1

of Canada

du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**Efficient Parallel Data Structures
For Fine-Grained Hypercube Multiprocessors**

by
Andrew Rau-Chaplin

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**

December 22, 1989

© copyright 1989, Andrew Rau-Chaplin

of Canada

du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

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

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

ISBN 0-315-59021-1

The undersigned recommended to the Faculty of Graduated Studies
and Research acceptance of the thesis

**Efficient Parallel Data Structures
For Fine-Grained Hypercube Multiprocessors**

submitted by Andrew Rau-Chaplin B.A.
in partial fulfillment of the requirements for
the degree of Master Of Computer Science

Thesis Supervisor

Chairman, School of Computer Science

Carleton University

December 22, 1989

ABSTRACT

In this thesis we show that for hypercube multiprocessors it is possible to design elegant yet efficient algorithms based on parallel implementations of advanced data structures. We describe a new general technique for the design of parallel versions of a large class of standard sequential data structures. In demonstrating this technique, we present for the hypercube multiprocessor new upper bounds for several fundamental problems in computational geometry, including next element search, trapezoidal decomposition, and triangulation.

We further extend our data structuring technique by presenting a parallel version of fractional cascading designed for fine-grained hypercube multiprocessors. Parallel fractional cascading allows us to address many additional applications in computational geometry.

Lastly, we describe a technique for creating and maintaining parallel versions of a restricted class of dynamic data structures, namely trees. Using these dynamic trees, we present a general approach for implementing parallel branch and bound search algorithms on fine-grained hypercube multiprocessors.

For My Parents:
John and Laraine Chaplin

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance, patience and friendship of Dr. Frank Dehne. He took in an admitted "scruffy" and has produced something of a "neat". In considering the student/supervisor relationship as an apprenticeship in being an academic, he has taught me more about research than I could have otherwise hoped for.

Joint research with Afonso Ferria was more fun than "work" has any right to be. Even his rare knack for generating counter examples has not reduced my desire to work with him in the again. Thanks Afonso.

No graduate experience would be complete without an endless dialog with one's fellow grad students. I'd like to thank in particular Alan Knight, Una-May O'Reilly, and Dwight Duego for making these years such an enjoyable and "intellectually" stimulating experience.

The support of Bell Northern Research through their Post Graduate Award Program is gratefully acknowledged. In particular, I'd like to thank Dr. Jack Dymont and members of the University Interaction Group for their time and assistance.

Lastly I want to thank my wife Michelle for her love and support during my time at Carleton. This thesis would not have been possible without her; little did she know that my return to university would necessitate her plowing roads, building barns, and shoveling roofs, in my absences.

TABLE OF CONTENTS

1 Introduction.....	1
2 An Introduction to Parallel Models of Computing	9
2.1 The Parallel Random Access Machine	9
2.1.2 Measuring the Complexity of a PRAM Algorithm.....	11
2.1.3 The PRAM vs Network Models	12
2.2 The Hypercube Multiprocessor	13
2.2.3 PRAM Simulation.....	16
2.2.4 Measuring the Complexity of a Hypercube Algorithm	16
2.2.5 The Connection Machine.....	17
3 Basic Hypercube Operations	18
3.1 Basic $O(1)$ Operations.....	18
3.1.1 The ExtractBit Function	19
3.1.2 The FlipBit Function.....	19
3.1.3 The WriteD and ReadD Operations.....	20
3.1.4 The SwapNeighbours Function.....	21
3.2 Basic $O(\log N)$ Operations	21
3.2.1 The Partial Sum Operation.....	21
3.2.2 The Rank Operation.....	22
3.2.3 The Number Operation.....	24
3.3 Data Movement Operations.....	25
3.3.1 The Concentrate Operation.....	25
3.3.2 The Route Operation.....	27
3.3.3 The RouteAndCopy Operation	28
3.3.4 The Reverse Operation.....	30
3.4 Parallel Sorting.....	31
3.5 Block Orientated Operations	32
3.5.1 The IdentifyBlockTail and IdentifyBlockHead Operations.....	33
3.5.2 The BroadcastBlockTail Operation.....	34
3.5.3 The BlockPsum Operation.....	35
3.6 Summary.....	37
4 Implementing Parallel Data Structures on Hypercube Multiprocessors.....	38
4.1 Problem Overview.....	38
4.2 Definitions	41

4.2.1 Ordered h-level Graphs.....	42
4.2.2 The m-way Search Problem for Ordered h-level Graphs	46
4.2.3 Using the m-way Search as a Algorithm Design Methodology.	48
4.3 An $O(\min\{s \log N, \log^2 N\} + h \log N)$ Time Algorithm for the m-way Search Problem on Ordered h-level Graphs.....	49
4.3.1 Initial Configuration.....	50
4.3.2 Algorithm Overview	51
4.3.3 Phase 1 of the M-Way Search Algorithm.....	52
4.3.4 Phase x ($2 \leq x \leq h$) of the M-Way Search Algorithm.....	57
4.4 Extensions.....	62
4.5 Conclusion.....	63
5 Applications of the M-Way Search	64
5.1 Parallel Binary Search.....	64
5.2 The General Next Element Search Problem.....	69
5.2.1 Building a Segment Tree on a Hypercube of Size $O(n \log n)$	72
5.2.2 Implementing Next Element Search Using a Segment Tree.....	75
5.3 The Trapezoidal Map Problem	77
5.4 Triangulation of a Simple Polygon.....	78
5.6 Conclusion.....	79
6 Parallel Fractional Cascading on Hypercube Multiprocessors.....	80
6.1 Problem Overview	80
6.2 Definitions	82
6.3 A Hypercube Implementation of Fractional Cascading.....	85
6.3.1 Initial Configuration.....	86
6.3.2 Algorithm Overview	88
6.3.3 Phase 1 of the Multi Iterative Search Algorithm.....	89
6.3.4 Phase x ($2 \leq x \leq p$) of the Multi Iterative Search Algorithm.....	97
6.4 Conclusion.....	101
7 An Application of Parallel Fractional Cascading.....	102
7.1 Review of Chazelle and Guibas' sequential stabbing algorithm	103
7.2 A parallel Algorithm for Determination of All Intersections of m Lines with a Simple Polygonal Path of Length n	112
7.3 Conclusion.....	116

8 Parallel Branch and Bound.....	117
8.1 Overview.....	118
8.2 Generalized Branch and Bound.....	120
8.3 An Algorithm for Parallel Branch and Bound with Global Information on a Fine-Grained Hypercube Multiprocessor.....	120
8.3.1 Algorithm Overview.....	122
8.3.2 Updating the Back-Up Tree.....	125
8.4 A Best-First Branch and Bound Search Example.....	128
8.4.1 nth Order Solitaire Checkers.....	129
8.4.2 A Best-First B&B Search Procedure for nth Order Solitaire Checkers Problem.....	130
8.5 Conclusion.....	131
9 Conclusion.....	133
9.1 Directions for Future Research.....	135
10 References.....	136

LIST OF FIGURES

Figure 1.1 A PRAM storing a shared data structure in shared memory.....	2
Figure 1.2 Standard processor network interconnection patterns	3
Figure 2.1 A Parallel Random Access Machine (PRAM).....	10
Figure 2.2 Network models of parallel computers.....	13
Figure 2.3 Hypercubes of dimensions $d=1, 2, 3$ and 4.	14
Figure 2.4 A $d=3$ hypercube with the binary representation of processor identifiers.....	14
Figure 2.5 An SIMD hypercube multiprocessor architecture.....	15
Figure 3.1 A hypercube with processor addresses specified.....	20
Figure 3.2 An example of the Psum Operation.....	22
Figure 3.3 Procedure Psum.....	22
Figure 3.4 An example of the Rank operation.....	23
Figure 3.5 Procedure Rank.....	23
Figure 3.6 An example of the Number operation.....	24
Figure 3.7 Procedure Number.....	24
Figure 3.8 An example of the Concentrate operation.....	26
Figure 3.9 Procedure Concentrate.....	26
Figure 3.10 An example of the Route operation.....	27
Figure 3.11 Procedure Route.....	28
Figure 3.12 An example of the RouteAndCopy operation.	29
Figure 3.13 Procedure RouteAndCopy	29
Figure 3.14 An example of the Reverse operation.....	30
Figure 3.15 Procedure Reverse.....	31
Figure 3.16 Register block(i) defines a series of 7 blocks.....	33
Figure 3.17 An example of the IdentifyBlockTail operation.....	34
Figure 3.18 Procedure IdentifyBlockTail.....	34
Figure 3.19 An example of BroadcastBlockTail operation.....	35
Figure 3.20 Procedure BroadcastBlockTail.....	35
Figure 3.21 An example of the BlockPsum Operation	36
Figure 3.22 Procedure BlockPsum.....	36
Figure 4.1 The one-structure-n-queries paradigm implemented on a PRAM	39
Figure 4.2 An ordered h-level graph.....	43

Figure 4.3 A k-nary tree ($k = 2$) as an ordered h-level graph ($h = 4$).....	44
Figure 4.4 Splits and joins	45
Figure 4.5 Inclusion hierarchy of directed graphs.	45
Figure 4.6 The planar acyclic graph, $K_{2,2}$, is not an ordered h-level graph.	46
Figure 4.7 The search path of a single query, q	47
Figure 4.8 The registers required at each processor, $PE(i)$	50
Figure 4.9 Global structure of the m-way search algorithm.....	51
Figure 4.10 A typical situation at the end of a phase.....	52
Figure 4.11 Outline of Phase 1.....	52
Figure 4.12 An illustration of Phase1.....	54
Figure 4.13 Detailed description of procedure SortBySourceSelected.	55
Figure 4.14 Detailed description of procedure MoveVerticesToQueries.....	56
Figure 4.15 Overview of Phase x , $2 \leq x \leq n$	58
Figure 4.16 An illustration of Phase 2.....	59
Figure 4.17 Details of procedure OrderQueriesByNextVertex	60
Figure 5.2 Constructing a binary search tree using the m-way search.....	66
Figure 5.3 Next element search in a slab	68
Figure 5.4 The next element search problem.....	70
Figure 5.5 A segment tree with node lists.	71
Figure 5.7 Each node list is converted into a small binary tree.	76
Figure 5.8 The Trapezoidal Map of a set of line segments	77
Figure 5.9 A simple polygon before and after triangulation	78
Figure 6.1 An example catalog graph..	81
Figure 6.2 An example of a monotonic catalog graph.....	84
Figure 6.3 Monotone graphs in relation to several other classes	85
Figure 6.4 A Hive Graph (An example of a non monotonic data structure).....	85
Figure 6.5 The registers required at each processor $PE(i)$	87
Figure 6.6 Global structure of the Multi Iterative Search algorithm.....	88
Figure 6.7 A typical situation at the end of a Phase.....	89
Figure 6.8 Outline of Phase 1.	89
Figure 6.9 An illustration of Phase1	91
Figure 6.10 Sketch of Procedure MoveVerticesToQueries.	91
Figure 6.11 An example trace of MoveVerticesToQueries.....	92
Figure 6.12 Sketch of Procedure SelectCatalogs.	93
Figure 6.13 An example trace of Procedure SelectCatalogs.	94

Figure 6.14 Sketch of Procedure SearchCatalogsForQueries.....	95
Figure 6.15 An example trace of SearchCatalogsForQueries.....	95
Figure 6.16 Sketch of Phase x , $2 \leq x \leq p$	97
Figure 6.17 Example of Phase $x=2$	99
Figure 6.18 Sketch of Procedure OrderQueriesByNextVertex.....	99
Figure 7.1 Intersections of a line with a simple polygonal path of length n	103
Figure 7.2 The slopes of edges of a convex polygon.....	104
Figure 7.3 Intersection testing by construction of parallel supporting lines.....	105
Figure 7.4 A convex hull decomposition tree.....	106
Figure 7.5 The tree T' (shown in gray) with k leaves is subtree of T	108
Figure 7.6 Two convex hulls have at most two common tangents.....	109
Figure 7.7 Edges (depicted in bold) are stored only once at the highest point they appear.....	110
Figure 7.8 Three intersection cases	111
Figure 7.9 The multiple stabbing problem.....	112
Figure 7.10 Identifying the merge points of two convex hulls.....	113
Figure 7.11 An inorder traversal of the subtree T' of T	114
Figure 7.12 Defining g to be an inorder traversal of the tree	115
Figure 8.1 A current back-up tree (bold nodes).	121
Figure 8.2 The registers required at each processor $PE(i)$	122
Figure 8.3 The level ordering of the current back-up tree.....	123
Figure 8.4 The global structure of algorithm Branch-And-Bound.....	123
Figure 8.5 Procedure UpdateTree.....	126
Figure 8.6 Calculating a node's new address in the back-up tree.	127
Figure 8.7 The initial setup of a n th Order Solitaire Checkers game ($n=5$).....	129

LIST OF TABLES

Table 2.1 The four variants of the PRAM.....	10
Table 9.1 Summary of results	134

1 INTRODUCTION

The use of efficient data structures is at the heart of algorithm design for the standard sequential model of computing. The literature on the design of efficient data structures is extensive and a significant amount of experience with data structures has been accumulated in both academic and industrial settings. Data structure abstractions are central to the design of efficient algorithms, be they in a sequential or parallel setting. For parallel algorithm design, many researchers therefore prefer to use the CREW-PRAM model of computation over other models, since this model allows the implementation of data structures in a straightforward manner. In this model an arbitrary number of processors, N , are connected to an infinitely large shared memory; each processor having concurrent read access to every memory location. Note that the structure of the PRAM memory resembles that of the standard sequential machine memory and can therefore be used to implement data structures in a similar manner. The usual approach to implementing data structures on the PRAM is to have all processors collectively build the data structure in shared memory. All individual processors can then search the data structure independently of each other (using the concurrent read operation) for $m=O(N)$ given search queries; see Figure 1.1.

Since these query processes need only read access to the data structure, each processor can execute (in parallel) its query independent of the others, in the standard sequential manner. Each processor can be considered to have its own copy of the entire data structure solely for the purposes of answering its query. We will refer to the simultaneous processing of n queries on a shared data structure as the one-structure- n -queries paradigm.

Figure 1.1 A PRAM storing a shared data structure in shared memory

This paradigm allows researchers to apply numerous results about sequential data structures to the design of efficient PRAM-algorithms. It has been extensively used in the implementation of well established data structures; see [ACG], [ACGOY], [Goo], and [DK].

Unfortunately, the PRAM is an idealized theoretical model of parallel computing, and not a practical parallel architecture. In situations where more than a small number of processors (≈ 1000) are required, the PRAM is not directly realizable in current or foreseeable VLSI technology [Ull p. 227]. The problem lies, primarily, in implementing the PRAM's shared memory. Implementing shared memory that supports $O(1)$ concurrent read access requires that a dedicated wire or other physical communication link exists between each memory cell and each processor. Since a PRAM can have an unbounded number of processors, each shared memory cell must support unbounded communication link fan-out. This is not possible with current (or foreseeable) VLSI technology. Due to practical physical limitations, like the number of wires that can be joined to a single point,

many developers of parallel computers have adopted processor network architectures, instead of ones based on the PRAM.

Processor networks are comprised of a large number of identical processors, each equipped with a small local memory. Each processor is connected via a communication link to a small number of other processors in a regular interconnection pattern thereby avoiding the unbounded communication link fan-out problem. Figure 1.2 illustrates several interconnection patterns commonly used in parallel computers based on processor network models.

Figure 1.2 Standard processor network interconnection patterns

For processor networks the parallel execution of independent queries on one joint data structure is obviously not as straight forward as in the PRAM case. The problem lies with the distributed nature of memory on processor networks. Lacking the shared memory of the PRAM any joint data structure on a processor network must be divided into small nodes of constant size; each node being stored on a different processor. Having distributed the nodes of a data structure across a processor network, one is left with a difficult task: How can the now distributed data structure be efficiently accessed by all processors in parallel? If a large number of processors require the same node of the data structure at the same time, how can these requests be handled efficiently; that is without queuing requests for the

required node at the processor that stores it? If the one-structure-n-queries paradigm is to be supported on a processor network, an access scheme must be developed that allows an arbitrary number of processors to access the same node of a joint data structure at the same time.

Due to the difficulty of solving the general data structure problem described above researchers have largely abandoned the idea of using data structures on processor networks. Processor network algorithms are primarily concerned with solving problem specific routing and collision avoidance problems and have tended to use only very simple data structures, if any. While this approach has led to a number of efficient processor network algorithms, the underlying routing and collision avoidance schemes developed in these algorithms have been largely based on ad hoc solutions to the individual problems. Consequently, these algorithms have tended to be inelegant, and difficult to design and implement. Working at this low level of abstraction, one which exactly models the hypercube's architecture, algorithm designers have been deprived of the standard data structure abstractions that form the basis of most algorithm design for sequential machines and the PRAM.

Another, more elegant approach to algorithm design on processor networks is to utilize data structures by simulating PRAM-algorithms that implement data structures in the one-structure-n-queries paradigm. Unfortunately, despite the elegance and relative ease of implementation of algorithms developed in this way, they tend to be much less efficient than algorithms designed directly for a specific processor network.

In this thesis, we consider a widely used processor network model: the fine-grained hypercube multiprocessor. A hypercube multiprocessor of size N is a set of N processing

elements, $PE(i)$, $0 \leq i \leq N-1$, where two processors $PE(i)$ and $PE(j)$ are connected by a communication link if and only if the binary representations of i and j differ in exactly one bit.

Hypercube multiprocessors are typically classified as being coarse-grained or fine-grained depending on the number of processors they support. In this thesis we will refer to hypercube multiprocessors with $N \geq 2^{10}$ processors as being fine-grained.

The best known example of a fine-grained hypercube multiprocessor is the Connection Machine from Thinking Machines Inc., of Cambridge Massachusetts. The largest model of the Connection Machine, the CM2, supports 2^{16} (65,536) physical processor, each capable of supporting 2^6 virtual processors, for a total of 2^{22} or $N = 4,194,304$ virtual processors. All of the algorithms presented in this thesis have been designed for fine-grained hypercube multiprocessors; therefore the Connection Machine can be always thought of as the target machine. See [BHRY] for Connection Machine implementations of some of many of the algorithms described in this thesis.

For the fine-grained hypercube architecture considered in this thesis, the standard PRAM simulation incurs an $O(\log^2 N)$ time overhead, for each parallel memory access on the PRAM [NS]. On the other hand, for many problems there exists direct hypercube algorithms which incurs only an $O(\log N)$ slowdown compared with the respective PRAM method [BM], [MM], [MS], [SZ], [IP], [Cham]. Clearly, we would like our hypercube algorithms to incur only the $O(\log N)$ time slowdown achieved by direct hypercube algorithms, while maintaining the elegance and relative ease of design and implementation afforded by algorithms based on the PRAM simulation approach.

In this thesis we will show, that for hypercube multiprocessors, it is possible to design elegant yet efficient algorithms, based on parallel implementations of advanced data

structures. We will present a general technique for the design and implementation of parallel versions of a large class of standard sequential data structures. In demonstrating this general technique we will present new algorithms for a range of problems and show that these algorithms are extremely efficient; i.e. as good as or better than the best known direct hypercube algorithms, solving the same problems.

The approach which we will take is to dynamically divide up data structures and distribute them over the hypercube network. Our central result will be an algorithm that allows a data structure, distributed over N processors, to be efficiently searched, by up to N queries, independently and in parallel. This result demonstrates that the one-structure- n -queries paradigm, that explains many researchers' preference for the PRAM model of computing, can also be efficiently supported on hypercube multiprocessors.

We will define a class of graphs called ordered h -level graphs, which includes many standard data structures, and show that on such graphs, stored on hypercube multiprocessors, $m=O(N)$ search processes can be efficiently executed independently and in parallel. We call this procedure the m -way search. In particular we show that an ordered h -level graph, with n nodes of constant size and s sources, can be searched by m independent search queries (with maximum path length h) in parallel on a hypercube multiprocessor of size N , $N=\max\{n,m\}$, in time $O(\min\{s \log N, \log^2 N\} + h \log N)$.

Our algorithm, published in [DR1] [DR2], will allow an arbitrary number of queries to access the same node at the same time. This is essential in implementing the one-structure- n -queries paradigm and cannot, for example, be obtained by any method that statically assigns nodes to processors¹. To demonstrate the use of the m -way search algorithm we

¹ One commonly used method of this type is to directly embed a data structure into the hypercube topology, via a graph embedding [LS]. Note that whereas such embeddings are sometimes useful they do not solve the problem of multiple simultaneous query access.

present algorithms for several applications in computational geometry, including an $O(\log^2 N)$ time algorithm for triangulation of simple polygons.

Having shown how the one-structure-n-queries paradigm can be realized by the m-way search algorithm, we proceed to generalize this result. Our first major generalization is to present a parallel version of fractional cascading [CG1] for fine-grained hypercube multiprocessors. A parallel version of this important data structuring technique allows us to efficiently address, in the parallel setting many applications in computational geometry.

Our parallel version of fractional cascading, published in [DFR1], is based on an enhanced version of the m-way search algorithm which extends the power of the m-way search algorithm in several ways. Firstly it removes several restrictions on the type of graphs that can be used to represent data structures. It also reduces restrictions on the type of paths queries can follow during the search process. Lastly, it relaxes the constant space restriction we previously imposed on the size of individual nodes in our shared data structures. Instead of insisting that each node have only constant space, it allows individual nodes to have arbitrary space as long as the total space allocated is $O(N)$.

Both the m-way search algorithm and parallel fractional cascading address problems of searching in static data structures. Our final extension is to address the problem of representing and using dynamic data structures on hypercube multiprocessors. If we wish to implement parallel data structures supporting insert and delete queries, in addition to search queries, we must address significantly more complex problems. In the closing chapter of this thesis, we present a technique for creating and maintaining a restricted class of dynamic parallel data structures, namely trees. We apply our technique to the design of branch and bound search algorithms for fine-grained hypercube multiprocessors. Our solution, published in [DFR2], supports the deletion of $d < n$ nodes, and the creation of $f <$

N - n nodes of an n node data structure, in time $O(\log \max\{n, N-d+f\})$, where N is the size of our hypercube.

The thesis is organized as follows: Chapter 2 provides a short introduction to models of parallel computing, while in Chapter 3 we describe standard hypercube data movement and communication operations. In Chapter 4 we present an m -way search algorithm for ordered h -level graphs with applications of this algorithm being given in Chapter 5. In Chapter 6, a parallel version of fractional cascading is presented, while Chapter 7 describes an application of this data structuring technique. Lastly, in Chapter 8, we present an algorithm for parallel branch and bound on fine-grained hypercube multiprocessors. Chapter 9 concludes this thesis with a summary of results and a discussion of possible future work.

2 AN INTRODUCTION TO PARALLEL MODELS OF COMPUTING

This chapter introduces two models of parallel computing: the Parallel Random Access Machine (PRAM), and the hypercube processor network. The PRAM is an idealized theoretical model of parallel computing. A model that, in focusing on the central concerns of parallel computing, allows researchers to develop algorithms in an abstract setting, free of many practical implementation concerns. Unfortunately, despite the PRAM's power as a tool for algorithm design, in situations where more than a small number of processors are required (≈ 1000), the PRAM is not directly realizable in VLSI technology [Ull p. 227].

The hypercube processor network, on the other hand, is a viable machine architecture even when many thousands of processors are required. We will refer to machines based on this model as hypercube multiprocessors. In the remainder of this chapter we will describe both parallel computation models in detail. Although most of this thesis will focus on algorithms and design techniques for hypercube multiprocessors, it is important to understand the PRAM model, for it is by comparison to algorithms designed for this model that we will evaluate our hypercube results.

2.1 THE PARALLEL RANDOM ACCESS MACHINE

The PRAM is comprised of a sequence $PE(0), \dots, PE(N-1)$ of N processors where each processor, $PE(i)$, has an infinite local memory, $M(i)$, and access to a infinite shared memory, S ; see Figure 2.1 for an illustration.

Figure 2.1 A Parallel Random Access Machine (PRAM)

In this model, processor communication takes place through the infinite shared memory. There are four primary variants of the model based on how processors may access locations in the shared memory, as shown in Table 2.1.

