $BlockRegion[i].Cn[j] = Corner[j];$
6. While ($MA \neq 0$)
7. $Area = \prod_{i=0}^{D-1} W[i]$
8. $k = floor(MA / Area)$
9. For ($j = 0..D-1$)
10. $l = (k \bmod \prod_{m=1}^{j+1} P_m) / \prod_{m=1}^j P_m$
11. $BlockRegion[i].Cn[j] = BlockRegion[i].Cn[j] + l * W[j]$
12. $W[j] = W[j] / P_{j+1}$
13. $MA = MA - k * Area$

14. For ($j = 0 \dots D - 1$)
15. $BlockRegion[i].Wd[j] = W[j]$.

The above algorithm calculates the region of the MFP block, i . It goes through all the ancestors of the block in the sequence of their levels from small to large, calculates their regions, and finally discovers the region of block i . Let B be an ancestor block. Array W contains the size of block B . $W[j-1]$ contains the width of block B in the j -th dimension, and it is divided by P_j in Step 2 and Step 12 when the level of block B increases. Variable $Area$, calculated in Step 7, contains the area of block B . Variable MA is equal to the sum of the areas of block B 's left siblings and itself, and it is calculated in Steps 3 and 13. In Step 8, MA is used to calculate the id of block B , and assigns it to variable k . In Step 10, the id of block B 's projection in each dimension is calculated and assigned to variable l . The formula derives from the definition of the id of an MFP block. $BlockRegion[i].Cn[j]$ contains the coordinate of the lower bound of block B in the $(j + 1)$ -th dimension. Steps 6 to 13 are iterated as the level of block B increases until it is equal to the level of block i , such that block B becomes block i . Finally, in Steps 15 and 16, the widths of block i in each dimension are assigned.

After we calculate the region of each MFP block, we can implement the algorithms for the various queries on an MFP Tree using the same algorithms we used on the IBGF. The algorithm for the point queries is the same as Algorithm 3.2.10. The algorithm for the range queries is the same as Algorithm 3.2.11. The algorithm for the neighbor queries is the same as Algorithm 3.2.12.

In Chapter 3, we have introduced two multi-dimensional access methods, the IBGF, and the MFP Tree. In Chapter 4, we explain the experiments conducted on them and present the results.

Chapter 4 Experimental Results

This chapter introduces the experiments conducted on the IBGF and the MFP Tree. Section 4.1 describes the distribution of the experimental data sources and explains their generation. Section 4.2 presents the parallel environments on which the experiments are conducted. Section 4.3 diagrams, discusses and analyzes the experimental results.

4.1 Data Sources

In this section, we discuss the data sources used in the experiments. To represent the diversity of multi-dimensional applications, we generate four types of data sets, which have uniform, correlated, hotspot, and cluster distributions, respectively. Each is composed of 10,000 two-dimensional points in domain $[0,2000) \times [0,2000)$.

4.1.1 Uniform Data

A uniform distribution is a common one in multi-dimensional applications, where all the points in the universe are independent of each other and distributed evenly and uniformly, and the coordinates of each point in different dimensions do not rely on any other.

We generate two-dimensional uniformly distributed point sets by assigning each coordinate of each point a float selected independently, uniformly and randomly in the range $[0,2000)$. Figure 4.1 shows a uniformly distributed data set.

4.1.2 Correlated Data

A correlated distribution is found in some scientific applications in which different factors of data are dependent, and some factors rely on others functionally. In a correlated distributed data set, a point's coordinates in some dimensions are functionally dependent on its coordinates in other dimensions, thus all the points form a normal distribution along a certain line or a certain curve.

We generate two-dimensional correlated data sets meeting the function " $y = x$ " in the following way. For each point, we create 2 floats: A and B . A is a float generated uniformly and randomly in range $[0,2000)$. B is a float generated by a normal distribution, of which the mean is 0, and the deviation is defined dynamically. A point's coordinate in the first dimension is equal to A , and its coordinate in the second dimension is equal to $A + B$. Figure 4.2 shows a correlated data set.

4.1.3 Hotspot Data

A hotspot distribution is another popular distribution in multi-dimensional applications, in which all the points form a normal distribution around the center of the universe, such that the density of the points near the center is much higher than that of the points farther from the center.

We create two-dimensional hotspot data sets as follows. The coordinate of a point in the first dimension is generated by a normal distribution, of which the mean is the coordinate of the universe's center in the first dimension, and the deviation is defined dynamically. The coordinate of a point in the second dimension is generated by another normal distribution, of which the mean is the coordinate of the universe's center in the

second dimension, and the deviation is defined dynamically. Figure 4.3 shows a hotspot data set.

4.1.4 Clusters

A cluster distribution is often found in geographical applications, in which points are usually distributed in clusters. The densities of points in clusters are very high, and there are few points outside the clusters.

