1990

Fast Parallel Algorithms on a Class of Graph Structures With Applications in Relational Databases and Computer Networks.

Sridhar Radhakrishnan
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9112265

Fast parallel algorithms on a class of graph structures with applications in relational databases and computer networks

Radhakrishnan, Sridhar, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1990

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

Fast Parallel Algorithms On A Class Of Graph Structures With Applications In Relational Databases And Computer Networks

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Sridhar Radhakrishnan

B.Sc., Vivekananda College, University of Madras, India, 1983
B.S., University of South Alabama, Alabama, 1985
M.L.I.S., Louisiana State University, Louisiana, 1986
M.S., Louisiana State University, Louisiana, 1987

August, 1990

Acknowledgements

I would like to thank Professor S.S. Iyengar without whom this work would not have been a reality. His guidance at every step of my work was very valuable. My academic career at LSU was made possible my Dean Kathaleen Heim of the School of Library and Information Science. My sincere thanks to her. The help of Prof. Bert R. Boyce during my dual master's program will be greatly remembered. Prof. Donald H. Kraft will be remembered not only as my most favorite teacher, but, as a person who has a great sense of humor. Special thanks to Prof. Doris Carver and Prof. Bush Jones for accepting to be my committee members.

I am grateful to Prof. Leslie Jones for collaborating with me on several projects. I would like to extend my special thanks to Prof. S. Kundu who taught me the art of careful reading and technical writing. Thanks are also due to Prof. Zheng for his valiant efforts in drilling the concepts of computational geometry into my head. My sincere thanks to Mr. Farrell Jones of the CADGIS Research Lab, LSU for having me as a research assistant for three years. My sincere gratitude to Dr. David L. Feinstein, Dr. Niccollai, Dr. Hain, and Dr. Longenecker of the University of South Alabama.

Without my friends (I will name everyone of them!) I would not have enjoyed this very moment. They made everyday in my 5 years at LSU worthy of it. First, my deepest gratitude to Dr. Chandra (friend, BIL) who is my mentor. Sincere thanks to Dr. Laks who was patient in listening to my vague solutions to research problems. I am grateful to Mohan (Dr!) for all his help during my difficult times at LSU. Thanks

to Daryl Thomas for throwing quarters at me for coffee.

I would like to thank my dearest friends in the following way. Dai KK (Krishnakumar) ne high-level da! Gooja (Rajanarayanan) you will always be remembered. Hedju! (Vinayak Hegde) Yenu! Thanks for listening to all my philosophical talks. Phatak! I will certainly remember you on the 25th of March. Also, thanks to Junk! (Venky), Vilva, Rajen, and Sanjoy. Sankara marenthanu nenaichiya! Thanksda Gunda!

I owe my entire stay in the U.S to my uncle (Dr. Ek) and aunt. Sowmya (fiancee) thanks a ton for infusing fresh blood during the final lap of my Ph.D. I would like to dedicate my thesis to my parents (S.R. Radhakrishnan and Dr. Kamala Radhakrishnan).

Table of Contents

Abstract

The quest for efficient parallel algorithms for graph related problems necessitates not only fast computational schemes but also requires insights into their inherent structures that lend themselves to elegant problem solving methods. Towards this objective efficient parallel algorithms on a class of hypergraphs called *acyclic hypergraphs* and *directed hypergraphs* are developed in this thesis. Acyclic hypergraphs are precisely *chordal graphs* and its subclasses, and they have applications in relational databases and computer networks. In this thesis, firstly, we present efficient parallel algorithms for the following problems on graphs.

- determining whether a graph is strongly chordal, ptolemaic, or a block graph. If the graph is strongly chordal, determine the strongly perfect vertex elimination ordering.
- determining the minimal set of edges needed to make an arbitrary graph strongly chordal, ptolemaic, or a block graph.
- determining the minimum cardinality dominating set, connected dominating set, total dominating set, and the domatic number of a strongly chordal graph.

Secondly, we show that the query implication problem ($Q_1 \rightarrow Q_2$) on two queries, which is to determine whether the data retrieved by query Q_1 is always a subset of the data retrieved by query Q_2, is not even in NP and in fact complete in Π_2^P. We present several 'fine-grain' analysis of the query implication problem and show that the query implication can be solved in polynomial time given *chordal queries*.