	Exclusive Read (ER)	Concurrent Read (CR)
Exclusive Write (EW)	EREW-PRAM	CREW-PRAM
Concurrent Write (CW)	ERCW-PRAM	CRCW-PRAM

Table 2.1 The four variants of the PRAM

On an Exclusive-Read (ER) PRAM, a location in shared memory can be read by only one processor at any one time. On the Concurrent-Read (CR) variant, several processor (up to N) can read the same location in shared memory at the same time. Just as read

operations can be concurrent or exclusive, so can write operations. The Exclusive-Write (EW) PRAM allows only one processor to write to a location in shared memory in a time interval. In the Concurrent-Write (CW) PRAM, any number of processors (up to N) can simultaneously write to a single location in shared memory. A concurrent-write operation is performed under a conflict resolution rule, which is used to determine what value is to be stored in cases where several values are written to the same location. Combining the different read and write operations four basic PRAM models are defined; the EREW-PRAM, the ERCW-PRAM, the CREW-PRAM and, lastly, the CRCW-PRAM. By far the most popular of these models is the CREW-PRAM.

2.1.2 Measuring the Complexity of a PRAM Algorithm

In parallel computing, just as in sequential computing, it is important to have a general framework within which to evaluate the efficiency of algorithms. In the parallel setting there are typically three scarce resources to be considered: time, space and processors. In this thesis the size of a problem will be generally defined as n , and N will represent the number of processors being used. We will restrict the amount of data that can be communicated in a single read or write operation to a word size proportional to the log of the size of our parallel machine.

The cost of a PRAM-algorithm will be measured in terms of two resources: $P(n)$, the number of processors required, and $T(n)$, the time required to solve the problem. In general, we will define the overall cost of a PRAM-algorithm as the time/processor product, $\text{Cost}(n) = P(n) * T(n)$. This product is equal to the total number of $O(1)$ steps executed by the algorithm in the worst case.

We will want to compare the efficiency of our PRAM-algorithms with those solving the same problems on the sequential Random Access Machine (RAM) model. Let $T_s(n)$ be

the sequential time complexity of a given problem. A PRAM-algorithm for the same problem is asymptotically optimal if and only if $T(n) = O(\frac{T_s(n)}{P(n)})$.

Ideally we would always like to find an optimal algorithm for a given problem; unfortunately this is not always possible. In general, it is not unreasonable that there be some overhead involved in coordinating the efforts of a large number of processors (or people for that manner) that are working together to solve a problem. Typically, PRAM-algorithms have been defined as efficient if they are within a poly-log factor of optimal; that is, $T(m) = O(\frac{T_s(n)(\log^k n)}{P(n)})$.

Another way to examine the efficiency of a parallel algorithm is in terms of speedup. Speedup is usually defined as $\frac{T_s(n)}{T(n)}$. Clearly the larger the speedup the better the algorithm. Ideally we would like a linear speedup, $\frac{T_s(n)}{T(n)} = N$, as is achieved by optimal parallel algorithms. Note that no parallel algorithm can achieve a better than linear speedup, since such an algorithm would itself immediately imply a faster sequential algorithm.

2.1.3 The PRAM vs Network Models

As stated earlier, the PRAM represents an idealized theoretical model of parallel computing. Despite this several small-scale parallel computers have been successfully built based on this architecture [ECC], [GKLS], [BBN]. Unfortunately, as noted early, the PRAM architecture is not realizable on a large scale using VLSI technology. The problem lies primarily in implementing the PRAM's shared memory. Implementing shared memory that supports $O(1)$ concurrent read access requires the existence of a dedicated wire or other physical communication link between each memory cell and each processor. Since a PRAM can have an unbounded number of processors each shared memory cell must support unbounded communication link fan-out. This is not possible with current (or foreseeable) VLSI technology; for as N grows the number of wires or outdegree of each

memory cell must grow and soon a physical limit on the number of wires that can be joined to any memory cell is reached. Parallel computers based on a PRAM architecture are therefore viable for small values of N , up to 100 or maybe 1000, but not for very large values of N .

Any parallel machine architecture that requires the implementation of a complete graph on N vertices, as is the case for the PRAM's shared memory, will suffer from the same implementation problems as the PRAM. One approach to avoiding these problems is to design a parallel machine architecture in which processors are interconnected in some pattern; each processor being connected to only a small number of other processors. Such parallel machine models are commonly referred to as network models; see Figure 2.2.

Figure 2.2 Network models of parallel computers

2.2 THE HYPERCUBE MULTIPROCESSOR

In this thesis, we consider a very commonly used processor network: the fine-grained hypercube multiprocessor. A d -dimensional hypercube multiprocessor consists of $N = 2^d$ processors connected in a hypercube topology. Figure 2.3 illustrates hypercube networks of dimensions, $d = 1, 2, 3$ and 4.

Figure 2.3 Hypercubes of dimensions $d= 1, 2, 3$ and 4 .

Each processor, $PE(i)$, in a hypercube network has a unique identifier i , $0 \leq i \leq N-1$. Two processors $PE(i)$ and $PE(j)$ are connected by a communication link if the binary representations of i and j differ in exactly one bit. Figure 2.4 illustrates a three dimensional hypercube together with the binary representation of each processor's identifier.

Figure 2.4 A $d=3$ hypercube with the binary representation of processor identifiers

Hypercube multiprocessors (and other network processors) are typically classified as being coarse-grained or fine-grained depending on the size of N . In this thesis we will consider algorithms for fine-grained hypercube multiprocessors. For our purposes we will define a parallel machine as fine-grained if N , the number of processors, is greater than 2^{10} or 1024.

Another way of classifying parallel machines is in terms of how many different instruction streams are being executed at once. On an SIMD (Single-Instruction-Multiple-Data) machine at any point in time all processors are executing the same instruction. Typically SIMD machines have a "front-end" processor that runs a single program. This processor broadcasts instructions to all of the other processors, who execute the instruction simultaneously and in parallel, relative to data in their local memory. An SIMD hypercube machine architecture is illustrated in 2.5.

Figure 2.5 An SIMD hypercube multiprocessor architecture

An alternative to the SIMD approach is to allow any two processors to execute different instructions simultaneously and in parallel. Machines that adopt this approach are called MIMD (Multiple-Instruction-Multiple-Data) machines.

All of the algorithms presented in this thesis follow the SIMD approach. This approach appears particularly suitable for fine-grained architectures in which algorithms for problems of size n are typically designed to run on machines with $n=N$ processors.

2.2.3 PRAM Simulation

Earlier we stated that the PRAM could not be directly realized in VLSI hardware. This fact, however, does not preclude us from simulating a PRAM on a processor network. Unfortunately, despite the elegance and relative ease of implementation of algorithms developed in this way, they tend to be much less efficient than algorithms designed directly for a specific processor network. For the fine-grained hypercube architecture considered in this thesis, the standard PRAM simulation incurs an $O(\log^2 N)$ time overhead for each parallel memory access on the PRAM [NS]. On the other hand, for many problems there exists direct hypercube algorithms which incur only an $O(\log N)$ slowdown compared with the respective PRAM method [BM], [Cham], [IP], [MM], [MS], [SZ]. Clearly, we would like our hypercube algorithms to incur only the $O(\log N)$ time slowdown achieved by direct hypercube algorithms, while maintaining the elegance and relative ease of design and implementation afforded by algorithms based on the PRAM simulation approach.

One of the big open problems in the domain of hypercube algorithms is to find an optimal $O(\log N)$ time general simulation of the PRAM. Such an algorithm would be essentially equivalent to an $O(\log N)$ time sorting algorithm for the hypercube. Research on this problem has been ongoing since Batchers's $O(\log^2 N)$ result in 1968 [Bat], without any success to date.

2.2.4 Measuring the Complexity of a Hypercube Algorithm

The cost of a hypercube algorithm is measured in the same way as that of a PRAM-algorithm; $\text{Cost}(n) = P(n) * T(n)$, where $P(n)$ is the number of processors used, and $T(n)$ is the time. Clearly for most problems the PRAM time complexity, $T_p(n)$, will be strictly less than $T_h(n)$, the hypercube time complexity, primary due to communication delay. Recall that on a hypercube multiprocessor there is a communication delay of $O(\log N)$ for sending

a message between two arbitrary processors, corresponding to the diameter of the network. On the PRAM, through the use of the shared memory, interprocessor communication is an $O(1)$ time operation. Hence, in general we will define a hypercube algorithm for some problem as highly efficient if $T_h(n) = O(T_p(n) * \log n)$, where $T_p(n)$ is the time complexity of an optimal PRAM-algorithm for the problem and $T_h(n)$ is the time complexity of the hypercube algorithm. Since algorithms based on PRAM simulation incur an $O(\log^2 N)$ overhead, they clearly do not fall into the class of highly efficient hypercube algorithms.

2.2.5 The Connection Machine

As stated earlier, this thesis focuses on algorithms for fine-grained SIMD hypercubes. An example of such a machine is the Connection Machine from Thinking Machines Inc., of Cambridge Massachusetts. The largest model of the Connection Machine, the CM2, supports 2^{16} (65,536) physical processors, each capable of supporting 2^6 virtual processors, for a total of 2^{22} or $N = 4,194,304$ virtual processors. Since all of the algorithms presented in this thesis have been designed for fine-grained hypercube multiprocessors, the Connection Machine can be always thought of as the target machine². Some of the algorithms and basic operations presented in this thesis have been implemented in *LISP for the Connection Machine; See [BHRY] for details.

² Some care must be taken when analyzing the time complexity of *LISP programs since the complexity of individual operations varies dramatically. Some operations, for example local arithmetic, are $O(1)$ time operations, while other "elementary" operations, like the random access read (PREF) operation, require $O(\log^2 N)$ time.

3 BASIC HYPERCUBE OPERATIONS

In this chapter we describe some basic hypercube operations out of which, in later chapters, we will build our algorithms. Most algorithm design for fine-grained SIMD hypercubes is based on these fundamental operations.

In defining our operations we use the following notation. For every register x available at every processor, $x(i)$ refers to register x at processor $PE(i)$. In addition to those registers listed below, the implementation of the operations described in this chapter may require a constant number auxiliary registers, $aux_1(i) \dots aux_z(i)$ where $z = O(1)$.

All register names start with a lower case letter, while the names of operations and global variables are capitalized. For example, an operation that increments the same register on every processor might be referred to as $Increment(count(i), Number)$, where $Increment$ is the name of the operation, $count(i)$ is the register to be incremented and $Number$ is a global variable specifying the size of the increment.

Throughout this thesis the following definitions will hold unless otherwise specified. The size of the problem to be solved is n , while N is the size of the hypercube; we will assume without loss of generality that $n = N$ unless otherwise specified.

3.1 BASIC $O(1)$ OPERATIONS

As stated earlier, this thesis focuses on algorithms for fine-grained SIMD hypercube multiprocessors. In such machines, individual processors are small and capable of only a limited amount of local processing. Such machines derive their power not from the power of individual processors, but rather from the use of a large number of simple processors. One can expect a single fine-grained hypercube processor to perform integer arithmetic

operations, some floating point operations and bitwise operations, conditionals statements, and little else. All of these local operations can be considered as requiring $O(1)$ time.

In addition to local operations, the fundamental communication operations also take $O(1)$ time. These operations permit neighbouring processors in the hypercube topology to communicate. The $O(1)$ time complexity of both arithmetic and basic communication operations is defined in terms of a maximum word length of $O(\log N)$ bits. Both messages communicated between processor and values used in arithmetic operations should never be longer than this word length if $O(1)$ time complexity is desired. The following sections describe the basic $O(1)$ communication operations and two commonly used bitwise operations.

3.1.1 The ExtractBit Function

The ExtractBit function returns the i^{th} bit of a bit string of length $O(\log N)$. For example, $\text{ExtractBit}(3, 00001000) = 1$. Note that the bit string is labeled from right to left with the least significant bit being labeled 0. For the sake of simplicity, integers will generally be used to specify the bit strings, for example, $\text{ExtractBit}(3, 8) = 1$, since the bit string representation of 8 is 00001000. The ExtractBit function is commonly used in hypercube address calculations; see section 3.2.2 for an example.

3.1.2 The FlipBit Function

The FlipBit function ($\text{FlipBit}(i, s)$) returns a new bit string, s' , which is equal to s except in the i^{th} bit, which has been flipped. Again for the sake of simplicity, integers will generally be used to specify the bit strings, for example, $\text{FlipBit}(3, 8) = 0$, since the bit string representation of 8 is 00001000 and by flipping the 3rd bit we get 00000000.

3.1.3 The WriteD and ReadD Operations

WriteD and ReadD are the basic operations through which all interprocessor communication will be conducted. These operations specify communication between adjacent processors in the hypercube topology. Consider a hypercube with cube addresses as illustrated in Figure 3.1.

Figure 3.1 A hypercube with processor addresses specified

Note that processors that are adjacent in the hypercube topology have addresses that differ in exactly one bit. Communication between two processors whose addresses differ by the i^{th} bit is said to take place along the i^{th} dimension of the hypercube. The ReadD operation ($\text{ReadD}(\text{source}, \text{destination}, \text{dimension})$) specifies that each processor will copy the value of the source register from its neighbour in the specified dimension into its own destination register. The WriteD operation is defined in an analogous way.

Communication between processors in a specified dimension is very fast, i.e. $O(1)$, on hypercube computers, since such communication uses only a direct physical link (one wire) between processors and involves each processor in only one communication task.

3.1.4 The SwapNeighbours Function

The SwapNeighbours function (`swapNeighbours(destination,b,cond)`) swaps the values in register destination between processors with addresses differing in exactly the b^{th} bit, if the condition function, `cond`, evaluates to true. It is defined in terms of three `writeln` operations, much as a swap operation is defined in sequential computing, therefore taking $O(1)$ time.

3.2 BASIC $O(\log N)$ OPERATIONS

In this section we will describe some of the most fundamental hypercube operations. By fundamental we refer to operations that, due to the nature of the hypercube, tend to be a part of almost all algorithm designs.

3.2.1 The Partial Sum Operation

The partial sum or Psum operation [HS] assumes that every processor $PE(i)$ has some value stored in the register `source(i)`. The operation calculates $result(i) := source(0) + source(1) + \dots + source(i)$ for each $PE(i)$; see example depicted in Figure 3.2. This $O(\log N)$ time operation is so frequently used in hypercube algorithms that designers of hypercube computers have tended to provide highly optimized versions of it in their parallel programming languages [TMC].

	Input to Psum Operation															
	PE(0)	...														PE(15)
source(i)	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2
	Output of Psum Operation															
result(i)	1	3	6	10	15	21	28	36	45	53	60	66	71	75	78	80

Figure 3.2 An example of the Psum Operation

The algorithm for partial sum operation or Psum is as follows:

Algorithm Psum:

Input: A register source(i) containing values that can be combined with the + operator.

Output: For every PE(i), $\text{result}(i) := \sum_{0 \leq j \leq i} \text{Source}(j)$.

Method: As given by procedure Psum depicted in Figure 3.3. For proof and further details see [HS] ♦

```

Psum(source(i),result(i))
(1)  Every PE(i): result(i) := source(i)
(2)  For J := 0 to (log2 n)-1 do
(3)    if i ≥ 2J then
(4)      ReadD(result,aux,i-2J)
(3)  Every PE(i): result(i) := aux(i) + result(i)

```

Figure 3.3 Procedure Psum

Lemma 3.1: Operation $\text{Psum}(\text{source}(i), \text{result}(i))$ computes, in time $O(\log N)$, in register $\text{result}(i)$ of every processor PE(i), $\text{result}(i) := \sum_{0 \leq j \leq i} \text{Source}(j)$

3.2.2 The Rank Operation

The Rank operation stores in register, result(i), of each processor for which cond(i) is true the number of processors, j, with smaller processor addresses and for which Cond(j) is true [NS]. In addition, the value of result(i) for all processors with cond(i) = false is set

to the "infinity" parameter, INF. Figure 3.4 shows a typical situation before and after a call to the rank operation.

Figure 3.4 An example of the Rank operation

The Rank algorithm is as follows:

Algorithm Rank:

Input: A boolean register, $\text{cond}(i)$, and an integer value INF.

Output: For every $\text{PE}(i)$, if $\text{cond}(i) = \text{true}$ then $\text{result}(i) = \#$ of $\text{PE}(j)$ such that $\text{cond}(j) = \text{true}$ and $j < i$, else $\text{result}(i) = \text{INF}$.

Method: This algorithm can be defined in terms of a psum on a register in which each selected processor has stored a one, while unselected processors have stored zeros; see Figure 3.5. For proof and further details see [NS] ♦

Rank(result(i),cond(i),INF)
 (1) Every $\text{PE}(i)$: If $\text{cond}(i)$ then $\text{result}(i) := 1$ else $\text{result}(i) := 0$
 (2) Psum(result,result)
 (3) Every $\text{PE}(i)$: If not($\text{cond}(i)$) then $\text{result}(i) := \text{INF}$

Figure 3.5 Procedure Rank

Lemma 3.2: Operation $\text{Rank}(\text{result}(i), \text{cond}(i), \text{INF})$ computes, in time $O(\log N)$, in register $\text{result}(i)$ of every processor $\text{PE}(i)$ with $\text{Cond}(i) = \text{true}$, the number of processors $\text{PE}(j)$ such that $j < i$ and $\text{Cond}(j)$ is true .

3.2.3 The Number Operation

The number operation computes in register $\text{result}(i)$ of every processor $\text{PE}(i)$ the number of processors $\text{PE}(j)$ such that $\text{cond}(j)$ is true. Figure 3.6 shows a typical situation before and after a call to the Number operation.

Figure 3.6 An example of the Number operation

The algorithm for the Number operation is as follows:

Algorithm Number:

Input: A boolean register, $\text{cond}(i)$.

Output: For every $\text{PE}(i)$, $\text{result}(i) = \#$ of $\text{PE}(j)$ ($0 \leq j < N$) such that $\text{Cond}(j) = \text{true}$.

Method: As given in Figure 3.7. For proof and further details see [NS] ♦

```

Number(result(i),cond(i))
(1)  Every PE(i): aux(i) := aux2(i) := result(i) := 0
(2)  For b := 0 to (log2 n) - 1 do
(3)    WriteD(result(i),aux2(i),b)
(4)    result(i) := aux2(i) + result(i)

```

Figure 3.7 Procedure Number

Lemma 3.3: Operation $\text{Number}(\text{result}(i), \text{cond}(i))$ computes, in time $O(\log N)$, in register $\text{result}(i)$ of every processor $\text{PE}(i)$ the number of processors $\text{PE}(j)$ such that $0 \leq j < N$ and $\text{Cond}(j)$ is true .

3.3 DATA MOVEMENT OPERATIONS

The algorithms described in this section are slightly generalized versions of the standard hypercube data movement operations defined in [NS]. All of these operations route data between processors in $O(\log N)$ time using only direct neighbor communication. Each operation takes a source and destination register as two of its arguments; for example $[source(i), destination(i)]$. For the sake of notational convenience, we may specify several source/destination pairs in a single call to a data movement operation. For example, $OperationA([s1(i), d1(i)], c(i))$ followed immediately by $OperationA([s2(i), d2(i)], c(i))$ should be considered equivalent to $OperationA(\{[s1(i), d1(i)], [s2(i), d2(i)]\}, c(i))$.

Several of the basic data movement operations assume that there is a monotonicity relationship between the source and destination of data items. A register $addr(i)$ stores a monotonic sequence of addresses if and only if for all $PE(i)$ and $PE(j)$ $i < j$ implies $addr(i) < addr(j)$.

3.3.1 The Concentrate Operation

The concentrate operation copies selected items from a source register to a destination register such that the i^{th} selected item of the source is copied to the $i^{th}-1$ position in the destination register. The time complexity of this operation is $O(\log N)$ [NS]. Figure 3.8 shows a typical situation before and after a call to the Concentrate operation.

Figure 3.8 An example of the Concentrate operation

The algorithm for this operation can be described as follows:

Algorithm Concentrate:

Input: A boolean register, cond(i), and a register, source(i).

Output: For each PE(i) with cond(i) = true, register source(i) is copied to register destination(R(i)), where R(i) is the rank of cond(i) (see section 3.2.2).

Method: As given in Figure 3.9. For proof and further details see [NS] ♦

```

Concentrate([source(i),destination(i)],cond(i))
(1)  Rank(aux(i),cond(i),INF) :
(2)  If aux(i) ≠ INF then destination(i) := source(i)
(3)  For b := 0 to (log2 n) - 1 do
(4)    if (aux(i) ≠ INF and (extractBit(b,i) ≠ extractBit(b,aux(i)))) then cond(i) := 1
        else cond(i) := 0
(5)    swapNeighbours(destination,b,cond)
(5)    swapNeighbours(aux,b,cond)

```

Figure 3.9 Procedure Concentrate

Lemma 3.4: Operation *Concentrate*([source(i),destination(i)],cond(i)) copies source(i) to destination(Rank(cond(i))) for all PE(i), in time $O(\log N)$.

3.3.2 The Route Operation

The Route operation is a slightly generalized version of the Distribute operation described by Nassimi and Sahni in [NS]. Every processor $PE(i)$ has registers $source(i)$ and $destination(i)$, an address register $addr(i)$, and a boolean condition register $cond(i)$. It is assumed that the addresses stored in $addr(i)$ are strictly monotonic; that is, if $i < j$ then $addr(i) < addr(j)$. The Route operation routes, for every processor $PE(i)$ with $cond(i) = \text{true}$, $source(i)$ to $destination(addr(i))$ ³. The Route operation can be implemented with an $O(\log N)$ time complexity using a Concentrate operation followed by a Distribute operation as described in [NS]. Figure 3.10 show a typical situation before and after a call to the Route operation.

Input to Route Operation																
processor	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
cond(i)	T	F	F	F	T	F	F	T	T	T	F	T	F	F	T	F
address(i)	1	?	?	?	3	?	?	4	7	9	?	12	?	?	14	?
source(i)	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
destination(i)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
⋮																
Output of Route Operation																
destination(i)	a	A	c	E	H	f	g	I	i	J	K	l	L	n	O	p

Figure 3.10 An example of the Route operation

The algorithm for this operation can be described as follows:

Algorithm Route:

Input: A boolean register, $cond(i)$, a register $source(i)$, and an address register $addr(i)$ containing a set of strictly monotonic processor addresses.

³ The Route operation is the inverse of the Concentrate operation.

Output: For each $PE(i)$ with $cond(i) = \text{true}$, register $source(i)$ is copied to register $destination(addr(i))$.

Method: As given in Figure 3.11. For proof and further details see [NS] ♦

```

Route([source(i),destination(i)],addr(i),cond(i))
(1)  Concentrate([source(i),aux2(i)],cond(i))
(2)  Concentrate([addr(i),aux(i)],cond(i))
(3)  If aux(i)  $\neq$  INF then destination(i) := aux2(i)
(4)  For b :=  $(\log_2 n) - 1$  downto 0 do
      (5)  if (aux(i)  $\neq$  INF) and (extractBit(b,i)  $\neq$  extractBit(b,aux(i))) then cond(i) := True
           else cond(i) := False
      (6)  swapNeighbours(aux,b,cond)
      (7)  swapNeighbours(destination,b,cond)

```

Figure 3.11 Procedure Route

Lemma 3.5 *Operation $Route([source(i),destination(i)],addr(i),cond(i))$ given a strictly monotonic sequence of processor addresses in register $addr(i)$ copies $source(i)$ to $destination(addr(i))$ for all $PE(i)$ with $cond(i) = \text{true}$, in time $O(\log N)$.*

3.3.3 The RouteAndCopy Operation

Under the same monotonicity restriction as the Route operation, this operation copies, for every processor $PE(i)$ with $cond(i) = \text{true}$, register $source(i)$ to $destination(k)$, for all k such that $addr(i-1) < k \leq addr(i)$. This operation can be implemented with an $O(\log N)$ time complexity by using a Concentrate operation followed by a Generalize operation as described in [NS]. Figure 3.12 show a typical situation before and after a call to the RouteAndCopy operation.

Input to RouteAndCopy Operation																
processor	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
cond(i)	T	F	F	F	T	F	F	T	T	T	F	T	F	F	T	F
address(i)	1	?	?	?	3	?	?	4	7	9	?	12	?	?	14	?
source(i)	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
destination(i)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

Output of RouteAndCopy Operation																
destination(i)	A	A	E	E	H	I	I	I	J	J	L	L	L	O	O	p

Figure 3.12 An example of the RouteAndCopy operation.

The algorithm for this operation can be described as follows:

Algorithm RouteAndCopy:

Input: A boolean register, $\text{cond}(i)$, a register $\text{source}(i)$, and an address register $\text{addr}(i)$ containing a set of strictly monotonic processor addresses.