We create the cluster data sets by defining the number of clusters and the center and the deviation of each cluster dynamically. Each cluster is created in the same way as a hotspot distributed data set. Figure 4.4 shows a cluster data set.

Figure 4.1 Uniform Data Set

Figure 4.2 Correlated Data Set

Figure 4.3 Hotspot Data Set

Figure 4.4 Cluster Data Set

4.2 Parallel Environments

This section outlines the parallel environments in which the experiments are conducted. There are two parallel environments, a Beowulf system and a Sunfire system. The Beowulf system is a multi-processor, distributed-memory, and multi-disk environment, and the Sunfire system is a multi-processor, distributed-memory, and single disk array environment.

4.2.1 Beowulf System

The Beowulf system at Carleton University is composed of 64 nodes, each with 2 processors, a disk and its own RAM. Of these nodes, 32 have dual 1.7 GHz Xeon Processors and 1 GB RAM, and the other 32 nodes have dual 2.0 GHz Xeon Processors and 1.5 GB RAM. Each of these 64 nodes has 60 GB of disk storage. All nodes are interconnected via a Cisco 6509 switch using Gigabit ethernet.

4.2.2 Sunfire System

The Sunfire system at Carleton University is composed of 1 node with 20 - 900 MHz UltraSPARC III Cu Processors, and each processor has 20 GB of memory. It has 1.3 TB disk storage on one Sun StorEdge T3 enterprise disk array.

The Sun StorEdge T3 disk array is a special storage technique. It consists of two high-performance RAID controllers. They provide dual fibre-channel interfaces. Each of them has its own cache, delivering up to 2 GB of memory for fast access to frequently accessed data, and employs a separate array of nine disk drives.

4.3 Experimental Results and Analysis

This section illustrates the experimental results, and discusses and analyzes them.

To find the similarities and the differences between the IBGF and the MFP Tree, we did a large number of experiments. Through the experiments, we compare several aspects of the two multi-dimensional access methods, including their disk utilization, creation time, and the responding time and running time of range queries. To show the improvement of the IBGF over the grid file, we compare the directory size of the IBGF to that of the grid file. We also run experiments to test the performance of the dynamic insertion algorithm that we invented for the parallel IBGFs. We confine the page capacity to 5 points in most experiments, except those of Section 4.3.2, in which we confine the page capacity to 20 points.

4.3.1 Directory Size

As we explained in Section 2.2.2, the IBGF is invented to address the super-linear growth of the directory size of the grid file. In this section, we present our experimental results proving the advantage of the IBGF over the grid file with respect to the directory size.

In Section 3.2.2, we revealed the composition of the directory of the IBGF. It is composed of a one-dimensional array, whose entries contain the IBGF block's information, and two variables containing the number of pages and the maximum level of the partitions in the IBGF. In Section 2.2.1, we explained the construction of the directory of the grid file. It consists of D one-dimensional grid arrays, which contain the scales on

each dimension, and a D -dimensional grid array, which contains the grid block's information. To make the comparison easier and clearer, we ignore the space taken up by the 2 variables in the directory of the IBGF and scale in the directory of the grid file. We compare the number of entries instead of the exact space taken up in the one-dimensional array in the directory of the IBGF and the D -dimensional array in the directory of the grid file. Although each entry in the IBGF consists of two more integers than that in the grid file, the difference in space taken is only a small constant ratio and has few effects on the comparison results.

Table 4.2 shows the comparison results of our experiments. The rates of the directory size of the grid files to that of the IBGFs are calculated and converted to the exponents of the number of dimensions. The experimental results prove the advantages of the IBGF.

Data Set	Number of entries in the directory of the file			
	Grid file	IBGF	Grid file / IBGF	Grid file/IBGF(as the exponent of 2)
Uniform	9548	2880	3.32	1.73
Correlated	20410	2903	7.03	2.81
Hot Spot	13095	2934	4.46	2.16
Clusters	45628	2984	15.29	3.93

Table 4.1 Comparison of Directory Size in the Grid File and the IBGF

4.3.2 Partitions in IBGF and MFP Trees

We implemented the IBGF, the parallel IBGFs, and the MFP Tree and used them to store the four data sets described in Section 4.2, respectively. To make the partitions and blocks discernable in figures, we confine the page capacity to 20 points instead of 5 points.

Figures 4.5 to 4.8 show the partition of space in the IBGF. In them, the outbound black rectangle stands for the universe, each small colored rectangle represents a block, and the colored points in the rectangle represent the points falling in the block. Figures 4.9 to 4.16 show the different partitioning and disk assignments of blocks between the parallel IBGFs and the MFP tree in four-processor (or two $\times$ two processor) parallel environment is shown. The different colors of blocks show their disk allocation. The blocks with the same colors are assigned to the same disks.