Thirdly, we develop efficient parallel algorithms for manipulating directed hypergraphs H such as finding a directed path in H, closure of H, and minimum equivalent hypergraph of H. We show that finding a directed path in a directed

hypergraph is inherently sequential. For directed hypergraphs with fixed degree and diameter we present *NC* algorithms for manipulations. Directed hypergraphs are representation schemes for functional dependencies in relational databases.

Finally, we also present an efficient parallel algorithm for multi-dimensional range search. We show that a set of points in a rectangular parallelepiped can be obtained in $O(logn)$ time with only $2.\log^2 n - 10.logn + 14$ processors on a EREW-PRAM. A non-trivial implementation technique on the hypercube parallel architecture is also presented. Our method can be easily generalized to the case of d-dimensional range search.

Chapter 1

INTRODUCTION

1. INTRODUCTION

There has been a tremendous interest in algorithmic graph theory to develop efficient algorithms for various graph problems. This is to a large extent due to the increase in the application of graph theory to problems of practical interest. Of the various graph structures which have received wide attention, planar graphs, perfect graphs, bipartite graphs, trees, chordal graphs, and partial-k-chordal graphs occupy a special place. Studies on such restricted classes of graphs are well-motivated from the following points of view:

1. Solutions to problems on restricted graphs oftentimes are easier to obtain compared to arbitrary graphs.

2. Studies on restricted graph classes shed light on solutions to problems for arbitrary graphs.

3. Sometimes in real-life situations we may encounter only graphs with special structures.

4. In many situations, restricted graph classes are studied for their intrinsic mathematical interest.

Traditionally, researchers in algorithmic studies on graphs have focussed on the development of deterministic-sequential algorithms. In recent years, deterministic parallel algorithms for several important computational problems have been developed. This is supposed to be in preparation for a revolutionary switch from

sequential to parallel computing. In fact, one can expect parallel computing to dominate research initiatives for a few decades to come.

Recently, there has been a great spurt of research activity towards developing deterministic sequential and parallel algorithms for a class of graphs called *perfect graphs* [47]. Among the class of perfect graphs the *chordal graph* and its subclasses occupy the chief position. This is due to the fact that many polynomial-time algorithms can be designed systematically for the class of chordal graphs and its subclasses.

The theory of *hypergraphs* [15], has been extensively used in computer science as a mathematical model to represent concepts and structures from different domains: rewrite systems, databases, logic programming, etc. In all cases hypergraphs generalize the concept of graph in the sense that they consists of a set of nodes and a set of (hyper)edges defined over the nodes. A different model that has been used in several applications is the *directed hypergraph* [8]. Directed hypergraphs are a generalization of directed graphs in which an arc can have more than two nodes. In the next two subsections an overview of the chordal graph and its subclasses and the directed hypergraph will be presented.

2. THE STRUCTURE OF CHORDAL GRAPHS AND THEIR APPLICATIONS

Informally, a simple, loopless, and undirected graph is *chordal* if every cycle of length of at least four contains a chord (an edge connecting two vertices that are not consecutive in the cycle). A chordal graph is also called as a *triangulated graph.*

Chordal graphs have important graphs forming their subclass and these include permutation graph, strongly chordal graph, ptolemaic graph, block graph, interval graph, path graph, threshold graph, k-trees etc. [47]. Figure 1.1 gives a chordal hierarchy.