Output: For each $\text{PE}(i)$ with $\text{cond}(i) = \text{true}$, register $\text{source}(i)$ is copied to registers $\text{destination}(\text{addr}(i-1) + 1)$ to $\text{destination}(\text{addr}(i))$.

Method: As given in Figure 3.13. For proof and further details see [NS] ♦

RouteAndCopy([source(i),destination(i)],addr(i),cond(i))	
(1)	Concentrate([source(i),aux2(i)],cond(i))
(2)	Rank(aux(i),cond(i),INF)
(3)	If $\text{aux}(i) \neq \text{INF}$ then $\text{destination}(i) := \text{aux2}(i)$
(4)	For $b := (\log_2 n) - 1$ downto 0 do
(5)	if $(\text{aux}(i) \neq \text{INF})$ and $(\text{extractBit}(b,i) \neq \text{extractBit}(b,\text{aux}(i)))$ then cond(i) := 1 else cond(i) := 0
(6)	swapNeighbours(destination(i),b,cond(i))

Figure 3.13 Procedure RouteAndCopy

Lemma 3.6: Operation $\text{Route}([source(i),destination(i)],addr(i),cond(i))$ given a strictly monotonic sequence of processor addresses in register $\text{addr}(i)$ copies $\text{source}(i)$ to $\text{destination}(\text{addr}(i))$ for all $\text{PE}(i)$ with $\text{cond}(i) = \text{true}$, in time $O(\log N)$.

3.3.4 The Reverse Operation

This operation reverses the contents of the source register for the sequence of processors between PE(Start) and PE(End) placing the result in register destination(i). For every PE(i) with $\text{Start} \leq i \leq \text{End}$, this operation copies source(i) to register destination(j) where $j := \text{Start} + \text{End} - i$. Note that reversing, in the entire hypercube, a sequence of n values (each stored in one processor) corresponds to routing each value stored at processor PE(i) to processor PE(i'), where i' is obtained from i by inverting all bits in its binary representation. Hence, the Reverse operation can be implemented in time $\log(n)$ by first concentrating the sequence to be reversed, reversing it using the entire hypercube, and lastly using the Route operation to move the reversed sequence to the correct set of destination registers. Figure 3.14 show a typical situation before and after a call to the Reverse operation.

Figure 3.14 An example of the Reverse operation

The algorithm for this operation can be described as follows:

Algorithm Reverse:

Input: A source register source(i), and processor addresses Start and End specifying the beginning and the end of the sequence of values to be reversed, respectively.

Output: A destination register $\text{destination}(i)$, where $\text{destination}(i) := \text{source}(\text{Start} + \text{End} - i)$ for all $\text{PE}(i)$: $\text{Start} \leq i \leq \text{End}$.

Method: As given in Figure 3.15. Note that the Reverse operation is composed of calls to $O(\log N)$ operations and a $O(\log N)$ loop over $O(1)$ operations, we therefore conclude that the entire operation can be completed in $O(\log N)$ time ♦

```

Reverse([source(i),destination(i)],Start,End)
(1) destination(i) := aux(i) := source(i)
(2) For b := (log2 n) - 1 downto 0 do
    (3) swapNeighbours(aux(i),b,True)
(4) Every PE(i): if n-End ≤ i ≤ n-Start then
    addr(i) := (Start + i - n + End + 1) else addr(i) := INF
(5) Route([aux(i),destination(i)],addr(i),addr(i) ≠ INF)

```

Figure 3.15 Procedure Reverse

Lemma 3.7: *Operation $\text{Reverse}([source(i),destination(i)],\text{Start},\text{End})$ copies $\text{source}(\text{Start} + \text{End} - i)$ to $\text{destination}(i)$ for all i such that $\text{Start} \leq i \leq \text{End}$ in $O(\log N)$ time.*

3.4 PARALLEL SORTING

Parallel sorting has been extensively investigated for all models of parallel computing⁴. Cole recently described [Col] an optimal $O(\log N)$ time sorting algorithm for the CREW-PRAM and EREW-PRAM models. Unfortunately, obtaining an optimal sorting algorithm for the hypercube multiprocessor, remains an important open problem. To date, the best known general sorting algorithms for the hypercube model of computation have a time complexity of $O(\log^2 N)$ [Pla].⁵

In this section, we will briefly describe just one of the many $O(\log^2 N)$ sorting algorithms for the hypercube; the Bitonic sort. The Bitonic Sort, first described in [Bat] for

⁴ For a good survey of parallel sorting algorithms see S. Akl's monogram [Akl].

⁵ Some researchers [Akl] have reported sorting algorithms with $O(\log N)$ time complexity, but they are restricted to the use of a sublinear number of processors or the pipeline hypercube model.

sorting networks, is one of the best $O(\log^2 N)$ sorting algorithms for the fine-grained hypercube multiprocessor [HS]. It is a simple algorithm, with very low constants; it also defines a sub procedure, the Bitonic Merge, which is useful outside the context of general sorting. The Bitonic Sort, BitonicSort(source(i), destination(i), key(i)), given an unsorted source register source(i), permutes the contents of the source register with respect to the key register, key(i), and creates a sorted sequence in the destination register destination(i). The Bitonic Sort is a parallel version of the merge sort that recursively applies $O(\log N)$ Bitonic Merge operations to unsorted sequences.

A bitonic sequence is defined as a cyclic shift of a unimodal sequence. The Bitonic Merge operation, BitonicMerge(source(i), destination(i), key(i), Left, Peak, Right), converts with respect to key, key(i), a bitonic sequence given in a register source(i), into a sorted sequence stored in register destination(i) in time $O(\log N)$. We will use the Bitonic Merge operation outside the context of sorting. When doing so we will restrict ourselves to applying this operation to a particular bitonic sequence consisting of an increasing sequence starting at some position Left, and ending at position Peak, followed by a decreasing sequence starting at Peak+1 and ending at position Right.

3.5 BLOCK ORIENTATED OPERATIONS

Block operations are commonly used in hypercube algorithms. These operations identify or perform operations with respect to groups of contiguous processors that share some property. Most commonly these blocks are defined with respect to the contents of a specified register. Figure 3.16 shows how a register block(i), can define a sequence of blocks.

Figure 3.17 An example of the IdentifyBlockTail operation

The algorithm for this operation can be described as follows:

Algorithm IdentifyBlockTail:

Input: A register $\text{block}(i)$ defining a series of blocks.

Output: A register $\text{tail}(i)$ which equals the register $\text{block}(i)$ for a processor $\text{PE}(i)$ if and only if $\text{PE}(i)$ is not the tail of a block.

Method: As given in Figure 3.18 ♦

IdentifyBlockTail($\text{block}(i), \text{tail}(i)$) (1) $\text{Route}([\text{block}(i), \text{tail}(i)], i-1, i \neq 0)$ (2) $\text{tail}(N-1) := \text{tail}(N-1) + 1$
--

Figure 3.18 Procedure IdentifyBlockTail

Lemma 3.8: Operation $\text{IdentifyBlockTail}(\text{block}(i), \text{tail}(i))$ identifies the tail of each block defined by register $\text{block}(i)$ in $O(\log N)$ time. Operation $\text{IdentifyBlockHead}(\text{block}(i), \text{head}(i))$ identifies the head of each block defined by register $\text{block}(i)$ in $O(\log N)$ time.

3.5.2 The BroadcastBlockTail Operation

For each $\text{PE}(i)$, the BroadcastBlockTail operation assigns to register $\text{endOfBlock}(i)$ the address of the last processor in its block. The time complexity of this operation is $O(\log N)$, since it can be implemented by an IdentifyBlockTail operation followed by a

RouteAndCopy operation. Figure 3.19 show a typical situation before and after a call to the IdentifyBlock operation.

Figure 3.19 An example of BroadcastBlockTail operation

The algorithm for this operation can be described as follows:

Algorithm BroadcastBlockTail:

Input: A register block(i) defining a series of blocks.

Output: A register endOfBlock(i) which contains the address of the last processor in the same block as PE(i).

Method: As given in Figure 3.20 ♦

```

BroadcastBlockTail(block(i),endOfBlock(i))
(1)  IdentifyBlockTail(block(i);aux(i))
(2)  aux2(i) := i
(2)  RouteAndCopy([aux2(i),endOfBlock(i)],i,block(i)≠aux(i))

```

Figure 3.20 Procedure BroadcastBlockTail

Lemma 3.9: Operation *BroadcastBlockTail(block(i),endOfBlock(i))* sets *endOfBlock(i)* of every processor *PE(i)* to the address of the last processor in the same block as *PE(i)* in time $O(\log N)$.

3.5.3 The BlockPsum Operation

This operation performs a partial sum within blocks. Upon completion of this operation every *PE(i)* in a block, *B*, will store in register *result(i)* the partial sum of register *source(j)*,

$j \in B$. The BlockPsum operation can be implemented by a Psum operation followed by a RouteAndCopy operation. Hence its time complexity is $O(\log N)$. Figure 3.21 shows a typical situation before and after a call to the BlockPsum operation.

Input to BlockPsum Operation																
	PE(0)				...				PE(15)							
block(i)	9	9	9	9	2	2	2	8	3	3	2	9	9	0	0	0
source(i)	1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2

Output of BlockPsum Operation																
result(i)	1	3	6	10	5	11	18	8	9	17	7	6	11	4	7	9

Figure 3.21 An example of the BlockPsum Operation

The algorithm for this operation can be described as follows:

Algorithm BlockPsum:

Input: A register block(i) defining a set of blocks and a source(i) register.

Output: For every PE(i), $\text{result}(i) := \sum_{\text{head} \leq j \leq \text{tail}} \text{Source}(j)$, where head and tail are the addresses of the first processor and last processors in the same block as PE(j).

Method: As given in Figure 3.22. BlockPsum is a $O(\log N)$ time operation since it is comprised of a constant number of $O(\log N)$ time operations. ♦

BlockPsum(source(i),result(i),block(i))	
(1)	Psum(source(i),fullSum(i))
(2)	BroadcastBlockTail(block(i),tail(i))
(3)	IdentifyBlockHead(block(i),head(i))
(5)	RouteAndCopy([fullSum(i)-source(i),headSum(i)],tail(i),head(i) ≠ block(i))
(6)	result(i) := fullSum(i) - headSum(i)

Figure 3.22 Procedure BlockPsum

Lemma 3.10: Operation BlockPsum calculates the partial sum of register source(i) within each block defined by block(i) and stores the result in result(i), in $O(\log N)$ time.

3.6 SUMMARY

In this chapter we have described the fundamental hypercube operations out of which we will build our algorithms. We will use these operations repeatedly in the following chapters. See [BHRY] for code that implements many of these operations in *Lisp for a CM2 Connection Machine. We conclude with the following theorem.

Theorem 3.1: *The basic hypercube operations for data movement, block identification, merging, and ranking can be performed in $O(\log N)$ time. Sorting on the hypercube requires $O(\log^2 N)$ time.*

Proof: *By Lemma 3.1 through 3.10.*

4 IMPLEMENTING PARALLEL DATA STRUCTURES ON HYPERCUBE MULTIPROCESSORS

In this chapter, we study the general problem of implementing parallel versions of standard sequential data structures on fine-grained hypercube multiprocessors. We will be primarily interested in supporting multiple concurrent queries on data structures represented as directed graphs. Despite the fact that data structures are at the heart of algorithm design for parallel random access machines (PRAM) and sequential models of computing, only trivial instances have, so far, been used in the design of fine-grained hypercube algorithms. To understand the difficulties in using data structure abstractions on fine-grained hypercubes one must consider the differences between the hypercube multiprocessor and the PRAM.

4.1 PROBLEM OVERVIEW

One of the main differences, besides the difference in communication delay, between the PRAM and the hypercube processor network is that the PRAM has one large (shared) memory similar to a standard sequential computer, whereas the hypercube has its memory divided into pieces of constant size distributed over the network.

The fact that the PRAM memory resembles the structure of the standard sequential machine memory has been extensively used for the design of efficient PRAM-algorithms. It allows the implementation, on a PRAM, of well established data structures like segment trees [ACG], [ACGOY], [Goo] or subdivision hierarchies [DK], in such a way as to support multiple concurrent queries. Once such a data structure has been built in the shared memory of a concurrent read PRAM all processors can, in parallel, search in the data structure to answer a query in the standard manner. In effect, due to the use of the PRAM

concurrent read operation, each processor $PE(i)$ answering a query q_i can be considered to have its own copy of the entire data structure, as illustrated in Figure 4.1. We refer to this as the one-structure-n-queries paradigm.

Figure 4.1 The one-structure-n-queries paradigm implemented on a PRAM

For processor networks, the parallel execution of independent queries on one joint data structure is obviously not as straight forward, as in the PRAM case. The problem lies with the distributed nature of memory on processor networks. Lacking the shared memory of the PRAM, any joint data structure on a processor network, must be divided into small nodes of constant size; each node being stored on a different processor. Having distributed the nodes of the data structure across a processor network, one is left with a very difficult task: How can the now distributed data structure be efficiently accessed by all processors in parallel? If a large number of processors require the same node of the data structure at the same time, how can these requests be handled efficiently; ie. without queuing requests for the required node at the processor that stores it? If the one-structure-n-queries paradigm is to be supported on a processor network, an access scheme must be developed that allows an arbitrary number of processors to access the same node of a joint data structure at the

same time. No simple scheme that statically allocates nodes to processors can hope to accomplish this.

Due to the difficulty of solving the general data structure problem described above, researchers have largely abandoned the idea of using data structures on processor networks. This being the case designers of fine-grained hypercube algorithms have generally focused on low-level data movement operations (as presented in Chapter 2), and have largely discarded the idea of using general data structure abstractions. For each new algorithm, a new highly problem specific data movement scheme has been designed. Working at this low level of abstraction (one which exactly models the hypercube's architecture) algorithm designers have been deprived of the standard data structure abstractions that form the basis of most algorithm design for sequential machines and the PRAM.

In this chapter we will show, however, that for hypercube multiprocessors the parallel execution of independent search queries on one joint data structure can be efficiently implemented for a large class of standard data structures. In doing so we will demonstrate how the one-structure-n-queries paradigm can be implemented for hypercubes in a general way; one which permits the hypercube algorithm designer to work with a large class of data structure abstractions similar to those used in PRAM and sequential algorithm design, without sacrificing efficiency.

The approach which we will present is to dynamically divide the data structure up and distribute it over the entire hypercube. Parts of the data structure will be stored on every processor and, when required, multiple copies will be made of those parts that are in high

demand. Our central result will be an algorithm that allows a data structure of n nodes, distributed over an N processors hypercube, to be efficiently searched, by up to $m \leq N$ queries, independently and in parallel.

Our algorithm will allow an arbitrary number of queries to access the same node at the same time. This is essential in implementing the one-structure- n -queries paradigm and cannot, for example, be obtained by embedding graphs directly into the hypercube topology since such embeddings do not solve the problem of multiple simultaneous query access.

We define a class of graphs called *ordered h -level graphs* which includes most of the standard data structures. We show that for such a graph with n nodes and s sources, m search processes (with maximum path length h) can be efficiently executed, independently and in parallel, in time $O(\min\{s \log N, \log^2 N\} + h \log N)$ on a hypercube multiprocessor of size $N = \max\{n, m\}$. We call this procedure *m -way search*.

The chapter is organized as follows: In Section 4.2, we define ordered h -level graphs and the associated m -way search problem. In Section 4.3, we present a highly efficient hypercube algorithm for m -way search on ordered h -level graphs, and in Section 4.4 we describe some extensions to our algorithm. We conclude in Section 4.5.

4.2 DEFINITIONS

In this section we define ordered h -level graphs and the associated m -way search problem.

4.2.1 Ordered h-level Graphs

Assume we are given a directed graph $G=(V,E)$ with vertex set V and edge set E . An ordered h-partitioning of G is a partitioning of V into an ordered sequence of h disjoint sets $L_1, \dots, L_h$ together with an ordering of the elements in each subset L_i ($1 \leq i \leq h$).

For every ordered h -partitioning of G we define, for every $v \in V$, three numbers $Level(v)$, $Levelindex(v)$ and $Index(v)$ as follows:

- $Level(v) = i$ if and only if $v \in L_i$,
- $Levelindex(v)$ is the rank of v in the sequence of the vertices in $L_{Level(v)}$, and
- $Index(v) = \left(\sum_{1 \leq i \leq Level(v)-1} |L_i| \right) + Levelindex(v)$

A directed graph $G = (V,E)$ is called an ordered h-level graph if it has the following properties (see Figure 4.2 for an illustration):

- (1) There exists a constant $k = O(1)$ such that every node of G has an out-degree of at most k .
- (2) There exists an ordered h -partitioning $L_1, \dots, L_h$ of G such that
 - (a) every source of G is contained in L_1 ,
 - (b) if (v,w) is an edge of G then $Level(w) = Level(v) + 1$, and
 - (c) if $(v,w), (v',w')$ are two edges of G with $Index(v) < Index(v')$, then $Index(w) \leq Index(w')$. (We will refer to this property as edge monotonicity)

Figure 4.2 An ordered h-level graph.

Property 4.1: *Ordered h-level graphs are both acyclic and planar.*

The reverse of a graph $G = (V, E)$ is a graph $G^R = (V, E^R)$ where if $(x, y) \in E^R$ then $(y, x) \in E$. Note that the sources of G become the sinks of G^R and that the sinks of G become the sources of G^R .

Property 4.2. *If G is an ordered h-level graph with constant indegree $k' = O(1)$ then the reverse of G , G^R , is an ordered h-level graph.*

In the introduction we claimed that the class of ordered h-level graphs includes most standard data structures; we will now attempt to substantiate this claim.

Lemma 4.0: *All k-nary trees ($K = O(1)$) are ordered h-level graphs.*

Proof: By their definition k-nary trees satisfy the first property of ordered h-level graphs; they have a maximum outdegree (K). A k-nary tree, partitioned into its h levels with a left to right ordering of nodes within a level (Figure 4.3), also satisfies properties 2a-c as follows:

- The root of the tree is always in the level 1 partition, and is the graph's only source.
- All tree edges go from a node at one level to a node at the next level.
- Lastly, edges between levels can be represented by straight lines that never cross, thereby maintaining edge monotonicity (property 2c) ♦

Figure 4.3 A k-ary tree ($k = 2$) as an ordered h-level graph ($h = 4$).

Trees are the most widely used class of data structures. There are two-three, red-black, d-dimensional, segment, binary, quad and AVL trees, to name just a few. All k-ary trees can be represented by ordered h-level graphs but so can many other more complex structures. To represent a tree, splits are required; edges must be permitted from one node to a collection of others. Ordered h-level graphs, on the other hand, can represent structures with both splits and joins (see Figure 4.4). The combination of these two general properties make ordered h-level graphs eminently suitable for the implementation of a wide range of advanced data structures.

Figure 4.4 Splits and joins

However, despite supporting both splits and joins, ordered h-level graphs are not the most general form of directed graphs available for representing data structures. Figure 4.5 shows where ordered h-level graphs fit in the directed graph hierarchy based on restrictiveness.

Figure 4.5 Inclusion hierarchy of directed graphs.

It may seem on first reflection that the class of ordered h-level graphs includes all planar acyclic graphs, but this is not the case. Figure 4.6 is an example of a planar acyclic graph (K 2:2), that is not an ordered h-level graph. Note how edges a and b violate the edge

monotonicity property of ordered h-level graphs. An examination of this simple graph will show that there is no arrangement of the vertices within each level that will satisfy edge monotonicity.

Figure 4.6 The planar acyclic graph, $K_{2,2}$, is not an ordered h-level graph.

Our reasons for using the class of ordered h-level graphs, rather than a more powerful class will be discussed in detail later in this chapter. It suffices to say, at this point, that the use of ordered h-level graphs will lead to a more efficient solution to the m-way search problem than solutions based on general planar, or even non planar graphs.

4.2.2 The m-way Search Problem for Ordered h-level Graphs

Let $G = (V = L_1 \cup \dots \cup L_h, E)$ be an ordered h-level graph (with maximum out-degree k), and let U be a universe of possible search queries on G .

A search path for a query $q \in U$ is a sequence of h vertices of G , $\text{path}(q) = (v_1, \dots, v_h)$, defined by a successor function $f: (V \cup \{\text{start}\}) \times U \Rightarrow \mathbb{N}$ as follows:

- $\text{Index}(v_1) = f(\text{start}, q)$
- $\text{Index}(v_{i+1}) = f(v_i, q)$, $1 \leq i < h$.

Note that f is a function with the property that $f(\text{start}, q) \in L_1$ and for every vertex $v \in V$, $(v, f(v, q)) \in E$. We also define an associated successor rank function $g: (V \cup \{\text{start}\}) \times U \Rightarrow \{1, \dots, k\}$ as follows:

- $g(\text{start}, q)$ is the rank of $f(\text{start}, q)$ in the set $\{\text{Index}(v) \mid v \in L_1\}$, and
- $g(v_i, q)$ is the rank of $f(v_i, q)$ in the set $\{\text{Index}(w) \mid (v_i, w) \in E\}$, $1 \leq i < k$.

For example, if G is a k -ary search tree ($k=4$), for every query q , $f(\text{start}, q)$ is the root of the tree; for every node v , $g(v, q) \in \{1, 2, 3, 4\}$ indicates which child is to be visited next and $f(v, q)$ is the index of (or pointer to) that child. Figure 4.7 gives an example of such a tree with the search path of a single query q , presented in bold.

Figure 4.7 The search path of a single query, q .

Consider an ordered h -level graph, G , with n nodes stored in a hypercube multiprocessor, such that a node, v , with $\text{Index}(v)=i$ is stored in processor $PE(i)$. A search process for a query q with search path $(v_1, \dots, v_h)$ in G is a process divided into h time steps $t_1 < t_2 < \dots < t_h$ such that at time t_i , $1 \leq i \leq h$, there exists a processor which contains a description of both, the query q and the node v_i . Note, however, that we do not assume

that the search path is given in advance; we assume that it is constructed during the search by successive applications of the functions f and g . Also note that since two or more queries may require a node, v_i , at the same time step, t_j , as many as m copies of a single vertex may be required during one time step, where m is the total number of queries.

Given an ordered h -level graph G with n nodes and a set $Q = \{q_1, \dots, q_m\} \subseteq U$ of m queries, $m=O(n)$, then the *m-way search problem* consists of executing (in parallel) all m search processes induced by the m queries.

4.2.3 Using the m -way Search as a Algorithm Design Methodology.

Having defined ordered h -level graphs and their associated m -way search problem, let us consider how one might use a solution to the m -way search problem to help implement a data structure for use in another application. The m -way search can be used to implement a data structure for a parallel searching application given the following conditions: 1) The data structure can be shown to be an ordered h -level graph. 2) application specific f and g functions can be designed. Provided that these two conditions are met, an m -way search procedure can be implemented that will allow N queries to be simultaneously run through the data structure one level at a time. Essentially the algorithm designer is free to consider each query as operating on its own copy of the entire data structure in much the same way that queries and data structures interact in sequential computing. Note that we can now work with traditional high level data structure abstractions, similar to those used in sequential computing (and on the PRAM), instead of only hypercube routing algorithms.

4.3 AN $O(\min\{s \log N, \log^2 N\} + h \log N)$ TIME ALGORITHM FOR THE M-WAY SEARCH PROBLEM ON ORDERED H-LEVEL GRAPHS

Let $G = (V = L_1, \dots, L_h, E)$ be an ordered h -level graph (with h -partitioning $V = L_1, \dots, L_h$), where $|V| = n$, and such that every node has an out-degree of at most $k = O(1)$ and can be stored using $O(1)$ space and s denotes the number of sources of G , that is, the number of $v \in V$ with $\text{Level}(v) = 1$.

Furthermore, let U be a universe of search queries, each of which can also be stored using $O(1)$ space, and let $f: (V \cup \{\text{start}\}) \times U \Rightarrow \mathbb{N}$ and $g: (V \cup \{\text{start}\}) \times U \Rightarrow \{1, \dots, k\}$ be the successor function and successor rank function, respectively, describing the search path in G associated with every search query.

In this section, we consider the problem of solving, on a hypercube multiprocessor, the m -way search problem for a set $Q = \{q_1, \dots, q_m\} \subseteq U$ of m queries. We present an $O(\min\{s \log N, \log^2 N\} + h \log N)$ time algorithm for solving the m -way search problem on a hypercube of size N , where $N = \max\{n, m\}$. We assume that $f(x, q)$ and $g(x, q)$ can be computed in $O(\log N)$ time for any $x \in V \cup \{\text{start}\}$, $q \in U$.