In Section 3.4.1, we gave Observation 3.4.2, which says that there are more blocks in an MFP Tree than in an IBGF given the same input point set. Figures 4.5 to 4.16 support that observation. In the next section, we give the detailed experimental results and analysis. From the figures, we can also see how blocks are distributed onto multiple disks. In the MFP Tree figures, there exists a certain rule to assign blocks to disks, which results from the mapping scheme described in Section 3.4.2. In the parallel IBGF figures, it is difficult to see the rules used to assign blocks to disks because of the use of the proximity-based mapping method described in Section 3.3.2. The detailed experimental results will be illustrated in the following sections.

Figure 4.5 Uniform Data Set in an IBGF

Figure 4.6 Correlated Data Set in an IBGF

Figure 4.7 Hotspot Data Set in an IBGF

Figure 4.8 Cluster Data Set in an IBGF

Figure 4.9 Uniform Data Set in 4-Disk Parallel IBGFs

Figure 4.10 Uniform Data Set in 2×2 Disk MFP Tree

Figure 4.11 Correlated Data Set in 4-Disk Parallel IBGFs

Figure 4.12 Correlated Data Set in 2×2 Disk MFP Tree

Figure 4.13 Hotspot Data Set in 4-Disk Parallel IBGFs

Figure 4.14 Hotspot Data Set in 2×2 Disk MFP Tree

Figure 4.15 Cluster Data Set in 4-Disk Parallel IBGFs

Figure 4.16 Cluster Data Set in 2×2 Disk MFP Trees

4.3.3 Disk Utilization

Disk utilization is an important factor to consider in evaluating a file system. In this section, we present the experimental results on the disk utilization of the IBGF and the MFP Tree. According to the definition in Section 3.1, we derive the following formula:

$$\text{Number of Pages} = \text{Number of Points} / (\text{Page Capacity} * \text{Disk Utilization})$$

The disk utilization gives the number of pages that a file occupies given identical data set size and page capacity. The higher the disk utilization, the less disk space a file occupies. Conversely, the lower the disk utilization, the more disk space a file occupies.

Table 4.2 and Chart 4.1 show the disk utilization of the IBGF and the MFP Tree. The disk utilization of the IBGF falls in the range of 67% to 70%, while that of the MFP Tree is lower, falling in the range of 14% to 46%.

As we have described in Section 3.2.1 and Section 3.4.1, in the IBGF an overloaded block is divided into 2 equal-sized sub-blocks, and in the MFP Tree an overloaded block is divided into $\prod_{i=1}^D P_i$ equal-sized sub-blocks. The more sub-blocks an overloaded block is divided into, the more new pages the file occupies, although in the MFP Tree blocks do not become overloaded as frequently as in the IBGF. The experiments prove that the number of sub-blocks that an overloaded block is divided into plays a more important role in deciding the number of pages a file system occupies than the frequency with which a block becomes overloaded. Accordingly, we make the following observation:

Observation 4.1: The more sub-blocks an overloaded block is divided into, the lower the disk utilization of the file system.

Furthermore, Table 4.2 and Chart 4.1 also show that the disk utilization of the IBGF is not affected by the number of disks; however, the disk utilization of the MFP Tree tends to decrease as the number of disks increases. This is because the number of disks does not affect the partition of the IBGF, but is equal to the number of sub-blocks in the MFP Tree. The trend of disk utilization in the MFP Tree supports Observation 4.1.

Input Data	IBGF			MFP Tree			IBGF /MFP
	Number of Disks	Number of Pages	Disk Utilization	Number of Disks	Number of Pages	Disk Utilization	
Uniform	4	2880	69.44%	2*2	4330	46.19%	1.50
	8	2880	69.44%	2*4	5181	38.60%	1.80
	16	2880	69.44%	4*4	6376	31.37%	2.21
Correlated	4	2903	68.89%	2*2	4486	44.58%	1.55
	8	2903	68.89%	2*4	7582	26.38%	2.61
	16	2903	68.89%	4*4	13906	14.38%	4.79
Hot Spot	4	2934	68.17%	2*2	4357	45.90%	1.49
	8	2934	68.17%	2*4	6812	29.36%	2.32
	16	2934	68.17%	4*4	12361	16.18%	4.21
Clusters	4	2984	67.02%	2*2	4516	44.29%	1.51
	8	2984	67.02%	2*4	7106	28.15%	2.38
	16	2984	67.02%	4*4	12961	15.43%	4.34

Table 4.2 Comparison of Disk Utilization in the IBGF and the MFP Tree

Chart 4.1 Comparison of Disk Utilization between the IBGF and the MFP Tree

4.3.4 Creation Time

The time spent to create a file system is another important factor in evaluating a file system, especially if the applications are frequently updated. In this section, we present the experimental results for the creation time of the IBGF and MFP Tree.