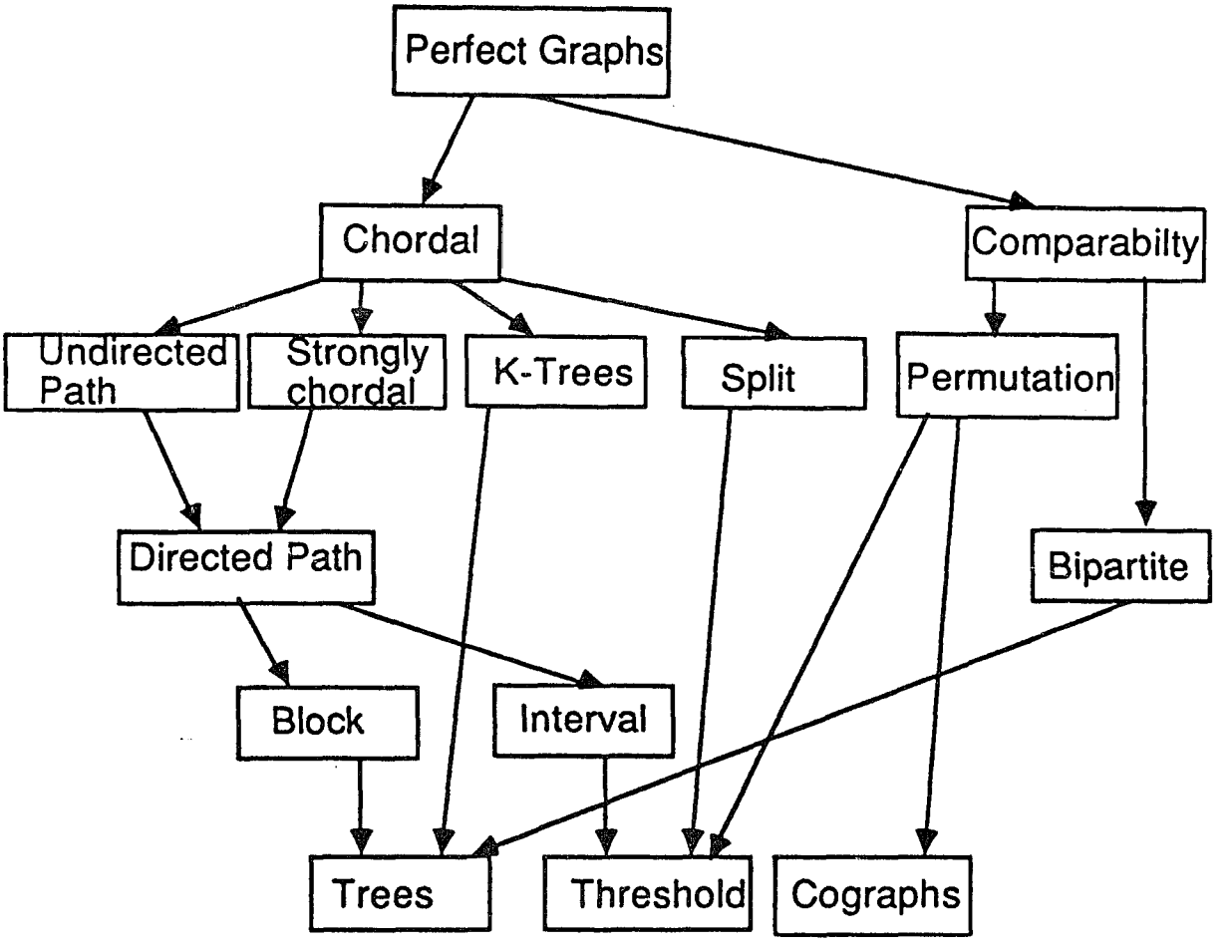

Figure 1.1: A hierarchy of chordal graphs

An important property of chordal graphs is that it exhibits an ordering (<) on its vertices $\{v_1, v_2, ..., v_n\}$ called the *perfect elimination ordering (PEO)* which satisfies the condition that node v_i and nodes v_j adjacent to v_i with $i < j$ form a complete subgraph (not necessarily maximal) [47]. A chordal graph $G = (V, E)$ can be transformed into a hypergraph H with vertices V and hyperedges $S = \{S_1, S_2, ..., S_q\}$,

where each S_i is a maximal clique (completely connected subgraph) of G. The hypergraph H thus obtained from a chordal graph is called an α-acyclic hypergraph [11]. Figure 1.2 shows a chordal graph, its PEO, and the α-acyclic hypergraph corresponding to it.

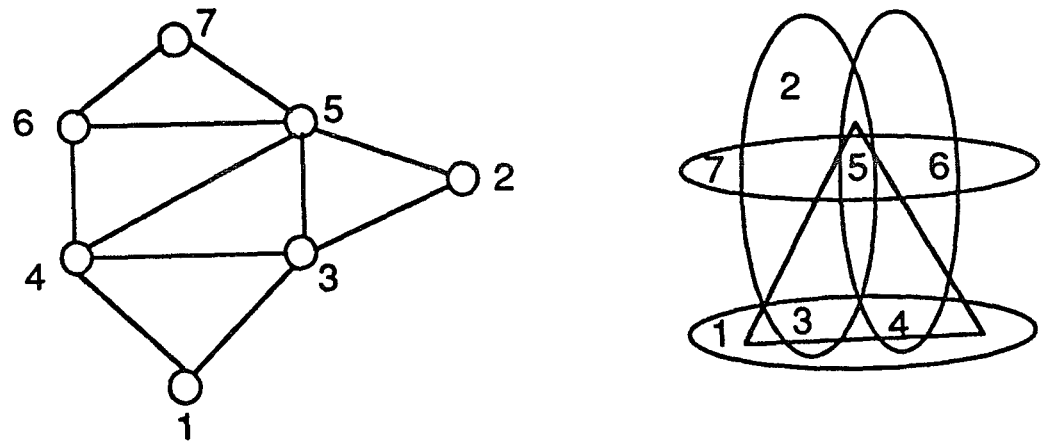

Figure 1.2: A chordal graph with PEO numbering of the vertices and the α-cyclic hypergraph corresponding to it.