For the remainder we assume that $n = m = N = 2^d$. If this is not the case our results can be applied without increasing their asymptotic time complexity by using a simulated hypercube of size at most $2^d \leq 2N$. In the Section 4.3.1 we describe the assumed initial configuration of the hypercube and in Sections 4.3.2 - 4.3.4 we present the details of our $O(\min\{s \log N, \log^2 N\} + h \log N)$ time m -way search algorithm.

4.3.1 Initial Configuration

The graph G is assumed to be stored in the hypercube such that each vertex v with $\text{index}(v)=i$ is stored in register $v(i)$ of processor $PE(i)$; register $v(i)$ contains fields $v.\text{data}(i)$, $v.\text{level}(i)$, $v.\text{levelindex}(i)$, and $v.\text{index}(i)$, storing a constant amount of data associated with vertex v , its level, levelindex and index, respectively. The edges of G are stored as adjacency lists; for every vertex v stored in register $v(i)$, the indices of the at most k successors of v (i.e. the indices of the vertices w such that $(v,w) \in E$) are stored in the fields $v.\text{successor1}(i)$, ..., $v.\text{successork}(i)$, respectively; see Figure 4.8.

Figure 4.8 The registers required at each processor, $PE(i)$.

The set $Q = \{q_1, \dots, q_m\}$ of m queries is stored in arbitrary order such that every processor $PE(i)$ stores one query in its register $q(i)$. Figure 4.8 shows the set of registers

necessary at every processor $PE(i)$. In addition to the registers $v(i)$ and $q(i)$ mentioned above, the algorithm assumes that every processor also has a register $v'(i)$ to store a copy of another vertex of G as well as other auxiliary registers which will be described later.

4.3.2 Algorithm Overview

The global structure of the m -way search algorithm for ordered h -level graphs is described in Figure 4.9. The m search processes of queries $q_1, \dots, q_m$ are executed in h phases; each phase moving all queries one step ahead in their search paths.

M-Way-Search:	
(1)	Phase ₁ {Match every query with the 1 st node in its search path.}
(2)	For $x := 2$ to h do
(3)	Phase _{x} {Match every query with the x^{th} node in its search path.}

Figure 4.9 Global structure of the m -way search algorithm

The algorithm permutes the queries (in the register $q(i)$) and copies some nodes into the register $v'(i)$, such that at the end of phase x ($1 \leq x \leq h$):

- all queries are sorted with respect to the index of the x^{th} node in their search path, and
- each processor $PE(i)$ contains a query q in its register $q(i)$, and stores in its register $v'(i)$ a copy of the x^{th} node in the search path of q ; we call this the match of q and the x^{th} node of its search path.

A typical situation at the end of a phase is depicted in Figure 4.10; each vertical column represents the register $q(i)$ and $v'(i)$ of a processor $PE(i)$.

	PE(0)			...										PE(15)		
q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
v'.Index(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
v'.Data(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

Figure 4.10 A typical situation at the end of a phase.

In Sections 4.3.3 and 4.3.4 we will describe the details of Phase 1 and Phase x ($2 \leq x \leq h$), respectively. The first phase is different from the remaining phases in that, when ordering the queries with respect to the index of the first node in their search path, the first phase starts with an arbitrary permutation of the queries. In Phase 2 and each subsequent phase the ordering of the previous phase can be used to improve the time complexity of the algorithm.

4.3.3 Phase 1 of the M-Way Search Algorithm

An outline of Phase 1 is given in Figure 4.11. The algorithm consists of four basic steps (see also Figure 4.12 for an illustration).

Phase₁:	
(1)	Every PE(i): $N(i) := f(\text{Start}, q(i))$
(2)	Number($LS(i)$, $v.\text{Level}(i)=1$) $s := LS(0)$ {Note: $LS(i)=LS(i')$ for all i, i' }
(3)	IF $s \geq \log(N)$ THEN Sort($\{[q(i), q(i)], [N(i), N(i)]\}, N(i)$) ELSE SortBySourceIndex(s)
(4)	MoveVerticesToQueries(1)

Figure 4.11 Outline of Phase 1.

Algorithm Phase₁:

Input: A set of unsorted queries $q(i)$ and s sources.

Output: All queries $q(i)$ are sorted with respect to the index of the source node that is the first node in their search path. In addition, each processor $PE(i)$ which stores a query $q(i)$ also stores in register $v'(i)$ a copy of $q(i)$'s source node.

Method: (See Figure 4.11) First, every processor $PE(i)$ calculates the index of the first node in the search path of its query $q(i)$, and stores this value in an auxiliary register $N(i)$. Then, in Step 2, the number of sources is calculated (here represented by a variable s). In Step 3, the queries are sorted by $N(i)$, the index of the first node in the search path. The sorting is performed in one of two possible ways depending on the number of sources. If $s \geq \log(N)$ then bitonic sorting [Bat] is used; if $s < \log(N)$, a procedure called *SortBySourceIndex* is used which sorts the queries in $O(s \log N)$ time and will be described below. Note that in many application s is a constant (e.g. for search trees) and, thus, the sorting step is performed in time $O(\log N)$.

Finally, in Step 4, the source nodes are copied to the queries for which they are the first node in their search path. Due to the ordering of the queries, this step can be performed in $O(\log(N))$ time using a procedure *MoveVerticesToQueries* which will also be described below ♦

The Initial State on entering Phase 1

v.index(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
v.level(i)	1	1	1	2	2	2	2	2	2	3	3	3	4	4	4	4
q(i)	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16

After Step 1 of Phase 1: $N(i)$ = index of the first node in path($q(i)$).

N(i)	2	3	3	2	1	2	1	2	2	3	3	2	3	2	1	2
------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

After Step 2 of Phase 1: $S(i)$ = number of sources.

S(i)	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

After Step 3 of Phase 1: $q(i)$ and $N(i)$ ordered by $N(i)$.

v.index(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
v.level(i)	1	1	1	2	2	2	2	2	2	3	3	3	4	4	4	4
q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3

After Step 4, the final step of Phase 1:
 $v'(i)$ = copy of the first node in path($q(i)$)

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
V'(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

Figure 4.12 An illustration of Phase 1.

Next we will discuss the details of Procedure SortBySourceIndex; see Figure 4.13. This procedure sorts the queries by the first node in their search path (sources) in $O(s \log N)$ time, where s is the number of sources.

SortBySourceIndex(s):	
(1)	Every PE(i): Shift(i):=0
(2)	FOR r:=1 TO s DO
(3)	Concentrate({[q(i),q'(i)], [N(i),N'(i)]}, N'(i)=r)
(4)	Route({[q'(i),q''(i)], [N'(i),N''(i)]}, i+Shift(i), N'(i)=r)
(5)	Number(H(i), N(i)=r) Every PE(i): Shift(i):=Shift(i)+H(i)
(6)	Every PE(i): q(i):=q''(i), N(i):=N''(i)

Figure 4.13 Detailed description of procedure SortBySourceSelected.

Algorithm SortBySourceIndex:

Input: A set of n unsorted queries, $q(i)$, each with an index, $N(i)$, specifying the index of one of s source nodes.

Output: The n queries $q(i)$ (along with their $N(i)$) sorted with respect to $N(i)$.

Method: (See Figure 4.13.) The algorithm uses a register $\text{Shift}(i)$ at each processor which stores the number of queries that have already been sorted. In Step 1, all registers $\text{Shift}(i)$ are initialized to zero. Then, Steps 3 to 5 are executed for each source of the graph. In each iteration, the queries $q(i)$ that need to be matched with that source [as well as the associated source indices $N(i)$] are copied into registers $q'(i)$ [$N'(i)$] of the same processors, concentrated (Step 3), and then appended at the end of the sequence of queries ordered so far (Step 5); finally the register $\text{Shift}(i)$ is updated (Step 5). These steps produce, in the registers $q''(i)$, the correct permutation of the queries, which is then copied back into the registers $q(i)$ [$N(i)$]; see Step 6. Algorithm SortBySourceIndex is a simple parallel implementation of selection sort. Each of s iteration over the main loop takes $O(\log N)$ time, since it is comprised of a series of $O(\log N)$ operations (Theorem 3.1). Hence, the overall time complexity of procedure SortBySourceIndex(s) is $O(s \log N)$ ♦

Lemma 4.1: *Algorithm SortBySourceIndex correctly sorts registers $q(i)$ and $N(i)$ with respect to $N(i)$ in time $O(s \log N)$.*

Once the queries have been sorted by the index of the first vertex in their search path, the matching process between each query and the first node in its search path can be performed in time $O(\log N)$ using the procedure `MoveVerticesToQueries` described in Figure 4.14.

MoveVerticesToQueries(CurrentLevel): (1) <code>Route({[N(i),N'(i)]}, i-1, i>0)</code> <code>PE(1); N'(N):=N(N) + 1</code> (2) <code>Every PE(i) with N'(i)≠N(i): Dest(i):=i</code> (3) <code>Route({[Dest(i),Dest(i)]} N(i), N'(i)≠N(i))</code> (4) <code>RouteAndCopy({[v(i),v'(i)]}, Dest(i), v.Level(i)=CurrentLevel)</code>
--

Figure 4.14 Detailed description of procedure `MoveVerticesToQueries`.

Algorithm `MoveVerticesToQueries(CurrentLevel):`

Input: A set of queries $q(i)$, sorted by their associated $N(i)$ register which stores the index of the next vertex required by each query.

Output: Each query $q(i)$, is matched with (i.e. on the same processor as) a copy of the vertex $v(i)$ with index $N(i)$.

Method: (See Figure 4.14) The parameter `CurrentLevel` (which is one for all sources) denotes the level of the nodes to which the queries are to be routed (i.e., to be matched with). The idea is to calculate for each block of queries requiring the same vertex the largest address of a query in that block (Steps 1 and 2), and to broadcast this information to the required vertex. Then a `RouteAndCopy` operation can be used to broadcast copies of each vertex to the blocks of queries that they are to be matched with (Step 4). Since Steps 1,3,4 all take $O(\log N)$ time by Theorem 3.1 and Step 2 takes $O(1)$ time, algorithm `MoveVerticesToQueries` requires $O(\log N)$ time overall ♦

Lemma 4.2: *Algorithm `MoveVerticesToQueries(CurrentLevel)` matches the set of queries sorted by the next vertex in their search path, with those vertices in $O(\log N)$ time.*

Note that Step 3 of algorithm Phase₁ takes $O(\min\{s \log N, \log^2 N\})$ time since the time complexity of algorithm SortBySourceIndex is $O(s \log N)$ (Lemma 4.1) and Bitonic sort takes $O(\log^2 N)$ time (Theorem 3.1). Since all other steps in Phase₁ take either $O(1)$ (Step 1) or $O(\log N)$ (Steps 2 and 4 by Theorem 3.1 and Lemma 4.2, respectively), Phase₁ can be completed in $O(\min\{s \log N, \log^2 N\})$ time. Given this analysis and Lemma 4.1 and 4.2 we obtain

Lemma 4.3: *Algorithm Phase₁ matches each query $q(i)$ with the first node in its search path in $O(\min\{s \log N, \log^2 N\})$ time.*

4.3.4 Phase x ($2 \leq x \leq H$) of the M-Way Search Algorithm

As indicated in Section 4.3.3, the purpose of each subsequent phase is to advance, in time $O(\log N)$, all queries by one step in their search path. After Phase $x-1$ has been completed, all queries are sorted with respect to the index of the $(x-1)^{\text{th}}$ node in their search path and each processor PE(i) contains a query q in its register $q(i)$ together with a copy of the $(x-1)^{\text{th}}$ node in the search path of q in its register $v'(i)$. The desired effect of Phase x is to have all queries sorted with respect to the index of the x^{th} node in their search path, and have each processor PE(i) contain in its register $v'(i)$, a copy of the x^{th} node in the search path of the query, $q(i)$. An outline of the algorithm for Phase x is given in Figure 4.15; see also Figure 4.16 for an example trace.

Phase(x), $2 \leq x \leq h$:

- (1) Every PE(i): $N(i) := f(v'(i), q(i))$, $G(i) := g(v'(i), q(i))$
- (2) OrderQueriesByNextVertex
- (3) MoveVerticesToQueries(x)

Figure 4.15 Overview of Phase x, $2 \leq x \leq h$.

Algorithm Phase(x):

Input: A set of queries, $q(i)$, sorted with respect to the index of the $(x-1)^{\text{th}}$ node in their search path.

Output: A set of queries, $q(i)$, sorted with respect to the index of the x^{th} node in their search paths; each processor PE(i) storing a $q(i)$, contains a register $v'(i)$ storing a copy of the x^{th} node of $q(i)$'s search path.

Method: (See Figure 4.15) First (in Step 1), every PE(i) computes for the query currently stored in its register $q(i)$ the index of the next node in its search path as well as the successor rank of that node (see Section 4.2.2). These values are stored in the auxiliary registers $N(i)$ and $G(i)$, respectively. In Step 2, all queries are sorted by the index of the next node in their search paths. This sorting operation is performed by a procedure OrderQueriesByNextVertex in time $O(\log N)$ (Lemma 4.4) by using the properties of the previous permutation of the queries. Once this ordering has been obtained, the nodes can be matched with the queries in time $O(\log N)$ (Lemma 4.2) in the same way as described in Section 4.3.3 ♦

Initial input to Phase 2.

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
V'(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

After Step 1 of Phase 2: N(i) = index of the next node in
path(q(i)); G(i) = its successor rank.

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	5	4	5	6	8	7	7	9	6	6	5	9	9	9	9	9
G(i)	2	1	2	2	4	3	3	5	2	2	1	1	1	1	1	1

After Step 2 of Phase 2:
q(i) and N(i) have been ordered by N(i).

q(i)	q7	q5	q15	q2	q1	q14	q16	q8	q4	q9	q16	q2	q3	q10	q11	q13
N(i)	4	5	5	5	6	6	6	7	8	9	9	9	9	9	9	9

After Step 3 of Phase 2:
v'(i) = copy of the next node in path(q(i)).

q(i)	q7	q5	q15	q2	q1	q14	q16	q8	q4	q9	q16	q2	q3	q10	q11	q13
N(i)	4	5	5	5	6	6	6	7	8	9	9	9	9	9	9	9
v'(i)	v4	v5	v5	v5	v6	v6	v6	v7	v8	v9	v9	v9	v9	v9	v9	v9

Figure 4.16 An illustration of Phase 2.

What remains to be discussed are the details of procedure OrderQueriesByNextVertex. This procedure, which is described in Figure 4.17, creates in time $O(\log N)$ a new ordering of the queries with respect to the indices of the next vertices in the query's search paths.

Figure 4.17 Details of procedure OrderQueriesByNextVertex.

Algorithm OrderQueriesByNextVertex:

Input: A set of n unsorted queries $q(i)$, each with an associated $N(i)$ specifying the index of the next vertex in each query's search path.

Output: The n queries $q(i)$ along with their $N(i)$ sorted with respect to $N(i)$.

Method: (See Figure 4.17) Consider all edges (v,w) and (v',w') where $\text{Level}(v)=\text{Level}(v')=x-1$ and w and w' have the same successor rank. The edge monotonicity property ensures that if $\text{Index}(v) < \text{Index}(v')$ then $\text{Index}(w) \leq \text{Index}(w')$. Therefore, the sub-sequence of queries, for which the successor rank of the next vertex in their search path has the same value r , is already sorted with respect to the index of the next vertex. Furthermore, notice that since each node has an outdegree of at most k , there are at most $k \cdot O(1)$ such sub-sequences. The idea for creating, in time $O(k \log N)=O(\log N)$, the new ordering of the queries is therefore to extract these k ordered sub-sequences and merge them in k bitonic merge steps. The details are shown in Figure 4.17: for each of the k possible successor ranks, the respective sub-sequence of queries is extracted (Step 3), its length calculated (Step 4), inverted (Step 5), and appended to the sequence of queries already ordered (Steps 6). Finally, in Step 7, the bitonic sequence created in Steps 3-6 can be converted into a sorted sequence, using the bitonic merge operation.

The main loop of algorithm *OrderQueriesByNextVertex* loops a constant number of times, k , over a set of $O(1)$ and $O(\log N)$ operations; therefore the time complexity of the entire operation is $O(\log N)$ ♦

Lemma 4.4. *Algorithm *OrderQueriesByNextVertex* creates in time $O(\log N)$ a sorted ordering of the queries, $q(i)$, with respect to the indices of the next nodes in the search paths, $N(i)$.*

Note that algorithms *OrderQueriesByNextVertex* and *MoveVerticesToQueries* require $O(\log N)$ time (Lemma 4.4 and 4.2); hence the time complexity of algorithm *Phase(x)* is $O(\log N)$. Given this analysis and Lemma 4.2 through 4.4, we obtain the following:

Lemma 4.5. *Algorithm *Phase(x)* advances all queries $q(i)$ one step in their search paths in time $O(\log N)$.*

Summarizing, we obtain the following property and theorem:

Property 4.4: *The m -way search consists of h phases such that at the end of phase x ($1 \leq x \leq h$) all queries are sorted with respect to the index of the x^{th} node in their search path, and each processor $PE(i)$ contains in its register $v'(i)$ a copy of the x^{th} node in the search path of the query currently stored in its register, $q(i)$.*

Theorem 4.1. *The m -way search problem for an ordered h -level graph with n nodes and s sources can be solved on a hypercube multiprocessor of size $N = \max\{n, m\}$, in time $O(\min\{s \log N, \log^2 N\} + h \log N)$.*

4.4 EXTENSIONS

The m -way search algorithm for ordered h -level graphs, presented in Sections 4.3.1 to 4.3.3, has been described in terms of a fixed number of queries, $m=N$. In this Section we will show that the number of queries m need not be a constant throughout the m -way search. We will show that, without changing the time complexity of the algorithm, the number of queries existing at any one phase of the algorithm may range from $m=O(1)$ to $m=O(N)$.

Recall from Property 4.4 that at the end of phase x ($1 \leq x \leq h$) all queries are sorted with respect to the index of the x^{th} node in their search path and each processor $PE(i)$ contains in its register $v'(i)$ a copy of the x^{th} node in the search path of the query currently stored in its register $q(i)$. A DeleteQueries operation can therefore be performed at the end of any phase x , by concentrating the queries that are not to be deleted along with their associated $v'(i)$ values. This deletion and compaction step being performed in $O(\log N)$ time using the Concentrate operation described in Chapter 3.

Extension 4.1: *At any phase x in a m -way search, up to m queries may be deleted without increasing the asymptotic time complexity of the m -way search algorithm.*

Similarly, at the end of any phase x , an AddQueries operation may be performed, in which any query $q(i)$ currently visiting a node $v'(i)$ can create $j=O(1)$ new search queries, each starting their search path at node $v'(i)$. The only restriction being that the total number of queries existing at any one time, m , should not exceed $O(N)$. In an AddQueries operation, each query stores in an auxiliary register, one plus the number of new queries it

wishes to create. A Psum operation is then performed on this register to calculate the new address of each query. Finally, each query $q(i)$ and its associated $v'(i)$ are copied to this new address, using a RouteAndCopy operation. Hence, each new query starts as a copy of the query that created it. Any additional information needed to initialize a new query can be distributed again via a RouteAndCopy operation. The AddQueries operation involves only a constant number of $O(\log N)$ operations, consequently it does not effect the asymptotic time complexity of the m -way search algorithm. Summarizing we obtain

Extension 4.6: *At any phase x in a m -way search, the number of queries visiting any node v can be increased by a constant number without increasing to the time complexity of the m -way search algorithm.*

4.5 CONCLUSION

In this chapter we have defined and solved the m -way search problem for ordered h -level graphs. In doing so we have suggested that the m -way search is a valuable new tool for defining and using data structures on hypercube multiprocessors. In Chapter 5 we will add further weight to our claim, by demonstrating how the m -way search technique can be used to generate elegant algorithms (based on advanced data structures) for solving complex problems, while establishing new upper bounds on their complexity (for the hypercube multiprocessor model).

5 APPLICATIONS OF THE M-WAY SEARCH

In this chapter we will present several applications of the m-way search technique to fundamental problems in computational geometry. These problems include the construction of the trapezoidal map for a set of S disjoint line segments and the triangulation problem for simple polygons.

5.1 PARALLEL BINARY SEARCH

Using the m-way search procedure defined in Chapter 4, many standard sequential data structures can be parallelized. In this section we will implement a binary search tree, which can be used for implementing multiple parallel binary search processes on a hypercube multiprocessor.

A binary search tree for n items is a standard (leaf orientated) balanced binary tree of size $2n-1$. Each internal node has a left child pointer, $lchild$, a right child pointer, $rchild$, and a unique key, such that $key(lchild(n)) < key(n) \leq key(rchild(n))$. Each leaf of a binary search tree consists of a key, $key(i)$, and a set of data fields. A binary search tree is an ordered h -level graph whose index corresponds to the level ordering of the nodes. A binary search tree for n items is stored on a simulated hypercube of size $2n-1$ where each node v with $index(v) = i$ is stored on virtual processor $PE(i)$; see Figure 5.1 for an example.

Figure 5.1 A binary search tree for 8 elements with level indices

Given a binary search tree, multiple binary search consists of answering $m = O(n)$ search queries by matching each query $q(i)$ with a leaf node $l(i)$ of the tree, such that $l(i)$ is the leaf node with the smallest key that is greater than (or equal to) the key of $q(i)$. We will now present an algorithm for constructing a binary search tree for n items on a hypercube multiprocessor of size $N = O(n)$. We will demonstrate that in addition to searching in data structures, the m -way search procedure can sometimes also be used to build them on hypercube multiprocessors.

Algorithm Construct Binary Search Tree:

Input: A set of n unsorted records $r(i)$ each containing a field $r.key(i)$.

Output: An ordered h -level graph representing a binary search tree for the records $r(i)$.

Method: Our method consists of three steps. First, the records $r(i)$ are sorted by their key field, $r.key(i)$. Then a binary tree is constructed with the records $r(i)$ as its leaf nodes. Finally, the keys (of the internal nodes) of the binary search tree are computed by a m -way search procedure as follows. A set of n queries is constructed at the leaf nodes; each query

consisting of the leaf node's key value. These queries are moved, in $O(\log n)$ phases, towards the root node. At the end of each phase the largest query of all queries at a node v is the key assigned to that node, and the smallest query is moved to the next level, all others being deleted. See Figure 5.2 for an example of the operation of this m-way search procedure.

Figure 5.2 Constructing a binary search tree using the m-way search

The time to perform the m-way search operation determines the time complexity of the entire algorithm. Since the m-way search procedure consists of $O(\log n)$ steps, each taking $O(\log n)$ time on hypercube of size $N=O(n)$, this algorithm requires $O(\log^2 n)$ time ♦

Lemma 5.1: *Algorithm Construct Binary Search Tree creates, in time $O(\log^2 n)$, a binary search tree for n elements (as an ordered h -level graph) on a hypercube multiprocessor of size $N=O(n)$.*

Having constructed a binary search tree, the multi binary search problem for $m=O(n)$ queries can be solved as follows:

Algorithm Multi Binary Search:

Input: A binary search tree of n items and a set of $m=O(n)$ queries, $q(i)$.

Output: Each query, $q(i)$, resides on a processor $PE(i)$, along with a copy of the leaf node $l(i)$ of the binary search tree, such that $l(i)$ is the leaf node with the smallest key, that is greater than (or equal to) the key of $q(i)$.