To reflect the real world, we simulate two situations to create files, the static one and the dynamic one. In the static situation, all of the 10,000 input points are given at one time, and the files are then created completely. In the dynamic situation, the input data are given at five times, 2000 points each time, and the creation of the files is not complete until all the input data are inserted. We do the experiments in both the Beowulf environment and the Sunfire environment. Charts 4.2 to 4.5 show the results.

We look at the static situation first. On the Beowulf system, the time spent to create an MFP Tree is about 5.61% to 15.22% of the time spent to create an IBGF. On the Sunfire system, the time spent to create an MFP Tree is 14.73% to 32.37% of the time spent to create an IBGF.

To reveal the reason for the significant differences between the creation times of the IBGF and those of the MFP Tree, we cut the time spent to create files into 3 parts: the time to access the disks, the time to construct the directories, and the time to assign blocks to disks. In the MFP Tree, the time spent to assign blocks to disks can be ignored. Charts 4.6 to 4.9 show the detailed results. The time spent to write to the disks is negligible compared to the other parts, as we expected. Instead, the time spent in directory construction and disk assignment makes up a significant portion of the total creation time. These results challenge the fact that disk access time is quite slow

compared with CPU processing time and main memory access time. The only reason we can give for this result is the big cache that exists in the Sunfire and Beowulf systems. However, we do find a dramatic increase in time to create an MFP Tree on the Sunfire system compared to that on the Beowulf system, while there is no such increase in the IBGF. In Section 4.2, we explained that on the Beowulf system, every 2 processors have 60GB of disk storage, while on the Sunfire system, all processors are linked to one disk array. The time to access the disk on the Sunfire system is thus slower than that on the Beowulf system. In Section 4.3.3, we show that the number of pages in the MFP Tree is about 1.5 to 4.8 times that in the IBGF. Accordingly, longer disk accessing time on the Sunfire system compared to that on the Beowulf system explains the slowdown in creating the MFP Tree.

The directory construction time of an MFP Tree is shorter than that of an IBGF. As we stated in Observation 3.4.1, the MFP Tree incurs less frequent block repartitioning than the IBGF using the same input data set. The observation is supported by the experimental results.

Now we look at the dynamic method. First of all, we compare the dynamic creation time to the static one. In Section 3.3.3, we introduced the insertion algorithm for the parallel IBGFs that we invented and implemented. Table 4.4 shows that the dynamic method works well in most data distribution, as fast as the static method or even faster. We are not sure why the algorithm does not work very well in the cluster distribution. The main difference between the cluster data source and other data sources is that the data are organized cluster by cluster, and not in a random sequence. The dynamic method

of the MFP Tree is not as fast as the static method, but it is steady with the same ratio in all the data distributions.

Data Set	Disks	Sunfire System			Beowulf System		
		IBGF	MFP	MFP /IBGF	IBGF	MFP	MFP /IBGF
Uniform	4	16990	2900	17.07%	15190	990	6.52%
	8	17010	2750	16.17%	15190	920	6.06%
	16	17470	2670	15.28%	15330	860	5.61%
Correlated	4	18330	3170	17.29%	16190	1180	7.29%
	8	18410	4020	21.84%	16260	1530	9.41%
	16	18760	5950	31.72%	16430	2500	15.22%
Hot Spot	4	19750	2910	14.73%	17250	1010	5.86%
	8	19710	3590	18.21%	17360	1230	7.09%
	16	20020	5250	26.22%	17510	1960	11.19%
Clusters	4	28970	4490	15.50%	24790	1610	6.49%
	8	29180	5890	20.19%	24780	2150	8.68%
	16	29410	9520	32.37%	24920	3780	15.17%

Table 4.3 Comparison of Static Creation Time (ms)

Data Set	Disks	Sunfire System			Beowulf System		
		IBGF	MFP	MFP /IBGF	IBGF	MFP	MFP /IBGF
Uniform	4	17120	4540	26.52%	15760	1360	8.63%
	8	17220	5610	32.58%	15800	1290	8.16%
	16	17940	5170	28.82%	15810	1240	7.84%
Correlated	4	17750	4780	26.93%	15980	1560	9.76%
	8	17710	5990	33.82%	16010	1950	12.18%
	16	17870	9430	52.77%	16110	2900	18.00%
Hot Spot	4	19620	4430	22.58%	17570	1360	7.74%
	8	19730	6700	33.96%	17570	1650	9.39%
	16	21240	9480	44.63%	17690	2380	13.45%
Clusters	4	67070	7330	10.93%	51250	2690	5.25%
	8	67280	9550	14.19%	51270	3420	6.67%
	16	67910	13360	19.67%	51340	4840	9.43%

Table 4.4 Comparison of Dynamic Creation Time (ms)