Chordal graphs are an important class of graphs since polynomial time sequential algorithms are available on them, otherwise known to be NP-complete on general graphs. Here is a list of NP-complete problems on general graphs known to be solved in polynomial time on chordal graphs [44].

1. Maximum Independent Set
2. K-colorability
3. Clique Cover

There also exist problems which are NP-complete on chordal graphs and known to be solvable in polynomial time for graphs in its subclasses. Here is a list of problems which are NP-complete on chordal graphs.

1. Minimum cardinality connected, total, and independent dominating set
2. Steiner tree
3. Minimum Fill-in i.e., minimum number of edges needed to make an arbitrary graph chordal.

Chordal graphs and their subclasses have a close relationship with the theory of acyclic hypergraphs or acyclic database schemes [3, 36]. It was shown by D'Atri and Moscarini [3] that a graph G is chordal, strongly chordal, ptolemaic, or block, if and only if the corresponding hypergraph H formed from the maximal cliques of G is acyclic, β-acyclic, γ-acyclic, or Berge-acyclic, respectively. Chordal graphs and its subclasses arise in several important applications, which include the following.

(a). Relational database design and query evaluation [5, 11, 36].

(b). Computing solutions of sparse system of linear equations [4, 37].

(c). Probabilistic expert systems [64].

(d). Reliability of communication networks in the presence of constrained line and site failures [39, 40, 96].

(e). VLSI design layout [88].

An important sub-class of a chordal graph is the strongly chordal graph intro-

duced by Farber [37]. Strongly chordal graphs are chordal graphs in which every even cycle of length at least 6 has a strong chord (i.e., an edge joining two vertices which are an odd distance apart in the cycle). The edge-vertex incidence matrix of a strongly chordal graph is "totally balanced." Totally balanced matrices are studied in proving certain min-max theorems in graphs theory [4]. An example of a strongly chordal graph is given in Figure 1.3

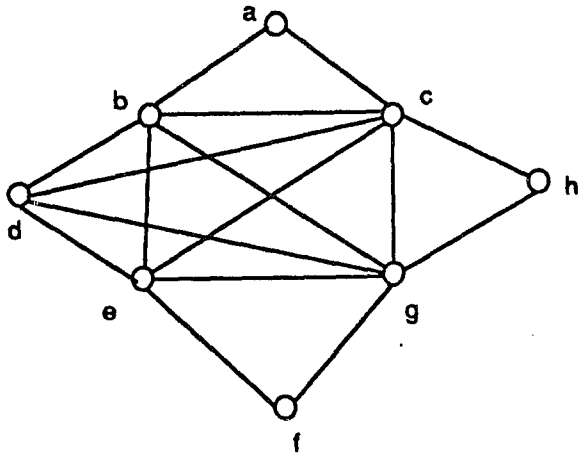

Figure 1.3: A strongly chordal graph.

Recently, there has been a growing interest in developing algorithms for a class of graphs called K-trees and partial-K-trees. Numerous NP-complete problems can be solved in polynomial-time (linear in most cases) when the input graph is a partial-K-tree [6, 17]. A K-tree is K-chordal graph in which every clique is of size at most $(K + 1)$. A partial-K-tree is a partial-K-chordal graph which is a subgraph of a K-chordal graph. A fast NC algorithm for recognizing partial-K-trees was given by Chandrasekharan and Hedetniemi [18].

3. AN OVERVIEW OF DIRECTED HYPERGRAPHS AND ITS APPLICATIONS

In this section, an informal discussion about directed hypergraphs is presented. Formal definitions are presented in Chapter 6. In the case of directed graphs, an arc (a, b) consists of the *source node a* and the *destination (sink) node b*. Directed hypergraphs are a generalization of directed graphs where the source nodes can be a set of nodes, called the *compound node*. There is a directed path from compound node A to compound node B if and only if there are directed paths from A to each of the component nodes forming B. Figure 1.4 gives an example of a directed hypergraph.