Method: Our algorithm consists of a single m -way search procedure in which the m queries, $q(i)$, start phase 1 of the m -way search at the root of the binary search tree and are moved down the tree in $O(\log n)$ phases. In each phase, an individual query $q(i)$ is routed to the right child of a node, $n(i)$, if $n.key(i) \leq q.key(i)$, otherwise the query is routed to the left child. Since the m -way search procedure consists of $O(\log n)$ steps, each taking $O(\log m)$ time, $m=O(n)$, this algorithm can be completed in $O(\log^2 N)$ time on hypercube of size $N=O(n)$ ♦

Theorem 5.1: *Algorithm Multi Binary Search answers $m=O(n)$ queries on a binary search tree of n items in time $O(\log^2 N)$ using a hypercube of size $N = \max\{m, 2n-1\}$.*

In some applications, the multi binary search algorithm described above is no more effective than simpler methods. Consider, for example, the problem of answering m queries on a set of n records, where more than one query, $q(i)$, may request any one record $r(i)$. If there exists a complete ordering of the set of all possible records and queries combined, a simple sort procedure will solve this problem as follows. Sort the combined set of m queries and n records in increasing order and for each query $q(i)$ report the first record $r(i)$ to its right in the sorted sequence. This simple procedure also takes $O(\log^2 N)$ time, $N=\max\{m,n\}$, as would a procedure based on the binary search method outlined above. Our parallel binary search method has, however, an important advantage over the simple sort based binary search method. Our method can be applied to problems which do

not have a complete ($O(1)$ time) ordering of the combined set of queries and records. Note that the multiple binary search only requires a partial order, where any two records as well as any query-record pair are comparable. For such cases, the simple sort based method can not be applied, but our method still provides a $O(\log^2 N)$ time solution.

It may appear, at first, that such situations rarely occur; this is not the case. For example, consider the problem of next element search in a slab. Given a set S of n non intersecting line segments $L_1, \dots, L_n$, that completely cross a vertical slab, and a set of m query points $p_1, \dots, p_m$ contained in the slab, the next element search problem in a slab consists of finding for each point p_i the line segment L_j first intersected by the vertical ray starting at p_i in an upwards direction (as illustrated in Figure 5.3.)

Figure 5.3 Next element search in a slab

This problem cannot be solved with the simple sort based binary search since there is no suitable complete ordering of the combined set of points and lines. The problem is in comparing points to points. Consider points p_3 and p_4 in Figure 5.3. Without reference to line segment L_3 , it is not possible to compare p_3 and p_4 , since the relative position of the

points alone does not contain sufficient information. For example, although P_3 is above p_4 , p_3 must be considered smaller than p_4 , due to the position of the line L_3 .

Despite the lack of a complete ($O(1)$ time) ordering of points and lines, the next element search problem in a slab can be solved efficiently on a hypercube using multi binary search as follows:

Algorithm Next Element Search in a Slab:

Input: A set S of n non intersecting line segments $L_1, \dots, L_n$ that completely cross a vertical slab, and a set of $m=O(n)$ query points $p_1, \dots, p_m$ which lie within the slab.

Output: For each point p_i the line segment L_j first intersected by the vertical ray starting at p_i is reported.

Method: Our algorithm consists of two steps. First, we construct a binary search tree $T(L)$ out of the line segments $L_1, \dots, L_n$. Then we perform a multi binary search on $T(L)$ with query set $p_1, \dots, p_m$. At the end of the multi binary search, each query $q(i)$ resides on a processor $PE(i)$ which also stores a line L_i . This line is the first line intersected by the ray starting at p_i in direction D_{next} , and can therefore be reported. The time complexity of this algorithm is $O(\log^2 n)$ since both the tree construction and tree search steps are of this time complexity ♦

Theorem 5.2: *The Next Element Search Problem in a Slab can be solved for n lines and $m=O(n)$ points, in time $O(\log^2 N)$, on a hypercube of size $N=\max\{n,m\}$.*

5.2 THE GENERAL NEXT ELEMENT SEARCH PROBLEM

In this section, we will present a highly efficient parallel implementation, for the hypercube multiprocessor, of a well known data structure: the segment tree [BW]. The segment tree is a widely used structure which has, for example, been used to obtain

efficient implementations of plane sweep algorithms in computational geometry [BW], [Meh], [PS] [AG]. Here, we consider an application of the segment tree to the next element search problem.

We will first review the definition of the general next element search problem, as well as the definition and some basic properties of segment trees. We will then show how to implement a segment tree on a hypercube multiprocessor, using the parallel m-way search algorithm for ordered h-level graphs, and then obtain a highly efficient solution for the (general) next element search problem.

The (general) *next element search* problem is a well known problem in computational geometry. Given a set S of n non intersecting line segments $I_1, \dots, I_n$ and a direction D_{next} (without loss of generality we will assume that D_{next} is the direction of the positive Y-axis), the next element search problem consists of finding for each point p_i of a set of $m=O(n)$ query points $p_1, \dots, p_m$ the line segment I_j first intersected by the ray starting at p_i in direction D_{next} , as illustrated in Figure 5.4.

Figure 5.4 The next element search problem.

A well known method for solving the next element search problem is to apply a plane sweep in direction D_{next} using a segment tree [BW], [Meh] [PS].

Let $I_1^{(x)}$ [$p_1^{(x)}$] be the projection of line segment I_1 [point p_1 , respectively] onto the x -axis, and let $(x_1, x_2, \dots, x_{2n})$ be the sorted sequence of the projections of the $2n$ endpoints of $I_1, \dots, I_n$ onto the x -axis. The range tree $T'(S) = (V_s, E_s)$ for S is the complete binary tree with leaves $x_1, \dots, x_{2n}$. For every node v of $T'(S)$, an interval $xrange(v)$ is defined as follows [PS]:

- if v is a leaf x_i , then $\text{xrange}(v) = [x_i, x_{i+1})$. ($[x_{2n}, x_{2n+1}) = [x_{2n}, x_{2n})$)
- if v is an internal node, then $\text{xrange}(v)$ is the union of all intervals $\text{xrange}(v')$ such that v' is a leaf of the subtree of $T(S)$ rooted at v .

The segment tree $T(S)$ is the range tree $T'(S)$ in which for every node v there is associated a node list $NL(v) \subseteq S$ which is defined as follows:

$$NL(v) = \{ I \in S \mid \text{xrange}(v) \subseteq I^{(x)} \text{ and not } (\text{xrange}(\text{father of } v) \subseteq I^{(x)}) \}.$$

See Figure 5.5 as an example.

Figure 5.5 A segment tree with node lists.

Note that a segment tree $T(S)$ is an ordered h -level graph, where h is the height of $T(S)$. Let $\text{Level}(v)$, for any node v , be the height of v in $T(S)$, $\text{Levelindex}(v)$ be the rank of v in $\{v' | \text{Level}(v') = \text{Level}(v)\}$ with respect to the ordering of these nodes by increasing x -coordinate of $x_{\text{range}}(v')$, and $\text{Index}(v)$ be defined as described in Section 4.2.1 of Chapter 4.

For every query point p , we define $\text{path}(p)$ to be the path in $T(S)$ from the root to the leaf v such that $p^{(v)} \in x_{\text{range}}(v)$. In order to solve the next element search problem, a copy of each query point p is distributed to each node along $\text{path}(p)$. At every node v on the path, the next element of p in $NL(v)$ is determined to form a set of partial answers for p ; this process will be referred to as *locating* p in $NL(v)$. For each query point, the final result to be reported is the closest next element found by any of the copies of p . What remains to be described is how to build the segment tree $T(S)$ and how to locate the query points in the node lists on their paths.

Note that each line segment can occur in $O(\log n)$ node lists and, thus, the sum of the lengths of all node lists is $O(n \log n)$ [Meh p. 214]. Hence, storing the segment tree on a hypercube multiprocessor requires $O(n \log n)$ processors.

5.2.1 Building a Segment Tree on a Hypercube of Size $O(n \log n)$

A segment tree $T(S)$ for n non intersecting line segments can be constructed on a hypercube multiprocessor of size $O(n \log n)$ by first constructing a range tree for the n line segments and then converting the range tree into a segment tree (by adding the node lists $NL(v)$).

Algorithm Construct Range Tree:

Input: A set S of n non intersecting line segments $L_1, \dots, L_n$

Output: An ordered h-level graph representing the range tree for the line segments $L_1, \dots, L_n$.

Method: Our method consists of 3 steps. First, the $2n$ endpoints of the line segments $L_1, \dots, L_n$ are sorted. Then a binary tree whose leaves are the $2n$ endpoints is constructed. Finally, the $xrange(i)$ values for the internal nodes of the range tree are calculated by an m-way search procedure as follows. At each leaf node, a query consisting of the projection $p_i^{(x)}$ of the respective endpoint p_i is generated. These queries are moved, in $O(\log n)$ phases towards the root node, where at each phase the lowest and highest query values at a node v are used to define $xrange(v)$ and are then passed up to the next level; all other queries being deleted. See Figure 5.6 for an example of the operation of this m-way search procedure ♦

Figure 5.6 A range tree for a set of eight endpoints

Lemma 5.2: A Range Tree for a set of n line segments can be constructed on a hypercube of size $O(n)$ in time $O(\log^2 n)$.

A range tree for a set of line segments $L_1, \dots, L_n$, can be converted into a segment tree for the same set of line segments by adding the node lists $NL(v)$ as follows:

Algorithm Construct Segment Tree:

Input: A Range Tree $T'(S)$ for a set S of n non intersecting line segments $L_1, \dots, L_n$

Output: An ordered h -level graph of size $O(n \log n)$ representing the segment tree for the line segments $L_1, \dots, L_n$. Each node in the segment tree has a pointer to its catalog in a compacted list of all catalogs. These catalogs being internally ordered by their associated node's index.

Method: Our method consists of a single m -way search procedure. Out of the set of n line segments $L_1, \dots, L_n$, a set of n queries is formed. This set of queries begins, in phase 1 of the m -way search, at the root of the range tree $T'(S)$ and is moved down the tree in the subsequent phases as follows. At each subsequent phase in the m -way search, each query at a node v , adds itself to $NL(v)$ if it is totally covered by $xrange(v)$ and is then deleted. The remaining queries are then compacted using a concentrate operation (see Chapter 3). A remaining query, q , at node v in level $i-1$ will move to each child v' of v if and only if the q intersects $xrange(v')$. This means that zero, one, or two copies of a query q at level $i-1$ may appear at level i . This procedure obviously constructs a correct set of node lists, as it exactly corresponds to the definition of such lists. What remains to be shown is that it does so without creating more than $O(n \log n)$ queries at any one step in the m -way search procedure. This can be shown by noting that any query generated at level i must create a node list entry at some level $j, j > i$. Since the total size of all node lists is at most $O(n \log n)$ [Meh p. 214], the total number of queries existing at any step in the m -way search can be no more than $O(n \log n)$. See Figure 5.5 for an example of a fully constructed segment tree ♦

Theorem 5.2: *A Segment Tree for a set of n line segments can be constructed on a hypercube of size $O(n \log n)$ in time $O(\log^2 n)$.*

5.2.2 Implementing Next Element Search Using a Segment Tree

Having described the implementation of a segment tree, we now show how the general next element search problem can be solved.

Algorithm Next Element Search:

Input: A Segment Tree $T(S)$ for a set S of n non intersecting line segments $I_1, \dots, I_n$ and a set of query points $p_1, \dots, p_m$ ($m = O(n)$).

Output: For each point p_i the line segment I_j first intersected by the ray starting at p_i in direction D_{next} is reported; see Figure 5.4.

Method: First, every query point p is duplicated down the path $\text{path}(p)$ from the root of $T(S)$ to the leaf v such that $p^{(x)} \in \text{xrange}(v)$, using a single application of the m-way search.⁶ All points copied to a node v are stored in a point list $PL(v)$ that may span several processors. Recall from Section 5.1 that the next element search problem in a slab can be solved on a hypercube in time $O(\log^2 n)$. For each node v , in $T(S)$, we consider $NL(v)$ as our set of line segments and $PL(v)$ as our set of points. Simultaneously, for every node v , we then perform the next element search algorithm in a slab with respect to $NL(v)$ and $PL(v)$. Note that all node lists must be converted into binary trees before the standard next element search algorithm can be applied; see Figure 5.7. This conversion step can be completed by having each node list element calculate its address in a binary tree and then using a sort operation on these addresses to construct the binary trees in time $O(\log^2 n)$. Having converted all $NL(v)$ into binary trees, the simultaneous performance of multi binary

⁶ Note that this will require $O(n \log n)$ processors, the same number required for the segment tree.

search on these trees is made possible by the fact that a forest of binary search trees is also an ordered h-level graph.

Figure 5.7 Each node list is converted into a small binary tree.

Finally, the $O(\log n)$ partial results generated for each point by the next element search in a slab procedure can be combined into a single final answer by sorting the partial answers by first key, point, and second key, distance. The final answer for any point p will be the first answer in the block of partial answers for p ♦

Summarizing, we obtain

Theorem 5.3. *The next element search problem for a set of n disjoint line segments and m query points can be solved on a hypercube of size $N \log N$ in time $O(\log^2 N)$; $N = \max\{m, n\}$.*

5.3 THE TRAPEZOIDAL MAP PROBLEM

Theorem 5.3 also implies a highly efficient hypercube solution for another fundamental geometric problem: the construction of the trapezoidal map [TW] (see Figure 5.8).

Figure 5.8 The Trapezoidal Map of a set of line segments

Given a set S of n disjoint line segments in the plane; for any endpoint p of a segment in S , the trapezoidal segments for p are the (at most two) line segments first intersected by the rays emanating from p in direction of the positive and negative y -axis, respectively. The construction of the trapezoidal map consists of finding, for each endpoint of the segments in S , its trapezoidal segments.

This problem is fundamental in computational geometry and is frequently used to solve other geometric problems; see e.g. [Goo], [TW], and [Yap]. Atallah, Cole, and Goodrich [ACG], [Goo] presented an $O(\log n)$ time algorithm for computing the trapezoidal decomposition on a PRAM with $O(n)$ processors and $O(n \log n)$ space. As a consequence

of Theorem 5.3, we obtain by two next element searches (one in the positive and the other in the negative y direction) with the endpoints of the segments serving as query points the following:

Corollary 5.1. *For a set of n disjoint line segments, the trapezoidal map can be computed in time $O(\log^2 n)$ on a hypercube of size $N = (n \log n)$.*

5.4 TRIANGULATION OF A SIMPLE POLYGON

Yap [Yap] has shown that on a PRAM with $O(n)$ processors and $O(n \log n)$ space, the triangulation of a simple polygon (see [TW] and Figure 5.9) can be computed in time $O(\log n)$ by applying two calls of the trapezoidal map algorithm (of [ACG], [Goo]).

Figure 5.9 A simple polygon before and after triangulation

By combining the result in [Yap] with Corollary 5.2, we obtain

Corollary 5.2. *An n -vertex simple polygon can be triangulated on a hypercube multiprocessor of size $n \log n$ in time $O(\log^2 n)$.*

5.6 CONCLUSION

In this chapter we have solved for the hypercube multiprocessor some fundamental problems in computational geometry using the m-way search algorithm. We have demonstrated that the m-way search algorithm can be used to construct advanced data structures that can, in turn, generate both elegant and efficient parallel hypercube solutions to these problems.

6 PARALLEL FRACTIONAL CASCADING ON HYPERCUBE MULTIPROCESSORS

In this chapter, we will enhance our technique for implementing data structures on fine-grained hypercubes in several important ways. In doing so we will demonstrate a parallel version of fractional cascading [CG1]. Fractional cascading is a well known technique for building data structures particularly suitable for computational geometry problems [CG2].

In implementing a parallel version of fractional cascading we will generalize and use many ideas developed in the m -way search algorithm. Recall that in the m -way search algorithm, ordered h -level graphs were used to represent data structures. In this chapter, we will introduce a new class of graphs capable of representing many more data structures than was previously possible. We will also lift several previously imposed restrictions on queries and their paths. In doing so we will allow queries to move both forwards and backwards in our data structures. We will also relax the constant space restriction we previously imposed on the size of individual nodes in our data structures. Instead of insisting that each node of a data structure has only constant space, we will allow individual nodes to have arbitrary space, so long as the total space allocated is $O(N)$.

6.1 PROBLEM OVERVIEW

Fractional cascading, as described by Chazelle and Guibas [CG1], is a general data structuring technique in which catalog graphs are used to represent data structures. A catalog graph is a directed acyclic graph where each node in the graph has associated with it

a catalog of ordered items from some ordered domain U . Figure 6.1 is an example of a data structure represented as a catalog graph.

Figure 6.1 An example catalog graph

An iterative search query on a catalog graph differs from the search queries described in Chapter 4 in that at every node visited the decision as to which node to visit next depends partly on a query look-up in the current node's catalog. An obvious sequential algorithm to perform this type of search is to route a query through the catalog graph (in p phases) performing binary search at each step along the query's path. This naive approach has a $O(p * \log N)$ time complexity. Chazelle and Guibas' principle result in [CG1] does, however, show that this can be improved to time $t_s = O(p + \log N)$ in the sequential setting

In the parallel setting, the analogous problem is considered in terms of a catalog graph (of size n) and a set of $m = O(n)$ search queries, each with a path length of, at most, p . This problem, referred to as the multi iterative search problem, can be easily solved in the PRAM by having each processor execute the sequential algorithm. This approach has been extensively used by Atallah, Cole, and Goodrich [ACG] [Goo].

In this chapter, we will study the multi iterative search problem for the hypercube multiprocessor. For *monotone* catalog graphs (to be defined in Section 6.3) of bounded (fixed) degree, we will show that m multiple iterative search queries (including catalog lookups) can be executed independently, in parallel, in time $O(t_s \log N)$ on a hypercube multiprocessor of size $N = \max\{n, m\}$. Again, note that our solution allows an arbitrary number of search queries to access the same node and its catalog at the same time. Our implementation of parallel fractional cascading is another example of a highly efficient hypercube implementation of the one-structure- n -queries paradigm described in Chapter 1.

The remainder of this chapter is organized as follows. In Section 6.2 we define fractional cascading and in Section 6.3 we present a general solution for parallel fractional cascading on a hypercube multiprocessor.

Compared to the sequential methods presented in [CG1], we need to impose restrictions on the type of catalog graphs (monotonicity) as well as the type of search queries. We will, however, demonstrate in Chapter 7 that despite these restrictions our method does provide parallel implementations (on hypercube multiprocessors) of many data structures. In Chapter 7, we will present an algorithm based on a convex hull decomposition tree and will show how this data structure can be built using parallel fractional cascading for other examples see [DFR1].

6.2 DEFINITIONS

Consider a directed planar graph $G=(V,E)$ with bounded (fixed) degree; that is, there exist constants μ_{out} and μ_{in} such that, for every vertex $v \in V$, the out-degree and in-degree of v are at most μ_{out} and μ_{in} , respectively. Assume that G is connected and has n_v vertices

(and therefore $O(n)$ edges). As in [CG1], we associate with each vertex v of G a catalog C_v consisting of an ordered collection of records from a totally ordered domain U . The graph G together with its catalogs is referred to as a catalog graph of size $n = \max\{n_v, n_c\}$, where n_c is the sum of the sizes of all catalogs.

A path π in G (of length p) is a sequence of vertices $v_0, v_1, \dots, v_{p-1}$ such that for each $0 \leq i < p$, either $(v_i, v_{i+1}) \in E$ [forward edge] or $(v_{i+1}, v_i) \in E$ [backward edge]. A multiple look-up query is a pair (q, π) where q is a value of U and π is a path in G . For each catalog C we denote by $\sigma(q, C)$ the successor of q in C , that is the first record of C whose value is greater than or equal to q . The iterative search problem consists of executing a multiple look-up query (q, π) by following the path π in G and determining for every vertex v on the path the successor of q in C_v . The path π is assumed to be specified on-line, that is the successor v_{i+1} of the vertex v_i in π is only known after the query has reached v_i and determined $\sigma(q, C_{v_i})$.

The above Definition of fractional cascading follows the one given in [CG1], but assumes fixed degree catalog graphs (instead of catalog graphs with locally bounded degree, see [CG1]) and uses a restricted definition of the considered type of paths. For the above case it follows from [CG1] that in $O(N)$ time and space it is possible to construct, for the standard sequential machine model, a data structure that solves the iterative search problem for a multiple look-up query with path length p in time $O(p + \log s)$.

We now study how to obtain a parallel implementation of fractional cascading on a hypercube multiprocessor of size N . Compared to the sequential methods presented in [CG1], we need to impose a second restriction on the type of catalog graphs. Consider a catalog graph $G=(V,E)$ of size N with n vertices. G is called a *monotone catalog graph* if there exists a one-to-one index function $Index: V \Rightarrow \{1, \dots, n\}$ with the following property:

2

OF/DE

2

MicroD

if (v, v') or (w, w') are two edges of G with $\text{Index}(v) < \text{Index}(w)$ then $\text{Index}(v') \leq \text{Index}(w')$.

Figure 6.2 shows an example of a monotone catalog graph of size $n = 16$.

Figure 6.2 An example of a monotonic catalog graph

Note that the class of monotone graphs includes all ordered h -level graphs and therefore all k -nary trees ($k=O(1)$). However, not every planar graph is monotone, consider the graph given in Figure 4.5 for instance. The class of monotone graphs is less restrictive than the class ordered h -level graphs described in Chapter 4 and yet more restrictive than the class of planar graphs. Figure 6.3 demonstrates the relationship between these classes of graphs.

Figure 6.3 Monotone graphs in relation to several other classes

Of course, monotone catalog graphs are not as general as the class of graphs used by Chazelle and Guibas in [CG1] and [CG2]. However, our experience, outlined in Chapter 7 and [DFR1], suggests that most data structures can be modeled by monotone graphs. The only non-monotone data structure we have found to date is Chazelle's hive graph [Cha] ; see Figure 6.4 for an example.

Figure 6.4 A Hive Graph (An example of a non monotonic data structure)

6.3 A HYPERCUBE IMPLEMENTATION OF FRACTIONAL CASCADING

In the remainder of this chapter we will show how, given a *monotone* catalog graph of size n and a set $Q = \{q_1, \dots, q_m\}$ of m multiple iterative search queries along paths of

length at most p , these m multiple look-up queries can be executed independently, in parallel, in time $O((p + \log N) \log N)$ on a hypercube multiprocessor of size $N = \max\{n, m\}$ ⁷. We first describe the assumed initial configuration of the hypercube, and then present details of the different phases of the algorithm.

6.3.1 Initial Configuration

The graph G is assumed to be stored in the hypercube such that each vertex v with $\text{Index}(v) = i$ is stored in register $v(i)$ of processor $\text{PE}(i)$. For every vertex v , let $\text{pred}(v)$ and $\text{succ}(v)$ be the sets of possible predecessors and successors in G ; i.e., the sets of at most μ_{in} and μ_{out} vertices w such that $(w, v) \in E$ and $(v, w) \in E$, respectively. We assume that register $v(i)$ contains fields $v.\text{index}(i)$, $v.\text{successor}_1(i), \dots, v.\text{successor}_{\mu_{\text{out}}}(i)$, and $v.\text{predecessor}_1(i), \dots, v.\text{predecessor}_{\mu_{\text{in}}}(i)$, storing the $\text{Index}(v)$, the indices of the vertices in $\text{succ}(v)$, and the indices of the vertices in $\text{pred}(v)$, respectively. Every register $v(i)$ also has a field $v.\text{EndCat}(i)$ storing the address (processor number) of the last record of the associated catalog.

We assume that each $\text{PE}(i)$ also has a register $c(i)$ to store a catalog record. Each record $c(i)$ contains a field $c.\text{index}(i)$ storing the index of the associated vertex of G and a field $c.\text{key}(i)$. The catalogs are stored in sorted order with respect to the index of the associated vertices, and each catalog is internally sorted with respect to the order on U (using $c.\text{key}(i)$).

For a given set $Q = \{(q_1, \pi_1), \dots, (q_N, \pi_m)\}$ of m multiple look-up queries, every processor $\text{PE}(i)$ stores in its register $q(i)$ one arbitrary query value q_j . The search paths π_j ($1 \leq j \leq N$) are determined *on-line* by the following two functions:

⁷ We assume, w.l.o.g., that $N = 2^d$; otherwise we at most double the data set to the next power of 2.

- start: $U \Rightarrow \{1, \dots, n_v\}$
- $g: V \times U \times U \Rightarrow \{-\mu_{in}, \dots, -1, 1, \dots, \mu_{out}\}$.

For each q_j , $\text{start}(q_j)$ is the index of the first vertex v_1 in its search path π_j .

Assume that the x^{th} vertex v_x of π_j is stored in register $v'(i)$ of processor $\text{PE}(i)$, and that the successor $\sigma(q_j, C_{v_x})$ of q_j in C_{v_x} has been determined; let $y = g(v'(i), q_j, \sigma(q_j, C_{v_x}))$. If $y < 0$ then $v.\text{predecessor}_y(i)$ is the index of the $(x+1)^{\text{st}}$ vertex v_{x+1} of π_j ; otherwise, $v.\text{successor}_y(i)$ is the index of the $(x+1)^{\text{st}}$ vertex.