Data Set	Disks	Sunfire System						Beowulf System					
		IBGF			MFP			IBGF			MFP		
		Static	Dyn-amic	Dyn-amic/Static	Static	Dyn-amic	Dyn-amic/Static	Static	Dyn-amic	Dyn-amic/Static	Static	Dyn-amic	Dyn-amic/Static
Uniform	4	16990	17120	1.01	2900	4540	1.57	15190	15760	1.04	990	1360	1.37
	8	17010	17220	1.01	2750	5610	2.04	15190	15800	1.04	920	1290	1.40
	16	17470	17940	1.03	2670	5170	1.94	15330	15810	1.03	860	1240	1.44
Cor-related	4	18330	17750	0.97	3170	4780	1.51	16190	15980	0.99	1180	1560	1.32
	8	18410	17710	0.96	4020	5990	1.49	16260	16010	0.98	1530	1950	1.27
	16	18760	17870	0.95	5950	9430	1.58	16430	16110	0.98	2500	2900	1.16
Hot Spot	4	19750	19620	0.99	2910	4430	1.52	17250	17570	1.02	1010	1360	1.35
	8	19710	19730	1.00	3590	6700	1.87	17360	17570	1.01	1230	1650	1.34
	16	20020	21240	1.06	5250	9480	1.81	17510	17690	1.01	1960	2380	1.21
Clusters	4	28970	67070	2.32	4490	7330	1.63	24790	51250	2.07	1610	2690	1.67
	8	29180	67280	2.31	5890	9550	1.62	24780	51270	2.07	2150	3420	1.59
	16	29410	67910	2.31	9520	13360	1.40	24920	51340	2.06	3780	4840	1.28

Table 4.5 Compare Dynamic Creation Time (ms) to Static Creation Time (ms)

Static Vs. Dynamic Creation Time on Uniform Data

Chart 4.2 Comparison of Creation Time on Uniform Data Set

Static Vs. Dynamic Creation Time on Correlated Data

Chart 4.3 Comparison of Creation Time on Correlated Data Set

Chart 4.4 Comparison of Creation Time on Hotspot Data Set

Chart 4.5 Comparison of Creation Time on Cluster Data Set

Chart 4.6 Analysis of Creation Time on Uniform Data Set

Chart 4.7 Analysis of Creation Time on Correlated Data Set

Chart 4.8 Analysis of Creation Time on Hotspot Data Set

Chart 4.9 Analysis of Creation Time on Cluster Data Set

4.3.5 Query Time

The performance of queries on a data set is one of the most important factors in assessing a multi-dimensional access method. Among all the various queries, the range query is one of the most popular in multi-dimensional applications. In this section, we present and analyze the experimental results of range queries on the parallel IBGFs and the MFP Tree.

We implement a program to generate the query windows. There are three sizes of query windows: $10\% \times 10\%$, $20\% \times 20\%$, and $30\% \times 30\%$ of the universe's size. For each size, we generate 1000 query windows randomly in the universe. Altogether, we generate 3000 query windows. We do experiments in both the Sunfire environment and the Beowulf environment. In each environment, we test query performance using 4 processors, 8 processors, and 16 processors, respectively. We run queries on static parallel IBGFs, dynamic parallel IBGFs, static MFP Trees, and dynamic MFP Trees built from the data sets of the four distributions described in Section 4.1.

First, we evaluate query performance based on response time. **Response time** is equal to $Max[N_i]$, where N_i is the number of pages retrieved from the i -th disk in a query. According to the definition, response time only considers the number of disk accesses, not the time to access disks, and it neglects the CPU time and main memory access time. The advantage of using response time is that it is not affected by the hardware environment or other running software.

Charts 4.10 to 4.13 show the experimental results of the response time of queries. Table 4.6 gives the detailed data. The response time of queries on dynamic MFP Trees

and static MFP Trees is the same as expected, because MFP Trees, whether created statically or dynamically, have the same partitions and disk assignments. However, queries on dynamic IBGFs and static IBGFs perform differently, because IBGFs created dynamically follow different rules from IBGFs created statically. The experimental results are as follows. The response time of queries on the dynamic IBGF is only about 0 to 0.12 times larger than that on the static IBGF. The response time of queries on the MFP Tree is about 0.3 to 3.25 times larger than that on the static IBGF. The former results prove that the dynamic parallel IBGF we invented and introduced in Section 3.3 performs well as far as range queries are concerned. The reason that the MFP Tree performs worse can be mainly attributed to its low disk utilization, such that a query accesses more pages in the MFP Tree than in the IBGF. Another difference between the MFP Tree and the parallel IBGFs is in the method used to assign pages to disks. In the MFP Tree, a certain mapping scheme is followed to assign pages to disks irrespective of the data distribution in the universe. In the parallel IBGFs, the mapping method considers the proximity between every two blocks and tries to assign pages to disks such that the proximity between every two blocks in one disk is minimized.