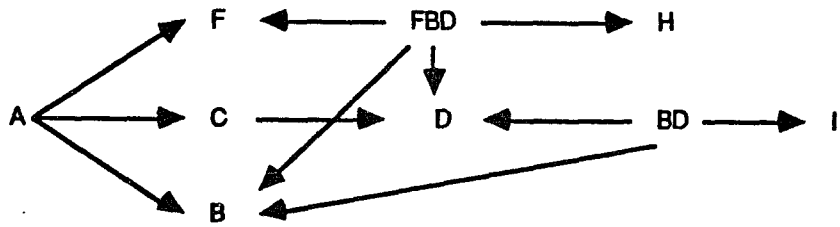

Figure 1.4: A directed hypergraph

As observed previously, directed hypergraphs may be often applied to provide a formal model of concepts in computer science. A few examples are presented now. It should be noted that all these are informal discussions.

An important application of directed graphs is in the area of database theory. Given a set of attributes U, a set of functional dependencies is a relation over $P(U) \times U$, where $P(U)$ is a set of attributes from U. A functional dependency from a set of attributes X to a single attribute i means that, given the values of all attributes in X, the value of attribute i is uniquely determined. Clearly a directed hypergraph may provide an immediate representation for such a relationship. In this context, a typical problem is to determine a set of functional dependencies which is equivalent to a given one but which is minimal with respect to some parameter (number of dependency rules in the set, length of the total representation, etc.). Sequential algorithms for the manipulation of functional dependencies are presented by Ausiello et al. [8].

Another field of application in which it is meaningful to look for equivalent (or strongly equivalent) representations of the same hypergraph is problem solving. Directed hypergraphs may be used in problem solving as an alternative to *and-or graphs*, for describing the relationship existing among a given problem P and the set of problems whose solution is required to solve problem P [46]. In this case, for example, given a hypergraph, it would be meaningful to look for an equivalent representation where the number of independent problems which may be solved in parallel is maximal.

Finally, another interesting application of directed hypergraphs arises in the representation and manipulation of the *Horn Formula*. Among various classes of logical formulae, Horn Formulae are particularly interesting in view of the fact that in Knowledge Based Systems is often represented by means of *if* ... *then* ... clausal rules.

Also in this case the use of directed hypergraphs is quite natural.

In the case of propositional calculus, for example given a set of propositional variables V and the truth values T and F, a Horn formula is a relation over $P(V') \times V''$, where $V' = V \cup \{T\}$ and $V'' = V \cup \{F\}$. A typical problem that we are interested in solving in this case is the implication, that, is the existence of a path, from a given set of variables X to a single q.

A particularly interesting case is when, in the process of building a Horn formula which represents our knowledge of a given domain by progressively adding new clauses we want to check on-line the existence of a path from T to F because such a path would imply the unsatisfiability of the whole formula [31].

4. AN OVERVIEW OF PARALLEL ALGORITHMS

In recent years, parallel computation has come to influence all areas of computer science and related disciplines. This is mostly because of possible limits to hardware speed and software efficiency in the realm of sequential computing. Even before the existence of real parallel machines, computer scientists were developing a theoretical framework to develop the model of parallel of computations based on processor capability, memory accessibility, and the pattern of interconnection among processors. Though there is no consensus on a model of parallel computation, many studies have focussed on the *parallel-random-access-machine* (PRAM) model (see Karp and Ramachandran [56]). The PRAM model is a parallel analog of the sequential RAM. It consists of several independent sequential processors, each with its own private

memory, communicating with each other through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location. PRAMs can be classified according to restrictions on the global memory access. In *EREW* (Exclusive Read Exclusive Write) PRAMs simultaneous access to any memory location for both reading and writing is forbidden. In a *CREW* (Concurrent Read Exclusive Write) PRAM simultaneous reads are allowed but not simultaneous writes. A *CRCW* (Concurrent Read Concurrent Write) PRAM allows both simultaneous reads and writes. In the case of a concurrent write we assume that an arbitrary processor succeeds, though other assumptions are possible (see Moitra and Iyengar [70]). These models are increasingly powerful in that order. It is known that the CREW and the CRCW PRAM models can be simulated by an EREW-PRAM model in $O(logP)$ time with $O(P)$ extra processors or with no extra processors in $O(\log^2 P)$ time [33]. It was shown in [94], that all PRAM models with P processors can be simulated by an ultracomputer (bounded-degree network of processors with no global memory) in $O(logP\ (loglogP)^2)$ time per step and with no extra processors. Having described the models of parallel computation we will be using in our work, we next turn our attention to the class of algorithms we will consider. In our discussion below we only consider sequential algorithms having polynomial-time complexity.