It is required that both functions, start and g , can be calculated in, at most, $O(\log N)$ time.

Figure 6.5 The registers required at each processor $\text{PE}(i)$.

Figure 6.5 shows the set of registers necessary at every processor $PE(i)$. In addition to the registers mentioned above, every processor $PE(i)$ also has a register $v'(i)$ to store another vertex of G as well as other auxiliary registers $R(i)$, $N(i)$, $q'(i)$, $q''(i)$, $c'(i)$, $N'(i)$, $N''(i)$, $LS(i)$, $Shift(i)$ and $Dest(i)$.

6.3.2 Algorithm Overview

The global structure of the multi iterative search algorithm is described in Figure 6.6. The iterative search processes for all m queries $q_1, \dots, q_m$ are executed in p phases; each phase moves all queries one step along their search paths.

Multi Iterative Search:

- (1) Phase 1: Match every query with the 1st node in its search path and perform the respective catalog lookup.
- (2) For $x := 2$ to p do
- (3) Phase x : Match every query with the x^{th} node in its search path and perform the respective catalog lookup.

Figure 6.6 Global structure of the Multi Iterative Search algorithm

The general idea is again that, in Phase x ($1 \leq x \leq p$), instead of routing the queries to the respective nodes (possibly resulting in collisions), these nodes are duplicated and routed to the respective queries. In order to obtain the desired time complexity, the algorithm first permutes the queries (in registers $q(i)$) such that they are sorted with respect to the index of the x^{th} node in their search path. It then creates, in registers $v'(i)$, copies of the respective nodes such that each processor $PE(i)$ containing a query q_j in its register $q(i)$, contains in its register $v'(i)$ a copy of the x^{th} node in the search path of q_j (we will call this a *match* of q_j with the x^{th} node in its search path). Finally, for each node v all queries that have v as the x^{th} vertex in their search path, and C_v , are merged into one sorted list; for each query its successor in C_v is determined.

A typical situation at the end of a phase is depicted in Figure 6.7; each vertical column represents the registers $q(i)$, $N(i)$, $v'(i)$ and $c''(i)$ of a processor $PE(i)$.

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{15}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}
$c''.key(i)$	15	15	5	5	13	-1	-1	-1	11	14	-1	-1	8	-1	14	14

Figure 6.7 A typical situation at the end of a Phase.

In the following sections we will present some details of Phase 1 and Phase x ($2 \leq x \leq p$) respectively. The first phase is again different from the remaining phases because it starts with an arbitrary permutation of the queries.

6.3.3 Phase 1 of the Multi Iterative Search Algorithm

An outline of Phase 1 is given in Figure 6.8. The algorithm consists of five basic steps (see also Figure 6.9 for an illustration).

Phase 1:	
(1)	Every $PE(i)$: $N(i) := \text{Start}(q(i))$
(2)	$\text{Sort}(\{[q(i), q(i)], [N(i), N(i)]\}, N(i))$
(3)	$\text{MoveVerticesToQueries}$
(4)	SelectCatalogs
(5)	$\text{SearchCatalogsForQueries}$

Figure 6.8 Outline of Phase 1.

Algorithm Phase₁:

Input: The vertices of the catalog graph $v(i)$, and the set of catalog elements $c(i)$.

Output: Every processor $PE(i)$ stores a query $q(i)$, a copy of the first vertex on $q(i)$ path $v'(i)$, and the result of the catalog search $c''(i)$. Note the queries $q(i)$ are now sorted with respect to $N(i)$.

Method: (See Figure 6.8) First, in Step 1, every processor $PE(i)$ calculates the index of the first node in the search path of its query $q(i)$ and stores this value in the register $N(i)$. Then in Step 2, the queries are sorted by the index of the first node in the search path, i.e. $N(i)$. In Step 3, the source nodes are copied to the queries for which they are the first node in their search path. Finally, in Steps 4 and 5, the catalogs associated with the current vertices are selected and, for each query, the successor record in the respective catalog is determined ♦

v.index(i)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v.endCat(i)	0	2	5	-1	-1	8	-1	-1	13	-1	14	15	-1	-1	-1	-1
c.value(i)	15	5	13	2	6	9	2	11	14	5	6	7	8	9	4	14
c.index(i)	0	1	1	2	2	2	5	5	5	8	8	8	8	8	10	11
q(i)	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅	q ₆	q ₇	q ₈	q ₉	q ₁₀	q ₁₁	q ₁₂	q ₁₃	q ₁₄	q ₁₅

N(i)	7	1	3	11	1	0	5	0	8	11	1	3	7	5	1	8
------	---	---	---	----	---	---	---	---	---	----	---	---	---	---	---	---

q(i)	q5	q7	q1	q4	q10	q14	q2	q11	q6	q13	q0	q12	q8	q15	q3	q9
N(i)	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11

$\mathbf{v}'(i)$	$\mathbf{v}_0$	$\mathbf{v}_0$	$\mathbf{v}_1$	$\mathbf{v}_1$	$\mathbf{v}_1$	$\mathbf{v}_1$	$\mathbf{v}_3$	$\mathbf{v}_3$	$\mathbf{v}_5$	$\mathbf{v}_5$	$\mathbf{v}_7$	$\mathbf{v}_7$	$\mathbf{v}_8$	$\mathbf{v}_8$	$\mathbf{v}_{11}$	$\mathbf{v}_{11}$
------------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-------------------	-------------------

[illegible]

At the end of Phase 1: Every $q(i)$ is stored on a $PE(i)$ that also stores:
a copy of its vertex $v'(i)$, and a copy of its catalog element $c''(i)$.

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}
$c''.key(i)$	15	15	5	5	13	-1	-1	-1	11	14	-1	-1	8	-1	14	14

Figure 6.9 An illustration of Phase 1

Once the queries have been sorted by the index of the first vertex in their search path, the matching process between each query and the first node in its search path can be performed in time $O(\log N)$ using the procedure `MoveVerticesToQueries` described in Figure 6.10.

MoveVerticesToQueries:

- (1) `IdentifyBlockTail( $N(i), N'(i)$ )`
- (2) Every $PE(i)$: if $N'(i) = N(i)$ then $Dest(i) := -1$
else $Dest(i) := i$
- (3) Every $PE(i)$: $addr(i) := -1$
- (4) `Route( $\{[Dest(i), addr(i)]\}, N(i), N'(i) \neq N(i)$ )`
- (5) `RouteAndCopy( $\{[v(i), v'(i)]\}, addr(i), addr(i) \neq -1$ )`

Figure 6.10 Sketch of Procedure `MoveVerticesToQueries`.

This procedure is a generalized version of the procedure `MoveVerticesToQueries` described in Chapter 4. See Figure 6.11 for an illustration of the operation of this procedure.

The Initial State on `MoveVerticesToQueries`

$v(i)$	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11

After Step 1: If $N(i) \neq N'(i)$ then i is the tail of a block

$N'(i)$	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11	12
---------	---	---	---	---	---	---	---	---	---	---	---	---	---	----	----	----

After Step 2: Dest(i) = i for block tails otherwise -1

Dest(i)	-1	1	-1	-1	-1	5	-1	7	-1	9	-1	11	-1	13	-1	15
---------	----	---	----	----	----	---	----	---	----	---	----	----	----	----	----	----

After Step 3: Initialize all addr(i)

addr(i)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
---------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

After Step 4: addr(i) = Address of last q(i) requiring v(i)

addr(i)	1	5	-1	7	-1	9	-1	11	13	-1	-1	15	-1	-1	-1	-1
---------	---	---	----	---	----	---	----	----	----	----	----	----	----	----	----	----

After Step 5: v'(i) = a copy of the vertex required by q(i)

q(i)	q ₅	q ₇	q ₁	q ₄	q ₁₀	q ₁₅	q ₂	q ₁₁	q ₆	q ₁₃	q ₀	q ₁₂	q ₈	q ₁₅	q ₃	q ₉
N(i)	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
v'(i)	v ₀	v ₀	v ₁	v ₁	v ₁	v ₁	v ₃	v ₃	v ₅	v ₅	v ₇	v ₇	v ₈	v ₈	v ₁₁	v ₁₁

Figure 6.11 An example trace of MoveVerticesToQueries.

Algorithm MoveVerticesToQueries:

Input: A set of queries $q(i)$, sorted by their associated $N(i)$ register which stores the index of the next vertex required by each query.

Output: Each query $q(i)$ is matched with (i.e. on the same processor as) a copy of the vertex $v(i)$ with index $N(i)$.

Method: (See Figure 6.1) The idea is to identify for each node that needs to be matched (those with $\text{Dest}(i) \neq -1$), the largest address of the block of queries to be matched with (Steps 1 to 4), and then broadcast each node to the respective block of queries (Step 5). The time complexity of this operation is $O(\log N)$ since Steps 1, 4, 5 all take $O(\log N)$ time by Theorem 3.1 and Steps 2 and 3 take $O(1)$ time ♦

Lemma 6.2: *Algorithm MoveVerticesToQueries correctly matches a set of queries, sorted by the next vertex in their search path, with copies of those vertices in $O(\log N)$ time.*

Having matched each query $q(i)$ with a copy of the next vertex in its search path $v'(i)$, it is now necessary to perform the catalog look up. Before catalog look up search can be performed, the set of catalog elements associated with vertices currently being visited by queries must be identified. This operation is performed by the following algorithm SelectCatalogs.

SelectCatalogs:

- (1) IdentifyBlockTail($N(i), N'(i)$)
- (2) RouteAndCopy($\{[v'.index(i), Dest(i)]\}, v'.endCat(i), N(i) \neq N'(i) \text{ and } v'.endCat(i) \neq -1$)
- (3) Number($H(i), c.index(i) = dest(i)$)
- (5) Concentrate($\{[c(i), c'(i)]\}, c.index(i) = dest(i)$)

Figure 6.12 Sketch of Procedure SelectCatalogs.

The initial state on entering SelectCatalogs

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}
$v'.endCat(i)$	0	0	2	2	2	2	-1	-1	8	8	-1	-1	13	13	15	15
$c(i)$	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
$c.index(i)$	0	1	1	2	2	2	5	5	5	8	8	8	8	8	10	11

After Step 1: If $N(i) \neq N'(i)$ then i is the tail of a block

$N'(i)$	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11	12
---------	---	---	---	---	---	---	---	---	---	---	---	---	---	----	----	----

After Step 2: If $Dest(i) \neq -1$ then catalog element $c(i)$ has been selected

$Dest(i)$	0	1	1	5	5	5	5	5	5	8	8	8	8	8	11	11
-----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----	----

After Step 3: $H(i)$ is the total number of selected catalog elements														
$H(i)$	12	12	12	12	12	12	12	12	12	12	12	12	12	12

At the end of SelectCatalogs: $c'(i)$ = a copy of all selected catalog elements														
$c'(i)$	c_0	c_1	c_2	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{15}		

Figure 6.13 An example trace of Procedure SelectCatalogs.

Algorithm SelectCatalogs:**Input:** The set of catalog elements, $c(i)$.**Output:** The set of catalog elements $c'(i)$ associated with vertices currently being visited by queries.

Method: (See Figure 6.12) First, in Step 1, the tail of each block formed by register $N(i)$ is identified. Then, in Step 2, every catalog element associated with a vertex currently being visited by a query is selected. Lastly, the selected catalog elements are concentrated into register $c'(i)$ and counted. The time complexity of this algorithm is $O(\log N)$ since all steps all take $O(\log N)$ time by Theorem 3.1 ♦

Lemma 6.3: *Algorithm SelectCatalogs selects the set of catalog elements associated with vertices currently being visited by queries in $O(\log N)$ time.*

Having selected the set of required catalog elements, we can now perform the catalog search of each query $q(i)$ in its catalog. At the end of the search procedure, each $PE(i)$ storing a query $q(i)$ and a vertex $v'(i)$ will also store a catalog element $c''(i)$ which is the result of searching with query $q(i)$ in the catalog of vertex $v'(i)$. The catalog search is performed by algorithm SearchCatalogsForQueries.

SearchCatalogsForQueries:	
(1)	$q.currentIndex(i) := i$
(2)	$Merge(q(i), N, c'(i), H(0), r(i))$
(3)	$RouteAndCopy(\{[r(i), r'(i)]\}, i, r(i) = \text{a catalog element})$
(4)	$Route(\{[r'(i), c''(i)]\}, r.currentIndex(i), r(i) = \text{a query})$
(5)	Every PE(i): If $c''.index(i) \neq N(i)$ then $c''(i) := -1$

Figure 6.14 Sketch of Procedure SearchCatalogsForQueries.

Procedure SearchCatalogsForQueries is the last step in Phase 1. Figure 6.15 illustrates the operation of this procedure.

The initial state on entering SearchCatalogsForQueries

$q(i)$	q5	q7	q1	q4	q10	q15	q2	q11	q6	q13	q0	q12	q8	q15	q3	q9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$c'(i)$	c0	c1	c2	c6	c7	c8	c9	c10	c11	c12	c13	c15				
$c'.key(i)$	15	5	13	2	6	9	5	6	7	8	9	14				
$c'.index(i)$	0	1	1	5	5	5	8	8	8	8	8	11				

After Step 2: $r(i)$ (a simulated double length register) =
the result of merging registers $c'(i)$ and $q(i)$

$r(i) =$	q5	q7	c0	q1	q4	c1	q10	c2	q15	q2	q11	c6	q6	c7	c8	q13
	q0	q12	c9	c10	c11	q8	c12	c13	q15	q3	q9	c15				

∴

After Step 3: if $r(i)$ = a query then $r'(i)$ (a simulated double length register) =
the result of the search with $q(i)$ in the catalog of $v'(i)$.

$r'(i) =$	c0	c0	c0	c1	c1	c1	c2	c2	c6	c6	c6	c6	c7	c7	c8	c9
	c9	c9	c9	c10	c11	c12	c12	c13	c15	c15	c15	c15				

After Step 4: if $c''(i) = N(i)$ then $C''(i)$ the result of the search
with $q(i)$ in the catalog of $v'(i)$.

$c''(i)$	c0	c0	c1	c1	c2	c6	c6	c6	c7	c9	c9	c9	c12	c15	c15	c15
----------	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----

After Step 5: $c''(i)$ = the result of the search with $q(i)$ in the catalog of $v'(i)$.

$c''(i)$	c0	c0	c1	c1	c2	-1	-1	-1	c7	c9	-1	-1	c12	-1	c15	c15
----------	----	----	----	----	----	----	----	----	----	----	----	----	-----	----	-----	-----

Figure 6.15 An example trace of SearchCatalogsForQueries.

Algorithm SearchCatalogsForQueries:

Input: The set of queries $q(i)$ sorted by the index of the vertex they are currently visiting and the set of catalog elements of those vertices $c'(i)$ (also sorted).

Output: Register $c''(i)$ containing a copy of the catalog element found in the catalog search, if any. The value -1 will be used to indicate no catalog element found.

Method: (See Figure 6.14) First, in Step 1, the current location of each query is recorded in $q.\text{currentIndex}(i)$. In Steps 2 and 3, for each query $q(i)$ its successor in the associated catalog is computed. This is obtained by a bitonic merge of the queries and catalog elements into a simulated double length register $r(i)$. The principal keys for the merge are the indices of the vertices being visited by the queries' $v'(i)$ and the catalogs' indices $c'.\text{index}(i)$. The secondary keys are the queries' and catalogs' values $q.\text{key}(i)$ and $c'.\text{key}(i)$ respectively. When the merge step is finished, for every query its successor catalog element is the first catalog element to its right. As several queries may have the same successor, Step 3 is required to make copies of such catalog elements. In Step 4 the located catalog elements are routed back to the processor holding the queries that requested them. Some catalog searches may result in no catalog element being found for a particular query $q(i)$. In these cases Step 5 sets the register $c''(i)$ to -1. The time complexity for SearchCatalogsForQueries is $O(\log N)$, since all of its steps take at most $O(\log N)$ time (see Theorem 3.1) ♦

Lemma 6.4: *Algorithm SearchCatalogsForQueries finds for each query $q(i)$ its successor in the catalog of vertex $v'(i)$ in $O(\log N)$ time.*

6.3.4 Phase X ($2 \leq X \leq P$) of the Multi Iterative Search Algorithm

The purpose of each subsequent phase is again to advance, in time $O(\log N)$, all queries by one step in their search paths. After Phase $x-1$ has been completed, all queries are sorted (in registers $q(i)$) with respect to the index of the $(x-1)^{\text{th}}$ node in their search path. Each processor $PE(i)$ contains in its register $v'(i)$ a copy of the $(x-1)^{\text{th}}$ node in the search path of the query stored in $q(i)$. In register $c''(i)$, $PE(i)$ stores a copy of the successor catalog element of query $q(i)$ in catalog $C_{v'(i)}$. The desired effect of Phase x is to have all queries sorted (in registers $q(i)$) with respect to the index of the x^{th} node in their search path, and have each processor $PE(i)$ contain (in its register $v'(i)$) a copy of the x^{th} node in the search path of query $q(i)$ and (in register $c'(i)$) a copy of the successor of $q(i)$ in $C_{v'(i)}$.

Phase x , $2 \leq x \leq p$:

(1) Every PE(i): $R(i) := g(v'(i), q(i), c'(i));$

If $R(i) > 0$

THEN $N(i) := v'.\text{successor}_{R(i)}(i)$

ELSE $N(i) := v'.predecessor_{R(i)}(i)$

(2) OrderQueriesByNextVertex

(3) MoveVerticesToQueries

(4) **SelectCatalogs**

(5) SearchCatalogsForQueries

Figure 6.16 Sketch of Phase x , $2 \leq x \leq p$.

Algorithm Phase(x):

Input: The vertices $v(i)$ of the catalog graph, the set of catalog elements $c(i)$, and a set of queries $q(i)$ sorted with respect to the index of the $(x-1)^{\text{th}}$ node in their search path

Output: Every query $q(i)$ is stored on a processor $PE(i)$ with a copy of the x^{th} vertex on $q(i)$'s path, and the result of the catalog search is stored in $c''(i)$. Note that, the queries $q(i)$ are sorted with respect to $N(i)$.

Method: (See Figure 6.16) First (in Step 1), every PE(i) computes for the query currently stored in its register $q(i)$ which edge to use for the next step in the search path as well as the

At the end of Phase 1: Every $q(i)$ is stored on a $PE(i)$ that also stores a copy of its vertex $v'(i)$, and a copy of its catalog element $c''(i)$.

$q(i)$	q_5	q_7	q_1	q_{13}	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_9	q_{12}	q_{15}	q_8	q_3
$N(i)$	2	2	3	4	5	5	5	5	5	5	8	8	9	10	11	14
$v'(i)$	v_2	v_2	v_3	v_4	v_5	v_5	v_5	v_5	v_5	v_5	v_8	v_8	v_9	v_{10}	v_{11}	v_{14}
$c''.key(i)$	2	2	-1	-1	2	2	11	14	14	-1	7	9	-1	4	14	-1

Figure 6.17 Example of Phase $x=2$.

What remains to be discussed is procedure `OrderQueriesByNextVertex`. This procedure, which is sketched in Figure 6.18, creates in time $O(\log N)$ the new ordering of the queries with respect to the indices of the next nodes in the search paths. It is a generalized version of the procedure by the same name described in Chapter 4. Instead of restricting queries to always moving from one level of the graph to another, as in Chapter 4, this procedure allows queries to move along any edge to any connected vertex. See Figure 6.18 for a sketch of `OrderQueriesByNextVertex`.

OrderQueriesByNextVertex:	
(1)	Every $PE(i)$: $Shift(i) := 0$
(2)	FOR $r := -\mu_{in}, \dots, -1, 1, \dots, \mu_{out}$ DO
(3)	Concentrate($\{[q(i), q'(i)], [N(i), N'(i)]\}, G(i)=r$)
(4)	Number($LS(i), G(i)=r$)
	$ls := LS(0)$
	$shift := Shift(0)$
(5)	Reverse($\{[q'(i), q''(i)], [N'(i), N''(i)]\}, 0, ls$)
(6)	Route($\{[q'(i), q''(i)], [N'(i), N''(i)]\}, i+shift, i < ls$)
(7)	BitonicMerge($\{[q''(i), q'''(i)], [N''(i), N'''(i)]\}, N''(i), 0, shift, shift+ls$)
	Every $PE(i)$: $Shift(i) := Shift(i) + LS(i)$
(8)	Every $PE(i)$: $q(i) := q'''(i), N(i) := N'''(i)$

Figure 6.18 Sketch of Procedure `OrderQueriesByNextVertex`.

Algorithm `OrderQueriesByNextVertex`:

Input: A set of n unsorted queries $q(i)$ each with an associated $N(i)$ specifying the index of the next vertex in each queries search path.

Output: The n queries $q(i)$ along with their $N(i)$ sorted with respect to $N(i)$.

Method: We first consider all forward edges to next vertices in the search paths; the backward edges are handled analogously. Let (v, w) and (v', w') be two such edges for two queries q and q' , respectively, with the property that $g(v, q, \sigma(q, C_v)) = g(v', q', \sigma(q', C_{v'}))$. From the monotonicity of G it follows that if $\text{Index}(v) < \text{Index}(v')$ then $\text{Index}(w) \leq \text{Index}(w')$. Therefore, the sub-sequence of queries q for which $g(v, q, \sigma(q, C_v))$ has the same value r is already sorted with respect to the index of the next vertex. Furthermore, since each node has an outdegree of at most μ_{out} , there are at most $\mu_{\text{out}} = O(1)$ such sub-sequences. The new ordering of the queries can therefore be created in time $O(\mu_{\text{out}} \log N) = O(\log N)$ by extracting these μ_{out} ordered sub-sequences and merging them in μ_{out} bitonic merge steps.

For the backward edges, the same idea applies because a monotone graph has the same monotonic properties forwards or backwards. Thus, the same steps described above can be used in order to sort the backward queries with respect to the indices of the next nodes in the search path. As in the previous case, the time for sorting them is $O(\mu_{\text{in}} \log N) = O(\log N)$, where μ_{in} is the constant maximum in-degree in the graph G .

The algorithm is sketched in Figure 6.18: for each of the $\mu_{\text{in}} + \mu_{\text{out}}$ possible values of $R(i)$, the respective sub-sequence of queries is extracted (Step 3), inverted (Step 5), appended to the sequence of queries already ordered (Step 6), and finally the newly created bitonic sequence is converted into a sorted sequence by a bitonic merge (Step 7). Since this algorithm consists of a constant number of $O(\log N)$ operations it has a time complexity of $O(\log N)$ ♦

Lemma 6.5. *Algorithm `OrderQueriesByNextVertex` creates in time $O(\log N)$ a sorted ordering of the queries $q(i)$ with respect to the indices of the next nodes in the search paths $N(i)$.*

Note that all the sub algorithms used in $\text{Phase}(x)$ require $O(\log N)$ time (Lemma 6.1 through 6.5) so that the time complexity of algorithm $\text{Phase}(x)$ as a whole is $O(\log N)$. Given this analysis and Lemma 6.1 through 6.1 we can now assert the following:

Lemma 6.6. *Algorithm $\text{Phase}(x)$ advances all queries $q(i)$ one step in their search in time $O(\log N)$.*

6.4 CONCLUSION

Given Lemma 6.1 through 6.6 we can now state the following property and theorem:

Property 6.1. *The multi iterative search algorithm consists of p phases such that at the end of Phase x ($1 \leq x \leq p$) all queries are sorted (in registers $q(i)$) with respect to the index of the x^{th} node in their search path, and each processor $PE(i)$ contains in its registers $v'(i)$ and $c'(i)$ a copy of the x^{th} node v_x in the search path of $q(i)$ and a copy of the successor record of $q(i)$ in C_{v_x} .*

Theorem 6.1. *For a monotone catalog graph of size n (and bounded degree), m iterative search queries along paths of length at most p can be executed independently, in parallel, in time $O((p + \log N) \log N)$ on a hypercube multiprocessor of size N ; $N = \max\{n, m\}$.*

7 AN APPLICATION OF PARALLEL FRACTIONAL CASCADING

In this chapter we present an example of how parallel fractional cascading can be applied to solve geometric problems. Using our parallel iterative search algorithm (for multiple queries) and a parallel data structure which will be described, we obtain a highly efficient hypercube solution for determining all intersections of m lines with a simple polygonal path of length n . This problem is an example of one of a large class of geometric stabbing problems, for the sake of brevity we will refer to it as the multiple stabbing problem. The hypercube algorithm which we will present solves the multiple stabbing problem in $O(N)$ space and $O((k \log \frac{n}{k} + \log N) \log N)$ time, where k is the maximum number of intersections between one of the m query lines and the polygonal path of length n , and $N = \max\{n, m\}$.