Although response time is a theoretically acceptable measurement of query performance, the actual running time of queries is more of a concern in practice. We thus compare the running time of range queries on the static IBGF, the dynamic IBGF and the static MFP Tree created from the four distribution data sets on the Sunfire environment and the Beowulf environment, using the same query windows. Charts 4.14 to 4.17 show the experimental results, and Table 4.7 gives the detailed data. As has been mentioned, running time is influenced by the configuration of the hardware environment and the

other software running on it simultaneously, and as such it may not be an accurate measurement. In keeping with this, we only use running time as a reference to compare the IBGF and the MFP Tree. As shown in the experimental results, superior response time does not necessarily imply superior running time, due to the use of caches in memory management. Fewer swaps between the cache and the external memory reduce the running time. Overall, dynamic IBGFs perform slightly worse than do the static ones, and the MFP Tree performs even worse than the dynamic IBGF, although not as badly as in the experimental results on the response time. The former results again show that our dynamic IBGF performs well. The same reason we gave to explain the MFP Tree's slow response time can be used to explain its slow running time.

Chart 4.10 Comparison of Response Time on Uniform Data Set

Chart 4.11 Comparison of Response Time on Correlated Data Set

Chart 4.12 Comparison of Response Time on Hotspot Data Set

Chart 4.13 Comparison of Response Time on Cluster Data Set

Input	Range	Disk Number		Average Response Time				
Data	Size			Static	Dynamic	Static&Dynamic	(B)	(C)
		IBGF	MFP	IBGF(A)	IBGF(B)	MFP(C)	/(A)	/(A)
Uniform	10%*10%	4	2*2	11.68	12.34	16.94	1.06	1.37
		8	2*4	6.54	6.81	13.23	1.04	1.94
		16	4*4	3.87	4.01	6.06	1.04	1.51
	20%*20%	4	2*2	36.74	40.37	52.68	1.10	1.30
		8	2*4	19.55	20.35	36.84	1.04	1.81
		16	4*4	10.77	10.98	22.28	1.02	2.03
	30%*30%	4	2*2	76.44	85.27	111.34	1.12	1.31
		8	2*4	39.80	41.61	76.56	1.05	1.84
		16	4*4	21.22	21.73	41.32	1.02	1.90
	Average	4	2*2	41.62	45.99	60.32	1.11	1.31
		8	2*4	21.96	22.92	42.21	1.04	1.84
		16	4*4	11.95	12.24	23.22	1.02	1.94
Correlated	10%*10%	4	2*2	12.22	12.75	18.21	1.04	1.43
		8	2*4	6.89	7.06	18.37	1.02	2.60
		16	4*4	4.00	4.11	13.88	1.03	3.38
	20%*20%	4	2*2	43.36	46.17	67.17	1.06	1.45
		8	2*4	23.04	23.88	63.93	1.04	2.68
		16	4*4	12.34	12.70	51.06	1.03	4.02
	30%*30%	4	2*2	96.60	104.24	150.64	1.08	1.45
		8	2*4	50.23	52.59	138.71	1.05	2.64
		16	4*4	26.28	27.18	115.5	1.03	4.25
	Average	4	2*2	50.73	54.39	78.67	1.07	1.46
		8	2*4	26.72	27.84	73.67	1.04	2.63
		16	4*4	14.21	14.66	60.14	1.03	4.01
Hot Spot	10%*10%	4	2*2	13.07	13.69	18.93	1.05	1.38
		8	2*4	7.16	7.48	18.19	1.04	2.43
		16	4*4	4.18	4.29	13.26	1.03	3.09
	20%*20%	4	2*2	53.44	56.63	80.4	1.06	1.42
		8	2*4	27.88	29.18	71.52	1.05	2.45
		16	4*4	15.02	15.22	56.46	1.01	3.71
	30%*30%	4	2*2	137.23	146.27	208.18	1.07	1.42
		8	2*4	70.44	74.11	178.55	1.05	2.41
		16	4*4	36.91	37.55	149.62	1.02	3.98
	Average	4	2*2	67.91	72.19	102.5	1.06	1.42
		8	2*4	35.16	36.92	89.42	1.05	2.42
		16	4*4	18.71	19.02	73.11	1.02	3.85
Clusters	10%*10%	4	2*2	10.37	10.59	16.24	1.02	1.53
		8	2*4	5.79	5.95	15.34	1.03	2.58
		16	4*4	3.42	3.56	11.54	1.04	3.24
	20%*20%	4	2*2	39.23	39.97	60.68	1.02	1.52
		8	2*4	20.53	21.17	53.72	1.03	2.54
		16	4*4	10.84	11.36	43.12	1.05	3.80
	30%*30%	4	2*2	95.46	97.72	145.11	1.02	1.48
		8	2*4	49.02	50.87	124.8	1.04	2.45
		16	4*4	25.36	26.49	103.53	1.04	3.91
	Average	4	2*2	48.35	49.43	74.01	1.02	1.51
		8	2*4	25.11	26.00	64.62	1.04	2.49
		16	4*4	13.21	13.80	52.73	1.05	3.77