Our solution is elegant in that it applies an advanced data structure without the need for extensive new routing methods as would be required by a direct hypercube solution. On the other hand, our algorithm is highly efficient: its time complexity differs only by a factor of $\log N$ from the best known PRAM solution⁸, as well as the best known sequential solution for the single query line version of the problem.

We will first review Chazelle and Guibas' [CG2] sequential algorithm for determining the intersections of one query line with a polygonal path of length n . This algorithm is based on sequential fractional cascading. We will then show how parallel fractional cascading can be applied to generate the m query line parallel solution to this stabbing problem on a hypercube multiprocessor.

⁸ A factor of $\log N$ better than standard simulation.

7.1 REVIEW OF CHAZELLE AND GUIBAS' SEQUENTIAL STABBING ALGORITHM

Consider the problem of determining, for a given simple polygonal path P of length n , all intersections with a given query line l , as illustrated in Figure 7.1.

Figure 7.1 Intersections of a line with a simple polygonal path of length N

Solving this stabbing problem in $O(n)$ for a given P and line l is trivial. This problem becomes much more interesting if we have many query lines $l_1, l_2, \dots, l_k$ and wish to successively test each of them for intersection with P as efficiently as possible. Chazelle and Guibas [CG2] have presented an $O(n)$ space, $O(n \log n)$ preprocessing time, and $O(k \log \frac{n}{k})$ query time sequential solution (where k is the number of reported results) based on sequential fractional cascading. We will first describe their data structure and algorithm and then show that with minor modifications it meets the requirement for parallel fractional cascading.

The central idea behind their algorithm is stated in the following lemma:

Lemma 7.1 (Chazelle & Guibas) *A straight-line l intersects a polygonal line path P if and only if l intersects the convex hull, $CH(P)$, of P .*

Proof. *Obvious*

Chazelle and Guibas [CG2] build a data structure based on convex hulls. Convex hulls can be represented by their slope sequence defined as follows. Let $c_1, c_2, \dots, c_x$ be the vertices of a convex polygon C in clockwise order, and let $c_i x$ be the horizontal ray starting at c_i in the direction of $+x\infty$. The slope of an edge $c_i c_{i+1}$ is the angle between $c_i x$ and $c_i c_{i+1}$; as illustrated in Figure 7.2.

In general, $\text{slope}(c_i) = \angle((c_i, c_i + x\infty), (c_i, c_{i+1}))$

Figure 7.2 The slopes of edges of a convex polygon

The slope-sequence of a convex polygon C is the ordering of the edges of C that forms the unique nondecreasing sequence of edge slopes. For example, the slope sequence of the convex polygon shown in Figure 7.2 is c_1, c_2, c_3, c_4, c_5 . Given a convex hull represented by its slope sequence, Chazelle and Guibas [CG2] prove the following lemma:

Lemma 7.2 (Chazelle & Guibas) *Let s and s' be the two slopes of a line l obtained by giving l its two possible orientations; if we know the position of s and s' within the slope-sequence of a convex polygon C , we can determine whether C and l intersect in constant time.*

The intersection algorithm referred to in Lemma 7.2 can be implemented as follows. First calculate s and s' , the two slopes of l (as depicted in Figure 7.3). Then locate these slopes in the slope-sequence of C , identifying c_i and c_j the two vertices that fall immediately after s and s' in the slope-sequence.

Figure 7.3 Intersection testing by construction of parallel supporting lines.

The query line l intersects C if and only if l falls between the supporting lines parallel to l that intersect c_i and c_j ; as shown in Figure 7.3.

Consider a convex hull decomposition tree for a polygonal path P as defined by Chazelle and Guibas in [CG2]. Let $F(P)$ and $S(P)$ denote respectively the first and second halves of

the path P of length n , that is, the subpaths of P consisting of the first $n/2$ and second $n/2$ edges. A convex hull decomposition tree, T , for a polygonal path P , is a binary tree whose root node stores the convex hull, $CH(P)$, and whose left and right subtrees recursively represent $F(P)$ and $S(P)$, respectively.

Figure 7.4 A convex hull decomposition tree

A convex hull decomposition tree, as illustrated in 7.4, is a binary tree whose leaves are the edges of the path P (and trivially their own convex hulls), and whose internal nodes correspond to convex hulls of subpaths of P in the obvious way.

The convex hull decomposition tree for a polygonal path P can be constructed in time $O(n \log n)$ by a simple recursive algorithm that constructs $CH(P)$ by merging $CH(F(P))$ and $CH(S(P))$ in linear time [PH]. This algorithm relies on the fact that the convex hull of a set of lines is equivalent to the convex hull of the lines' end points.

Having constructed a convex hull decomposition tree T for a polygonal path P , all intersections between P and a query line l can be reported by searching in P with query line l using the following algorithm:

Algorithm Report Intersections:

Input: A pointer, p , to the root of a convex hull decomposition tree T for a polygonal path P , plus a query line l .

Output: Each edge of P intersected by l is reported.

Method: If p points to a leaf node then determine whether l intersects the single edge stored at the leaf and report intersection, if any; else test if l intersects $CH(p)$. If l does intersect the convex hull $CH(p)$ then recursively call this algorithm for the children of p , else exit. ♦

The convex hull decomposition tree and the Report Intersections algorithm described above are suitable candidates for sequential fractional cascading. The convex hulls at each node in the tree stored as slope sequences meet the requirements for catalog elements of a sequential fractional cascading data structure. The branch and bound type searching described recursively by the Report Intersections algorithm also meets the requirements of sequential fractional cascading. Hence, the time complexity of the report intersections algorithm can be seen to be $O(\log n + |T'|)$, where $|T'|$ is the size of the subtree T' of T traversed by the query line l in reporting its intersections with P (excluding internal nodes where the intersection test fails). Since Chazelle and Guibas wish to prove the time complexity of this algorithm in terms of k , the number intersection to be reported, they must be able to define $|T'|$ in terms of k . Figure 7.5 shows a typical subtree T' (shown by gray lines) of T .

Figure 7.5 The tree T' (shown in gray) with k leaves is subtree of T .

Lemma 7.3: (Chazelle & Guibas) Consider the subtree T' of T which includes all leaves of T storing edges of P that are intersected by l . $|T'| \leq O(k \log \frac{n}{k})$, where k is the number of intersections of l with P .

Proof: See [CG2]Section 2.

Given Lemma 7.3, we have an upper bound on the time complexity of the algorithm. Given a convex hull decomposition tree of a polygonal path of length n , all intersections with a line l can be calculated in time $O(k \log \frac{n}{k} + \log n)$. Note that the $\log n$ component of this time complexity is the time it takes to perform the original binary search in the catalog of the root node which starts the fractional cascading process.

Lemma 7.4: (Chazelle & Guibas) Given a polygonal path P of length n , it is possible in time $O(n \log n)$ to build a data structure of size $O(n \log n)$, so that given any line l , if l intersects k edges of P , then these edges can be found and reported in time $O(\log n + k \log \frac{n}{k})$.

The algorithm described above solves the stabbing problem for an arbitrary polygonal path P . If we wish to consider only simple polygonal paths, it is possible to reduce the size of the fractional cascading data structure from size $O(n \log n)$ to size $O(n)$. The key idea relies on the following lemma; see Figure 7.6.

Lemma 7.5: (Chazelle & Guibas) *If P is simple, then $CH(F(P))$ and $CH(S(P))$ have at most two common tangents.*

Proof: See [CG2 pp170-171]

Figure 7.6 Two convex hulls have at most two common tangents

Lemma 7.5 implies that the number of different edges in a convex hull decomposition tree is linear ($4n-2$). We therefore redefine the structure of the convex hull decomposition tree as follows: every convex hull edge is stored only once at the highest possible node (see Figure 7.7). Clearly, the resulting tree has linear space requirements.

Lemma 7.6: (Chazelle & Guibas) *If we store each edge of a convex hull only at the highest possible node of P 's convex hull decomposition tree, then all edges stored at a particular node form a contiguous interval of edges of the respective convex hull.*

Proof: See [CG2 pp171-172]

Figure 7.7 Edges (depicted in bold) are stored only once at the highest point they appear

With only slight modification, the Report Intersection algorithm described earlier can be made to function on the new data structure. The decisions made at each leaf node reached during the search remains the same, but the decision made at each internal node differs, since only part of a convex hull is stored at such a node.

Consider the decision to be made at an internal node of the new convex hull decomposition tree. Such a node stores only a subset of contiguous edges of the respective convex hull. The situation is as shown in Figure 7.7 where the stored edges are depicted in bold.

Given that only a subhull of the original convex hull is stored at an internal node, can a correct intersection test be made in reference to the original hull? There are three cases to be considered (see [CG2 p. 172]).

Figure 7.8 Three intersection cases

If the line intersects the subhull stored at a node (case A) then it clearly intersects the complete convex hull. Otherwise, one of the following two cases must occur. Case B: the query line intersects neither the subhull nor the complete hull. Case C: the query does not intersect the stored subhull but does intersect the complete hull. We can distinguish between these cases by locating the two slopes of the query line in the slope sequence of the stored subhull.⁹ If the query line's slope is larger or smaller than all of the hull segment's slopes, then Case C applies (otherwise the current node's parent would not have passed this query down). If neither Case A nor Case C applies, then Case B must apply.

Lemma 7.6: (Chazelle & Guibas) *Given a simple polygonal path P of length n , it is possible in time $O(n \log n)$ to build a data structure of size $O(n)$, so that given any line l , if l intersects k edges of P , then these edges can be found and reported in time $O(\log n + k \log \frac{n}{k})$.*

⁹ Note this operation has already been performed by the intersection test performed for case A.

7.2 A PARALLEL ALGORITHM FOR DETERMINATION OF ALL INTERSECTIONS OF M LINES WITH A SIMPLE POLYGONAL PATH OF LENGTH N

This section presents a parallel algorithm for determining all intersections of m lines with a simple polygonal path of length n ; see Figure 7.9.

Figure 7.9 The multiple stabbing problem

The parallel hypercube algorithm described in this section will solve the multiple stabbing problem by applying parallel fractional cascading. We will solve the polygonal path stabbing problem for $m=O(n)$ query lines as opposed to the $m=1$ query line problem solved by sequential fractional cascading. In order to apply parallel fractional cascading to this problem we will have to 1) Build the convex hull decomposition tree as a parallel data structure meeting the requirements of parallel fractional cascading, and 2) demonstrate a report intersections algorithm that meets the requirements of parallel fractional cascading.

Algorithm Construct Parallel Convex Hull Decomposition Tree:

Input: The n edges of a simple polygonal path sorted by slope.

Output: A convex hull decomposition tree represented as a catalog graph meeting the requirements of parallel fractional cascading as specified in Chapter 6.

Method: First, construct a binary catalog tree of size $2n-1$ with empty catalogs. Then form, at the leaf nodes of the catalog tree, a set of queries consisting of the n line segments sorted by slope. The algorithm now proceeds in $\log n$ phases as queries are moved up the tree. In each phase a node may be visited by queries from its left and right children. The set of queries from any child form a convex hull. Upon arriving at a node the block of queries from the left and right children are merged to form a single convex hull as described in [ACGOY]. The $O(\log n)$ merging of two convex hulls can be implemented in terms of the bitonic merge operation described in Chapter 3. This process creates six types A,B,C,D,A',C' of queries (lines) as shown in Figure 7.10.

Figure 7.10 Identifying the merge points of two convex hulls

Type A queries are returned to the node's left child to form its catalog elements. Type C queries are returned to the node's right child to form its catalog elements. The remaining queries are compressed into a single sequence of lines sorted by slope and sent to the node's parent. If the node is the root node, the merged sequence is stored as its catalog.

Since the maximum path length of any query in this algorithm is $O(\log n)$ and $O(\log n)$ work is performed at each step the whole algorithm can be completed in time $O(\log^2 n)$ ♦

Theorem 7.1. *A convex hull decomposition tree for a simple polygonal path of length n that meets the requirements of parallel fractional cascading can be constructed on a hypercube multiprocessor of size n in time $O(\log^2 n)$.*

What remains to be shown is that a report intersections algorithm can be designed that satisfies the conditions for parallel fractional cascading. Clearly, the sequential Report Intersection algorithm given in the previous section can not be directly parallelized, since each of its "queries" reports only one intersection. Such an approach in a parallel setting would result in the creation of $O(n^2)$ queries, in the worst case. Hence, rather than having queries report only one intersection, each query will report all intersections of a query line with the simple polygonal path.

Consider T' the subtree of the convex hull decomposition tree T that must be traversed to report all intersections of a query line l with the polygonal path P ; see Figure 7.11.

Figure 7.11 An inorder traversal of the subtree T' of T

Next we show that for a line l , instead of traversing T in a branch-and-bound fashion as in [CG2], we can also traverse T in an inorder traversal with exactly the same type of decision to be made at every node. Such a traversal will meet the requirements for parallel fractional cascading on a hypercube described in Chapter 6.

Algorithm Report Intersections:

Input: A convex hull decomposition tree T for a simple polygonal path P , plus m query lines.

Output: For each query line l each edge of P intersected by l is reported.

Method: The general approach will be to apply parallel fractional cascading where sequential fractional cascading was applied in the previous sequential Report Intersections algorithm. Start all the queries at the root of a convex hull decomposition tree, T . Note that a query at any node v can check if it intersects the convex hull stored at that node in exactly the same way as in the sequential algorithm. If v is a leaf, the intersection of the line stored at the leaf and a query can be calculated directly and reported. All that remains is to define the function g used by a query to decide which node to visit next. Let g compute, from an intersection result, and an incoming direction, the outgoing direction of a query as defined in Figure 7.12.

Incoming Direction	No intersection	Intersection
1	4	5
2	N/A	6
3	N/A	4

Figure 7.12 Defining g to be an inorder traversal of the tree

For example, if q most recently visited the parent of a node (ie. incoming direction = 1) and q intersects the convex hull stored at the current node, then q should next visit the node's left child (ie. outgoing direction = 6). Function g causes a query to perform an inorder traversal of the subtree T' of T that contains all intersected edges as leaf nodes. Note, that since each query q visits a node in T' at most three times, the length of q 's path is of the same order as in the sequential algorithm ♦

7.3 CONCLUSION

Applying Theorem 6.1, we obtain

Theorem 7.1. *Given a simple polygonal path P of length n and a set of m arbitrary query lines then, for all query lines, all intersections with P (with a maximum of k results per query) can be determined on a hypercube multiprocessor of size N in time $O((k \log \frac{n}{k} + \log n) \log n)$; $N = \max\{n, m\}$.*

Note that we assume that at the end of each phase of parallel fractional cascading, every processor can report a result without having to store it. Otherwise, after each phase the reported results need to be concentrated in order to obtain an even data distribution, and the number of processors and time complexity increase to $O(n+M)$ and $O((k \log \frac{n}{k} + \log n) \log(n+M))$, respectively, where M denotes the total output size for all queries.

8 PARALLEL BRANCH AND BOUND

Branch-and-bound (B&B) search methods such as A* search, Alpha-Beta search, depth-first and best-first search are used in areas such as VLSI design, theorem proving, linear programming, chess playing, and many other Artificial Intelligence and Operations Research applications ([Win], [Nil]). Since most of the addressed problems are NP-hard, and therefore the size of the search space grows exponentially, many researchers have studied the parallelization of B&B methods (e.g. [HD], [New], [MC], [UYII], [WLY]). So far, studies have been aimed at using coarse-grained multiprocessors for parallel B&B implementations.¹⁰ Particular attention has recently been given to coarse-grained hypercube multiprocessors such as the FPS hypercube, NCUBE, or Intel iPSC (e.g. [LW], [Qui], [AC], [AM], [Fel], [PW], [SGB], [TM]).

Parallel B&B methods for coarse-grained multiprocessors aim at splitting the tree searched by the B&B method into subtrees so that subtrees can be searched in parallel by different processors. The essential differences among the proposed coarse-grained algorithms are how they deal with the two major problems arising from such an approach:

(1) Load balancing: When a subproblem (a subtree to be searched) is assigned to an individual processor, the size of the subtree is not known in advance. Therefore, the sizes of the problems assigned to individual processors may vary significantly and this unbalanced distribution of work load may result in performance degeneration.

(2) Global information: For sequential B&B methods, the pruning of the search tree often depends on global information such as the best solution found so far. On a

¹⁰ These multiprocessors are comprised of a small number ($<2^{10}$) of powerful processors, each having a considerable amount of memory.

multiprocessor, the distribution of global information causes additional computational overhead or, if information is not completely distributed, the parallel method may search more nodes than the respective sequential algorithm (unnecessary search).

In this chapter, we study parallel B&B on fine-grained hypercube multiprocessors (hypercubes with a large number, more than 2^{10} , of small processors). The Connection Machine is an example of an existing fine-grained system. In the context of parallel B&B, fine-grained multiprocessors obviously have the advantage of a larger number of processors and increased parallelism. However, each processor in a fine-grained system has only a small constant amount of memory available. For any node p of the search tree let the back-up path of p be the path from p to the root of the search tree. In a fine-grained multiprocessor, it is impossible for one processor to store the back-up path even for the one single node that it is currently examining. Therefore, the above mentioned coarse-grained parallel B&B methods cannot be applied to a fine-grained system because all these methods assume that every processor stores the back-up path for its currently processed node (at least back to the tree node where the process was created).

Furthermore, the much larger number of processors (and, therefore, concurrent search processes) makes it imperative, for fine-grained B&B algorithms, that global information (e.g., the current best solution) is continuously available at every processor since otherwise the amount of unnecessary search would become intolerable.

8.1 OVERVIEW

In the remainder of this chapter we describe a highly efficient B&B algorithm for fine-grained hypercube multiprocessors that solves the problem of storing all current back-up paths in one data structure distributed over a hypercube. Our method uses a global scheme

where all processors collectively store all back-up paths such that each processor needs to store only a constant amount of information. Instead of storing the individual back-up paths, we store a current back-up tree which is defined by the union of the back-up paths of all search tree nodes currently under examination. (This also ensures that paths in the search tree shared by several back-up paths are stored only once, thus providing optimal storage utilization.) At each iteration of the algorithm, all nodes of the current back-up tree may decide whether they need to create new children, be pruned, or remain unchanged. We describe an algorithm that, based on these decisions, updates the current back-up tree and distributes global information in $O(\log n)$ steps, where n is the size of the current back-up tree. This method also yields a dynamic allocation mechanism for allocating search processes to processors, and provides optimal load balancing. Even if very drastic changes in the current back-up tree occur, the performance of our load balancing mechanism does not deteriorate.

Note that the $O(\log n)$ overhead mentioned above is measured by considering only word size messages, exchanged between adjacent processors, as $O(1)$ operations. In this model, an $O(\log n)$ overhead is very small in practice; in fact, one single "PREF" operation (parallel inter-processor read) [TMC] in Connection Machine *LISP has a much larger time $O(\log^2 N)$ complexity, where N is the size of the hypercube multiprocessor.

The remainder of this chapter is organized as follows: Section 8.2 introduces a generalized definition of Branch and Bound. In Section 8.3, we will present our parallel B&B algorithm for fine-grained hypercube multiprocessors in terms of our generalized definition. Section 8.4 describes an example application of parallel branch and bound. Section 8.5 summarizes our results.

8.2 GENERALIZED BRANCH AND BOUND

In this chapter we consider a form of generalized branch and bound that can be defined in terms of five rules. Any particular B&B algorithm can be realized by providing a suitable implementation of each of the five rules given below [AC]. In Section 8.4 we will demonstrate the use of these rules by presenting an example of how they can be used to solve the n th order solitaire checkers problem.

- i) The cost rule: Given a leaf node in the search tree and its parent's cost, this rule defines the cost up to and including the leaf. We will restrict ourselves to positive cost rules where if v is an descendant of v' then $\text{cost}(v) \geq \text{cost}(v')$.
- ii) Bounding rule: Given a node in the search tree, this rule returns 1 if the node is no longer feasible (i.e., should be deleted) otherwise returns a 0.
- iii) Selection rule: Given a node in the search tree, this rule returns an integer specifying the number of children this node should create in this iteration.
- iv) Expansion rule: Given a node, an integer specifying the number of children to be created, and a pointer to where they should be created, this rule creates the new children.
- v) Termination rule: Given global information, this rule returns true if a satisfactory solution has been found or all possibilities have been explored.

8.3 AN ALGORITHM FOR PARALLEL BRANCH AND BOUND WITH GLOBAL INFORMATION ON A FINE-GRAINED HYPERCUBE MULTIPROCESSOR

A branch and bound algorithm searches in the space of all the feasible solutions for a given problem. These feasible solutions are usually seen as a search tree S over the solution space. In parallel B&B algorithms, the search in S for an optimal or satisfactory solution is

performed concurrently at several nodes of S . These nodes are referred to as *active nodes*. The subtree T of S defined by the union of the back-up paths of all active nodes is the *current back-up tree* as introduced in Section 8.1 (an example of a current back-up tree is shown in Figure 8.1). Note that any node of the current back-up tree can be an active node, which allows for the implementation of search procedures that expand internal nodes as well as leaf nodes. This is essential for the implementation of some search procedures, for example best-first search.

Figure 8.1 A current back-up tree (bold nodes).

In this section, we will describe a parallel B&B algorithm for fine-grained hypercube multiprocessors. Our method stores the current back-up tree so that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all active nodes of the current back-up tree decide whether they need to create new children, be pruned, or remain unchanged. Based on these decisions, the algorithm described below updates a current back-up tree of n nodes in $O(\log n)$ steps. This method also yields a dynamic allocation of search processes to processors, such that the work load balancing problem is solved optimally.

8.3.1 Algorithm Overview

We assume that every processor $PE(i)$ has a constant size register, $node(i)$, to store one node of the current back-up tree, T . Register $node(i)$ will contain a record with fields $node.data(i)$, $node.cost(i)$, $node.parent(i)$, $node.children(i)$, $node.firstChild(i)$, $node.level(i)$ and $node.newChildren(i)$. Each of these fields stores a constant amount of data associated with the node $node(i)$, its cost, parent, number of children, position of first child, level, and number of new children, respectively. See Figure 8.2 for a list of registers required at every processor.

Figure 8.2 The registers required at each processor $PE(i)$.

The current back-up tree, T , is stored on the hypercube as follows:

Consider the level ordering of the nodes of T as shown in Figure 8.3. Each node v of T is stored in register $node(j)$ of processor $PE(j)$, where j is the index of v with respect to the level ordering of T .

Figure 8.3 The level ordering of the current back-up tree

The global structure of the B&B algorithm is described in Figure 8.4. The back-up tree T starts as a single root node stored on processor $PE(0)$. The main loop of the algorithm iterates until the termination rule returns true.

Branch-And-Bound:

- (A) Initialize the current back-up tree to be the single root node in processor $PE(0)$. Initialize $node.data(0)$ and set all other fields of $node(0)$ to 0. Also create a dummy node at $PE(1)$ with $node.parent(1) = \infty$.
- (B) Repeat Until termination rule returns true.
 - (1) Every $PE(i)$ storing a newly created node uses the evaluation rule to compute $node.cost(i)$ and global information is distributed to all processors.
 - (2) Every $PE(i)$ uses the bound rule to set $alive(i) = 1$ if the node is still feasible and else $alive(i) = 0$.
 - (3) Every $PE(i)$ uses the selection rule to set $node.newChildren(i) = \text{number of children to be created}$.
 - (4) Procedure `UpdateTree` is called which creates from the old back-up tree a new back-up tree with all non-feasible nodes removed and new nodes added.
 - (5) Every $PE(i)$ uses the expansion rule to load for each $node(i)$ the data into its $node.newChildren(i)$ new children.
- (C) Report best node and the path from the root to the best node.

Figure 8.4 The global structure of algorithm Branch-And-Bound

Overview of Algorithm Branch and Bound:

Input: The root node of the search tree and appropriate implementations of the five branch and bound rules.

Output: The first solution that meets the conditions of the termination rule.

Method Overview: The global structure of our method is shown in Figure 8.4. In each iteration, the algorithm updates the tree by adding new nodes in a manner prescribed by the selection and expansion rules, while at the same time pruning the tree using the cost and bound rules. A single pass through the main loop consists of five steps. First, the evaluation rule is used to calculate the cost function for all the nodes that were created in the last iteration. The evaluation rule may also maintain global knowledge concerning the progress of the search. For example, after assigning new costs to nodes, the evaluation rule may calculate a global minimum cost to be used by the bound and termination rules. The maintenance of such global knowledge does not increase the fundamental time complexity of the algorithm, since such global knowledge can be calculated by a global minimization and broadcast procedure in time $O(\log n)$, which is the same time complexity as the `updateTree` algorithm (Lemma 8.1).

The second step in the main loop of the algorithm uses the bound rule to identify nodes to be deleted in this iteration. Any particular implementation of the bound rule must never delete a parent node without also deleting its children. The parallel bounding of a large number of nodes is a particularly effective aspect of our fine-grained B&B algorithm since, with very low overhead, a large number of nodes in the back-up tree may be deleted and the data structure compacted.

In the third step, the selection rule is used to assign to each node `node(i)` an integer, `node.newChildren(i)`, that indicates the number of children the node needs to create during this iteration. The selection rule may also use global information to instruct each node exactly which of its possible children should be created.

Procedure `UpdateTree` (Step 4), represents the main part of our algorithm. In Steps 1 through 3 all the information needed to extend the back-up tree to its next state has been collected. Now, the back-up tree must be transformed by deleting the bounded nodes and

creating space for the new leaf nodes. This operation is performed by procedure `UpdateTree` which is described in detail in Section 8.3.2.

Finally, in Step 5, the new leaf nodes are initialized by the expansion rule. At this point each node has information about how many new children should be created, and about the address of the free processors allocated for them by procedure `UpdateTree`. Therefore, Step 5 can be implemented with an $O(\log n)^{11}$ time complexity by using a `RouteAndCopy` operation to copy the data of each node creating new children into the processors storing these children and then creating for each child in parallel its actual data set locally at the respective processor ♦

8.3.2 Updating the Back-Up Tree

The core of our parallel B&B algorithm is procedure `UpdateTree` which updates the back-up tree by deleting infeasible nodes and creating space for new leaf nodes to be initialized by the expansion rule. Note that the nodes of the new back-up tree created by procedure `UpdateTree` must again be stored in level order to maximize storage space utilization and performance. This makes it necessary to compute for each node, v , its address in the new back-up tree, in order to be able to relocate it. It is this relocation of nodes that provides the optimal task allocation mechanism for solving the load balancing problem.

The procedure `UpdateTree` calculates for each node the address of its first child on the hypercube after the update is completed. Each parent then broadcasts this information to all of its children. Since all nodes may move, all parent nodes must also broadcast their own new address to their children so that the children can update their parent pointers. When

¹¹ Note that since nodes of our back-up tree are always stored on a subcube of the hypercube (ie. all of our data is left justified on the hypercube), we have been able to state our time complexity results in terms of n the size of our problem, rather than the size of our full cube N .

these steps have been completed, all non-deleted nodes are routed to their new locations. This route operation will leave the necessary spaces for the new children that are to be created by the expansion rule called in the main procedure. Procedure UpdateTree is described in detail in Figure 8.5.

UpdateTree:

- 1.0) BlockPsum(alive(i), alive'(i), node.parent(i))
Route(alive'(i), node.children(i), node.parent(i), node.parent(i) $\neq$ node.parent(i + 1))
Every PE(i): node.oldChildren(i) := node.children(i)
Every PE(i): node.children(i) := node.children(i) + node.newChildren(i)
- 2.0) Psum(node.children(i), x')
Each PE(i): x'(i) := x'(i) + 1
IdentifyBlock(node.level(i), endOfLevel(i))
Every PE(i): nextEndOfLevel(i) := endOfLevel(i + 1)
RouteAndCopy(x'(i), x(i), nextEndOfLevel(i), i = endOfLevel(i))
- 3.0) BlockPsum(node.children(i), y(i), level(i))
Each PE(i): y(i) := y(i) - node.children(i)
- 4.0) Each PE(i): newFirstChild(i) := x(i) + y(i)
Each PE(i): lastChild(i) := node.firstChild(i) + node.oldChildren(i) - 1
RouteAndCopy(newFirstChild(i), firstSibling(i), lastChild(i), node.oldChildren(i) > 0)
Every PE(i): newAddress(i) := firstSibling(i) + alive'(i) - 1
- 5.0) RouteAndCopy(newAddress(i), node.parent(i), lastChild(i), node.oldChildren(i) > 0)
Each PE(i): node.firstChild := newFirstChild(i)
- 6.0) Route(node(i).node(i), newAddress(i), alive(i) = 1)

Figure 8.5 Procedure UpdateTree.

Algorithm UpdateTree:

Input: For all PE(i) registers alive(i) and node.newChildren(i) where alive(i) is set to 1, if the node at i is not to be deleted, or 0 otherwise and node.newChildren(i) equals number of new children to be created for node node(i) in this iteration.

Output: A new back-up tree of n nodes contained in the first n processors of our hypercube. Note that the new nodes created during this iteration will not be completely initialized until the expansion rule is invoked.

Method: To understand the algorithm, as given in Figure 8.5, consider the node p in Figure 8.6. The address (processor number) of the first child of p, after the update is completed, is the sum of two numbers x and y, defined as follows: x is the number of

nodes in the updated tree up to (and including) p 's level, while y is the number of children belonging to nodes to the left of p in p 's level. The sum of these two numbers indicates the position of p 's first child in the new back-up tree.

Procedure UpdateTree is composed of six steps. In Step 1, the number of children at each node is recalculated to account for the nodes that have been marked for deletion and the new nodes that are to be added. This operation is performed by first performing a partial sum over the number of alive nodes within each block of nodes that share the same parent. The result of the partial sum is then sent to each parent by its last child. Step 1 is completed by setting register `node.children(i)` of each node to the sum of its undeleted children and its new children to be created.

Figure 8.6 Calculating a node's new address in the back-up tree.

In Step 2, the x value for each node is computed. The x value for any node at level k is one plus the sum of the number of children of all nodes up to (and including) the last node in level $k - 1$. The x value for all nodes can therefore be calculated as follows. First, a partial sum on the number of children, `node.children(i)`, of every node is computed and

stored in register $x'(i)$; $x'(i)$ is then incremented by one to account for the root node. Then, the last node in each block of nodes sharing the same parent sends its x' value to its parent node.

In Step 3, the value of y for each node is calculated. This is performed by computing a partial sum on register $\text{node.children}(i)$ for all nodes in the same level. In order not to count the children of a node in its own $y(i)$ count, $\text{node.children}(i)$ must then be subtracted from the value provided by the partial sum operation.

In Step 4, the new address of the first child of each node is computed and stored in register $\text{newFirstChild}(i)$, using the $x(i)$ and $y(i)$ values previously calculated. This value is then broadcast to all the children of each node; since the back-up tree is stored on the hypercube in level order, all children of a node can calculate their new addresses by adding to the new address of their first sibling the number of other siblings that are before them.

At this point in the algorithm every node stores in register $\text{newAddress}(i)$ its address in the new back-up tree. In Step 5, all nodes with children broadcast their new address to their children, so that the children can update their parent pointers. Step 6 completes the UpdateTree procedure by routing every node to its new address. Since Steps 1-6 of the Update tree algorithm each take at most $O(\log n)$ time by Theorem 3.1, we can conclude that the UpdateTree algorithm also takes $O(\log n)$ time ♦

Lemma 8.1: *The UpdateTree operation on a current back-up tree of size n_c that creates a new back-up tree of size n_n can be completed in $O(\log \max\{n_c, n_n\})$ time.*

8.4 A BEST-FIRST BRANCH AND BOUND SEARCH EXAMPLE

To demonstrate how B&B algorithms can be defined in terms of our five general rules, consider the Solitaire Checkers Problem. The B&B search procedure we will sketch is an example of the best-first approach to searching. Note that, our aim here is to present an

example of a best-first B&B algorithm and its implementation on a fine-grained hypercube multiprocessor. No claim is made about the optimality of this solution to the n th order solitaire checkers problem.

8.4.1 n th Order Solitaire Checkers

n th order Solitaire Checkers is a one player game. The squares of a $2n$ by $2n$ board are given alternating light and dark colors and a set of $2n^2 - 2n$ pieces are placed on the dark squares at opposite ends of the board such that the two middle rows of the board are left empty (see Figure 8.7 for an example of the initial setup of a 5 by 5 game). A piece may move by diagonally by jumping over another piece, and landing on an unoccupied square. Both forward and backward jumps are allowed, with the jumped piece always being removed from the board. The aim of the game is to remove as many pieces as possible.

Figure 8.7 The initial setup of a n th Order Solitaire Checkers game ($n=5$).

The n th Order Solitaire Checkers Problem is computationally intractable. (In fact, the related n th order two player checkers problem has been shown to be P-space hard; see [Dew].) Therefore we can only hope to solve n th order Solitaire Checkers for small values of n . One possible approach is a B&B procedure like the one described in the next section.

8.4.2 A Best-First B&B Search Procedure for nth Order Solitaire Checkers Problem

The following five definitions describe a best-first solution to the nth order solitaire checker problem:

i) Cost(v, parent-cost)

Return(0)	If v is the root
Return(parent-cost + 1)	Otherwise

ii) Bound(v)

Return(1)	If all possible children of v have been explored and bounded
Return(0)	Otherwise

iii) Selection(v)

freep = # of free processors.

depth = maximum depth of any node in the current backup tree.

distance = $2n^2 - 2n - \text{depth}$ (Max distance to a solution)

Select the q = (freep - distance) nodes with highest cost.

Return(floor(freep/q))	For selected nodes v
------------------------	----------------------

Return(0)	Otherwise
-----------	-----------

iv) Expansion(v)

This is a heuristic function. For best results it should favor moves that result in pieces moving from back rows forward and from the edges of the board into the center of the board.

v) Termination

Return(True)	If distance = 0 ¹²
--------------	-------------------------------

Return(False)	Otherwise
---------------	-----------

¹² A less demanding definition might be to return true when distance $\leq n$

Note that the above definitions do not contain enough information for us to be able to calculate the time complexity of one pass through the resulting the B&B procedure. Clearly, the cost rule and bound rule can be calculated in $O(1)$, and global information required by the selection and termination rules can be calculated in $O(\log n)$. The time complexity of the selection and expansion rules will depend on how the current back-up tree is stored on the hypercube. If the current back-up tree is stored in level order, as described in the previous section, both of these rules can be implemented such that they require $O(\log n)$ time. In the case of the selection rule the best q nodes in a level-ordered-tree will be the q leaf nodes stored on the q processors with processor addresses. Therefore the selection rule can be implemented using a concentrate operation, followed by a Psum operation, followed by a route operation, in time $O(\log n)$. In the case of the expansion rule, each new set of children of some node v , can be created by an $O(\log n)$ time step, using a RouteAndCopy operation. Based on this analysis, and the assumption that the back-up tree is stored in level order, it is clear that one pass through this best-first B&B procedure takes $O(\log n)$ time.

8.5 CONCLUSION

In this chapter we have presented a highly efficient B&B algorithm for fine-grained hypercube multiprocessors. Our method uses a global storage allocation scheme where all processors collectively store all back-up paths such that each processor needs to store only a constant amount of information. At each iteration of the algorithm, all nodes of the current back-up tree may decide whether they need to create new children, be pruned, or remain unchanged. We have described an algorithm that, based on these decisions, updates the current back-up tree and distributes global information in $O(\log n)$ steps, where m is the current number of nodes. This method also provides a dynamic allocation mechanism that achieves optimal load balancing. This load balancing scheme is optimal in the sense that

even if very drastic changes in the current back-up tree occur, the performance of the load balancing mechanism remains constant.

It is difficult to tightly quantify the time complexity of a general branch and bound algorithm since it relies heavily on application specific details. What can be evaluated is the amount of time required to perform one expansion or contraction step. Given that Steps 1, 2, 3, 5 of algorithm UpdateTree (see Figure 8.5) take at most $O(\log n)$ time (as, is the case for the best-first B&B algorithm given in Section 8.4), and that Step 4 takes $O(\log n)$ (Lemma 8.1), we can conclude that the time complexity of one expansion or contraction step of the B&B algorithm as a whole is $O(\log n)$.

Theorem 8.1: *Parallel Branch and Bound on a hypercube multiprocessor of size N , in which the size of the back-up tree is n , $n \leq N$, can be implemented such that the expansion or contraction of n nodes takes $O(\log n)$ time.*

9 CONCLUSION

In this thesis we have shown that highly efficient algorithms based on parallel implementations of advanced data structures can be designed for fine-grained hypercube multiprocessors. We have introduced a new general technique for the design and implementation of a large class of data structures that support concurrent query access.

We have shown that a data structure represented as an ordered h -level graph, with n nodes and s sources, can be searched by m independent search queries (with maximum path length h), in parallel, on a hypercube multiprocessor of size $N = \max\{n, m\}$, in time $O(\min\{s \log N, \log^2 N\} + h \log N)$. Our algorithm allows an arbitrary number of queries to access the same node at the same time.

We have further extended this result by presenting a parallel version of fractional cascading [CG1] designed for fine-grained hypercube multiprocessors. A parallel version of this important data structuring technique allows us to address, in the parallel setting, many applications in computational geometry. We have shown that for a monotone catalog graph of size n (and bounded degree), m iterative search queries along paths of length at most p can be executed independently, in parallel, in time $O((p + \log N) \log N)$ on a hypercube multiprocessor of size $N = \max\{n, m\}$.

Our parallel version of fractional cascading is based on an enhanced version of the m -way search algorithm. These enhancements increase both the number of data structures that can be represented and the type of search processes that can be performed. Together the m -way search algorithm and parallel fractional cascading form a new technique for the

design of highly efficient algorithms based on parallel implementations of advanced data structures. Using this technique, we have demonstrated elegant yet highly efficient algorithms for a range of fundamental computational geometry problems. As Table 9.1 illustrates, the algorithms presented in this thesis improve upon the best previously known solutions to their respective problems by a factor of $O(\log n)$. In fact, due to the complex structure of these problems, several of them were previously unsolved for the hypercube multiprocessor, except by PRAM simulation.

Problems	Time complexity of best previous hypercube algorithm (by PRAM simulation)	Time complexity of hypercube algorithm presented in this thesis
Multi Binary Search	$O(\log^3 N)$	$O(\log^2 N)$
Next Element Search	$O(\log^3 N)$	$O(\log^2 N)$
Trapezoidal Map	$O(\log^3 N)$	$O(\log^2 N)$
Triangulation of a simple polygon	$O(\log^3 N)$	$O(\log^2 N)$
Stabbing simple polygonal paths	$O((k \log n/k + \log N) \log^2 N)$	$O((k \log n/k + \log N) \log N)$
Slanted Range Search*	$O((k + \log N) \log^2 N)$	$O((k + \log N) \log N)$

Table 9.1 Summary of results

One of the big open problems in the domain of hypercube algorithms is the optimal $O(\log N)$ time general simulation of the PRAM. Such an algorithm would be essentially equivalent to an $O(\log N)$ time sorting algorithm for the hypercube. Research on this problem has been ongoing since Batchier presented an $O(\log^2 N)$ result in 1968 [Bat]. The enhanced m -way search technique presented in this thesis is significant in that it can be considered as an optimal $O(\log N)$ time PRAM simulation for a large class of standard data structures.

* Presented in [DFR2]

9.1 DIRECTIONS FOR FUTURE RESEARCH

There are several directions in which one could attempt to further generalize the parallel data structuring techniques presented in this thesis. Currently, the most general class of graphs for which the searching techniques presented in this thesis are defined, is the class of monotone graphs. Recall that monotone graphs are planar, but not all planar graphs are monotone. Ideally, one would like to be able to use the class of planar graphs to represent data structure and still be able to efficiently search them in parallel.

Another possible area for further related work is to attempt to generalize the data structuring techniques presented in this thesis to a larger class of dynamic data structures. Currently, efficient algorithms based on parallel implementations of dynamic trees can be designed using the techniques presented in this thesis. It may be possible to dynamize other data structures in a similar way. Ideally one would like dynamic versions of every static monotone (or planar) data structure.

10 REFERENCES

- [AC] S.Anderson and M.C.Chen, "Parallel branch and bound algorithms on the hypercube," in *Proc. of Hypercube Multiprocessors 1987*, M.T.Heath, ed., SIAM Press, Philadelphia, PA, pp 309-317.
- [ACG] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading divide-and-conquer: a technique for designing parallel algorithms," Technical Report CSD-TR-665, Department of Computer Science, Purdue University, 1987.
- [ACGOY] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, "Parallel computational geometry," *Algorithmica* 3:3, 1988, pp 293-327.
- [AG] M.J. Atallah, M.T. Goodrich, "Efficient plane sweeping in parallel," in *Proc. ACM Symp. Computational Geometry*, 1986, pp 216-225.
- [Akl] S.G. Akl, "Parallel Sorting Algorithms," Academic Press, 1985.
- [AM] T.S.Abdelrahman and T.N.Mudge, "Parallel branch and bound algorithms on hypercube multiprocessors," in *Proc. of Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1492-1499.
- [Bat] K.E. Batchner, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conference*, 1968, pp 307-314.
- [BBN] BBN Laboratories Inc., "Butterfly Parallel Processor Overview".
- [BHRY] A. Bonopera, V. Ho, A. Rau-Chaplin, and D. Yeong, "Connection Machine Implementations of Hypercube operations and Data Structuring Techniques," Technical Note 90-1, Dept. of Computer Science, Carleton University.
- [BM] L. Boxer and R. Miller, "Dynamic computational geometry on meshes and hypercubes," in *Proc. Int. Conf. on Parallel Processing*, 1988, pp 323-330.
- [BW] J.L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers* 29:7, 1980, pp 571-576.
- [CG1] B. Chazelle, L.J. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica* 1:2, 1986, pp 133-162.
- [CG2] B. Chazelle, L.J. Guibas, "Fractional cascading: II. Applications," *Algorithmica* 1:2, 1986, pp 163-192.
- [Cha] B. Chazelle, "Filtering search: a new approach to query answering," *SIAM J. Comp.* 15:3, 1986, pp 703-724.
- [Cham] R.M. Chamberlain, "An alternative view of LU factorization with partial pivoting on a hypercube multiprocessor," in *Proc. Conf. on Hypercube Multiprocessors*, 1987, pp 569-575.
- [Col] R. Cole, "Parallel merge sort," in *Proc. IEEE Symp. on Foundations of Computer Science*, 1986, pp 511-516.

- [Dew] A.K. Dewdney, "The Armchair Universe," Macmillan of Canada, 1988.
- [DFR1] F. Dehne, A. Ferreira and A. Rau-Chaplin, "Parallel branch and bound on a hypercube multiprocessor," in *Proc. IEEE International Workshop on Tools for Artificial Intelligence*, Herndon, VA, 1989, pp 616-622.
- [DFR2] F. Dehne, A. Ferreira and A. Rau-Chaplin, "Parallel fractional cascading on a hypercube multiprocessor," to appear in *Proc. Allerton Conf. on Communication, Control and Computing*, Monticello, Ill., 1989.
- [DK] N. Dadoun and D.G. Kirkpatrick, "Parallel processing for efficient subdivision search," in *Proc. ACM Symp. on Computational Geometry*, 1987, pp 205-214.
- [DR1] F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry," to appear in *Proc. Int. Workshop on Graphtheoretic Concepts in Computer Science (WG'89)*, Aachen (W.-Germany), 1989.
- [DR2] F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry," to appear in the *Journal of Parallel and Distributed Computing*.
- [ECC] Encore Computer Corporation, "Multimax Technical Summary".
- [Fel] E.W.Felten, "Best-first branch and bound on a hypercube," in *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1500-1504.
- [GKLS] D.D. Gajski, D.J. Kuck, D.H. Lawrie and A.H. Sameh, "Cedar - A large Scale Multiprocessor," in *Proc. 1983 Int'l Conf. Parallel Processing*, Aug 1983, pp 524-529.
- [Goo] M.T. Goodrich, "Efficient parallel techniques for computational geometry," Ph.D. thesis, Department of Computer Science, Purdue University, 1987.
- [HD] S.R.Huang and L.S.Davis, "Parallel iterative A* search : an admissible distributed heuristic search algorithm," *International Joint Conference on Artificial Intelligence 89*.
- [HS] W.D. Hills and G.L. Steele, JR., "Data Parallel Algorithms," *Comm. ACM* 29:12, 1986, pp 1170-1183.
- [IP] O.H. Ibarra, T-C. Pong, et al., "Hypercube algorithms for some string comparison problems," in *Proc. Int. Conf. on Parallel Processing*, 1988, pp 190-193.
- [LS] M. Livingston, Q.F. Stout, "Embeddings in Hypercubes," in *Proc. Int. Conf. on Math. Modeling*, 1987.
- [LW] G.J.Li and B.W.Wah, "Coping with anomalies in parallel branch and bound algorithms," *IEEE Trans. on Comp.*, vol. c-35, no. 6, Jun 86, pp 568-573.
- [MC] A.Marsland and M.Campbell, "Parallel search of strongly ordered game trees," *Computing Surveys*, vol. 14, no. 4, Dec 82, pp 533-551.
- [Meh] K. Mehlhorn, "Data structures and algorithms 3: multi-dimensional searching and computational geometry," Springer Verlag, 1984.

- [MM] R.E. Miller, S.E. Miller, "Using hypercube multiprocessors to determine geometric properties of digitized pictures," in *Proc. Int. Conf. on Parallel Processing*, 1987, pp 638-640.
- [MS] R. Miller, Q.F. Stout, "Some graph and image processing algorithms for the hypercube," in *Proc. Conf. on Hypercube Multiprocessors*, 1987, pp 418-425.
- [New] M.Newborn, "Unsynchronized iteratively deepening parallel alpha-beta search," *IEEE Trans. on PAMI*, vol. 10, no. 5, Sep 88, pp 687-694.
- [Nil] Nils J. Nilsson, "Principles of Artificial Intelligence," Tioga Publishing Co, 1980.
- [NS] D. Nassimi, S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. on Computers* 30:2, 1981, pp 101-106.
- [PH] F.P. Preparata and S.J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Comm. ACM* 2:20, Feb. 1977, pp 87-93.
- [Pla] C.G. Plaxton, "Load Balancing, Selection and Sorting on the Hypercube," in *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures*, June 1989, pp 64-73.
- [PS] F.P. Preparata and M.I. Shamos, "Computational geometry - an introduction," Springer Verlag, 1985.
- [PW] R.P.Pargas and D.E.Wooster, "Branch and bound algorithms on a hypercube," in *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1514-1519.
- [Qui] M.J.Quinn, "Implementing best-first branch and bound algorithms on hypercubes multicomputers," in *Proc. Hypercube Multiprocessors 1987*, M.T.Heath, ed., SIAM Press, Philadelphia, PA, pp 318-326.
- [SGB] K.Swan, J.Gawkowski and B.Blake, "Process and workload migration for a parallel branch and bound algorithm on a hypercube multicomputer," in *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1520-1530.
- [SZ] S.R. Seidel, L.R. Ziegler, "Sorting on hypercubes," in *Proc. Conf. on Hypercube Multiprocessors*, 1987, pp 285-291.
- [TM] F.S.Tsung and M.H.Ma, "A dynamic load balancer for a parallel branch and bound algorithm," in *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1505-1513.
- [TMC] Thinking Machines Corporation, "Lisp Reference Manual," version 5.0, September 1988, pp 45-48.
- [TW] R.E. Tarjan and C.J. Van Wyk, "An $O(n \log \log n)$ time algorithm for triangulating a simple polygon," *SIAM Journal of Computing* 17, 1988, 143-178.
- [UII] J.D. Ullman, "Computational Aspects of VLSI," Computer Science Press, 1984.
- [UYII] H.Usui, M.Yamashita, M.Imai, T.Ibaraki, "Parallel searches of game trees," *Systems and Computers in Japan*, vol. 18, no. 8, 1987, pp 97-109.
- [Win] P.H. Winston, "Artificial Intelligence," Addison-Wesley, July 1984.

-
- [WLY] B.W.Wah, G.J.Li and C.F.Yu, "Multiprocessing of combinatorial search problems," *Computer*, Jun 85, pp 93-108.
- [Yap] C.-K. Yap, "Parallel triangulation of a polygon in two calls to the trapezoidal map," *Algorithmica* 3:2, 1988, pp 279-288.

END

1 9/0 2/9 1

FIN