Table 4.6 Comparison of Query Response Time

Chart 4.14 Comparison of Running Time on Uniform Data Set

Chart 4.15 Comparison of Running Time on Correlated Data Set

Chart 4.16 Comparison of Running Time on Hotspot Data Set

Chart 4.17 Comparison of Running Time on Cluster Data Set

Input Data Set	Range Size	#of Disks	Average Running Time									
			Sunfire					Beowulf				
			Static IBGF (A)	Dyn IBGF (B)	MFP (C)	(B)/(A)	(C)/(A)	Static IBGF(D)	Dyn IBGF (E)	MFP (F)	(E)/(D)	(F)/(D)
Uniform	10% x10%	4	830	870	1080	1.05	1.30	560	440	490	0.79	0.88
		8	690	690	1040	1.00	1.51	410	400	530	0.98	1.29
		16	620	630	1000	1.02	1.61	400	410	580	1.03	1.45
	20% x20%	4	1580	1760	2130	1.11	1.35	780	640	680	0.82	0.87
		8	1100	1130	1760	1.03	1.60	480	500	650	1.04	1.35
		16	870	870	1370	1.00	1.57	450	450	660	1.00	1.47
	30% x30%	4	2790	3090	3760	1.11	1.35	920	860	880	0.93	0.96
		8	1830	1860	2900	1.02	1.58	580	600	830	1.03	1.43
		16	1330	1290	2000	0.97	1.50	490	450	770	0.92	1.57
	Average	4	1733	1907	2323	1.10	1.34	753	647	683	0.86	0.91
		8	1207	1227	1900	1.02	1.57	490	500	670	1.02	1.37
		16	940	930	1457	0.99	1.55	447	437	670	0.98	1.50
Cor-related	10% x10%	4	830	880	1110	1.06	1.34	450	450	510	1.00	1.13
		8	730	710	1380	0.97	1.89	410	410	710	1.00	1.73
		16	620	630	1920	1.02	3.10	390	390	1130	1.00	2.90
	20% x20%	4	1780	1890	2410	1.06	1.35	640	640	730	1.00	1.14
		8	1230	1310	2460	1.07	2.00	490	520	910	1.06	1.86
		16	930	950	2620	1.02	2.82	450	450	1300	1.00	2.89
	30% x30%	4	3370	3680	4610	1.09	1.37	870	910	1060	1.05	1.22
		8	2160	2290	4270	1.06	1.98	580	640	1160	1.10	2.00
		16	1510	1570	3910	1.04	2.59	510	540	1500	1.06	2.94
	Average	4	1993	2150	2710	1.08	1.36	653	667	767	1.02	1.17
		8	1373	1437	2703	1.05	1.97	493	523	927	1.06	1.88
		16	1020	1050	2817	1.03	2.76	450	460	1310	1.02	2.91
Hot Spot	10% x10%	4	880	910	1130	1.03	1.28	460	450	500	0.98	1.09
		8	720	720	1340	1.00	1.86	410	410	660	1.00	1.61
		16	650	660	1750	1.02	2.69	400	400	1060	1.00	2.65
	20% x20%	4	2060	2190	2780	1.06	1.35	670	690	760	1.03	1.13
		8	1420	1450	2660	1.02	1.87	510	530	870	1.04	1.71
		16	1060	1070	2680	1.01	2.53	450	460	1210	1.02	2.69
	30% x30%	4	4610	4910	6240	1.07	1.35	1000	107	1200	1.07	1.20
		8	2930	3080	5450	1.05	1.86	640	720	1190	1.13	1.86
		16	1990	1980	4850	0.99	2.44	530	570	1480	1.08	2.79
	Average	4	2517	2670	3383	1.06	1.34	710	737	820	1.04	1.15
		8	1690	1750	3150	1.04	1.86	520	553	907	1.06	1.74
		16	1233	1237	3093	1.00	2.51	460	477	1250	1.04	2.72
Clusters	10% x10%	4	760	780	1070	1.03	1.41	400	400	440	1.00	1.10
		8	660	670	1240	1.02	1.88	380	370	620	0.97	1.63
		16	610	610	1780	1.00	2.92	370	370	1050	1.00	2.84
	20% x20%	4	1580	1640	2140	1.04	1.35	490	510	580	1.04	1.18
		8	1140	1160	2090	1.02	1.83	430	450	740	1.05	1.72
		16	840	890	2430	1.06	2.89	410	400	1140	0.98	2.78
	30% x30%	4	3220	3360	4470	1.04	1.39	670	720	810	1.07	1.21
		8	2100	2140	3850	1.02	1.83	490	560	920	1.14	1.88
		16	1420	1440	3810	1.01	2.68	450	470	1310	1.04	2.91
	Average	4	1853	1927	2560	1.04	1.38	520	543	610	1.04	1.17
		8	1300	1323	2393	1.02	1.84	433	460	760	1.06	1.76
		16	957	980	2673	1.02	2.79	410	413	1167	1.01	2.85

Table 4.7 Comparison of Query Running Time

Chapter 5 Summary and Future Work

We implemented the IBGF. We designed the improved algorithms of parallel IBGFs and implemented them. We compared the parallel IBGFs to the MFP tree on different data distributions with regards to maintenance, range queries, and disk utilization. The MFP Tree takes less time to do updates in dynamic applications, while the IBGF tends to have better disk utilization and performs better in range queries. The MFP Tree tends to perform worse when the disk number becomes bigger. The 2×2 MFP Tree has the best overall performance.

Since accessing the disk is much slower than accessing main memory and CPU calculations, our current work places significant emphasis on decreasing the number of disk accesses. However, as the test results show, main memory accesses and CPU calculations take on an important role in the broader running time due to the large caches used in UNIX systems. Our future work will improve the algorithms to reduce the time to access the main memory and the CPU processing time. By changing the IBGF's directory structure from an array to a tree, it is possible to decrease the point query time to $O(ML)$, where ML is the maximum level of the blocks.

To improve the MFP Tree, the partition method, which was based on the disk number, should be altered to take into account the distribution of data. One method would be to use 2×2 partitions each time, and assign the disk number according to the level. To combine the MFP Tree and the IBGF, to incorporate the advantages of each, is also a promising idea.

References

- [1] J. Nievergelt and H. Hinterberger. The Grid File: an Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38-71, 1984.
- [2] Doron Nussbaum. *Parallel Spatial Modeling*. Ph.D. thesis, 2001.
- [3] M Ouksel. The IBGF. *PODS*, pages: 20-27, 1985.
- [4] Bongki Moon, Anurag Acharya, and Joel Saltz. Study of Scalable Declustering Algorithms for Parallel Grid Files. *IPPS*, 1996, 434-440.
- [5] Ibrahim Karmel and Christos Faloutsos. Parallel R-trees. *ACM SIGMOD* 21(2): 195-204, 1992.
- [6] Volker Gaede and Oliver Gunther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170-331, 1998.
- [7] Walter A Burkhard. Interpolation-Based Index Maintenance. *BIT* 23: 274-294, 1983.
- [8] Kyoosang Cho, Yijie Han, Yugyung Lee, and E.K.Park. Dynamic and Hierarchical Spatial Access Method using Integer Searching. *CLKM* 2001: 341-348.
- [9] Klaus Hinrichs. Implementation of the grid file: design concepts and experience. *BIT*, 25:569-592, 1985.
- [10] Michael Freeston. "The BANG File: a new kind of grid file." *ACM SIGMOD*: 260-269, 1987.
- [11] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. "Twin Grid Files: Space Optimizing Access Schemes." *ACM SIGMOD* 1988: 183-190.

- [12] Hans-Werner Six and Perter Widmayer. "Spatial Searching in Geometric Databases." ICDE 1988:496-503.
- [13] Henk Blanken, alle Ulbema, Paul Meek, and Bert van den Akker. "The Generalized Grid File: Description and Performance Aspects." ICDE 1990: 380-388.
- [14] Jianzhong Li, Jaideep Srivastava, and Doron Rotem. "CMD: A Multidimensional Declustering Method for Parallel Database Systems." VLDB 1992: 3-14.
- [15] Jianzhong Li, Jaideep Srivastava, and Doron Rotem. "Algorithms for loading Parallel Grid Files." ACM SIGMOD: 347-356, 1993.
- [16] Scott T. Leutenegger and David M. Nicol. "Efficient Bulk-Loading of Gridfiles." IEEE 1997: 410-420.
- [17] Yvonne Zhou, Shashi Shekar, and Mark Coyle. "Disk Allocation Methods for Parallelizing Grid Files." ICDE 1994: 243-252.
- [18] Duen-Ren Liu and Shashi Shekhar. "A similarity Graph-Based Approach to Declustering Problems and Its Application towards Parallelizing Grid Files." IEEE 1995: 373-381.
- [19] Mireille Regnier. "Analysis of Grid File Algorithms." BIT 25 (1985), 335-357.
- [20] B.W. Kernighan and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs." The Bell System Technical Journal, 49(1): 291-307, 1970.
- [21] Christian Bohm, Stefan Berchtold, and Daniel A. Keim. "Searching in High-Dimensional Spaces – Index Structure for Improving the Performance of Multimedia Databases" ACM Computing Surveys, 33(3):323-373, 2001.
- [22] Hpcvl hardware. <http://www.hpcvl.org/hardware/index.html>.

[23] Normal distribution random number generator.

<http://www.geocities.com/WallStreet/9245/vba4.htm>.

[24] The bi-variate normal distribution.

<http://www.math.uah.edu/statold/special/special7.html>.