The most natural and a very practical idea of parallelism to make use of a fixed number of processors say, P (independent of the size of the input say, n). In this case, it is not hard to see that the speedup of a parallel algorithm (over the sequential one) for a problem will depend on the input size and when the input size is increased, the

number of processors has to be dependent on the input size. The agreement in this case is to make use of only a reasonable amount of hardware i.e., a polynomial number of processors. The polynomiality of resources (time and processors) has been accepted as a reasonable demand.

Having justified the use of a polynomial number of processors let us consider the issue of *time*. The worst-case time-complexity of a parallel algorithm is a measure of the maximum time taken by any of the processors over all inputs. Using a polynomial number of processors we may hope to do as good as getting a constant-time parallel algorithm. However, creating a polynomial number of processors requires nonconstant time in a reasonable model of parallel computation. For example, if we want to create say $O(n^r)$ processors, we need $O(\log_2 n^r)$ time assuming a binary-tree configuration for processor creation. Hence reasonably good speedup can be said to have been achieved when the parallel time-complexity is a *polylog* (polynomial in the logarithm) of n. Note that in this case the speedup is exponential! Without much further ado, let us say that the class of *NC algorithms* (due to Nicholas Pippenger [76]) consists of algorithms which run in $O(\log^r n)$ time and make use of a polynomial number of processors. NC algorithms have been found for several important problems in areas like algebra, graph theory, computational geometry etc. For a good survey on NC algorithms see Karp and Ramachandran [56].

In contrast to NC is the class of problems that cannot be speedup very much using a polynomial number of processors. A class that seems to capture this notion is the class of P-complete problems. The class P is the set of all problems that can be

solved in polynomial time on a deterministic Turing machine. The question as to whether or not $P = NC$ is open. It is known that, for example, if any P-complete problem is in NC, then $P = NC$. The following problem called the circuit value problem (CVP) is known to be P-complete [75]. Circuit value problem asks for the output of a boolean circuit given certain number of inputs which are either true or false.

With this brief introduction to parallel algorithms we list the contributions of the thesis in the next section.

5. MAIN FOCUS, CONTRIBUTIONS, AND OUTLINE OF THE THESIS

The central theme of this thesis is to develop fast parallel algorithms for a class of graph structures with applications in relational databases and computer networks. The mathematics and the algorithmic framework presented in this thesis are very interesting and deep and offers the potential for far reaching applications as large scale parallel computers come into their own. In the following paragraphs the focus of the thesis together with the results obtained are listed.

1. Parallel algorithms for the recognition of strongly chordal, ptolemaic, and block graphs are developed. Given a graph G with n vertices and m edges we present parallel recognition algorithms to determine if G is strongly chordal, ptolemaic, or block. We obtain the following worst-case time and processor bounds for our recognition algorithms which run on a CRCW-PRAM model.

(1) $O(\log^2 n)$ time using $O((n+m)^{3/2}/\log^2 n)$ processors for recognizing a strongly chordal graph;

(2) $O(\log^2 n)$ time using $O(n+m)$ processors for recognizing a ptolemaic graph;

(3) $O(\log^2 n)$ time using $O(n+m)$ processors for recognizing a block graph.

Our recognition algorithm for strongly chordal graphs has a better processor bound than the one proposed by Dahlhaus and Karpinski [28], who show that strongly chordal graph can be recognized in $O(\log^2 n)$ time using $O(n^4)$ processors. The above results are presented in Chapter 2.

2. A parallel algorithm to obtain the *strongly perfect elimination ordering* (SPEO) of the vertices of a strongly chordal graph is developed. Various domination problems which have applications in computer networks are solved in linear-time sequentially, given the SPEO ordering of the strongly chordal graphs [21]. SPEO ordering is also used in Gaussian elimination and other computations on sparse matrices [4]. The parallel algorithm for the construction of SPEO works in $O(\log^2 n)$ time with $O((n+m)^{3/2}/\log^2 n) + M(n))$ processors on a CRCW-PRAM, where $M(n)$ is the cost of multiplying two $n \times n$ boolean matrices. Our SPEO construction algorithm is a significant improvement over the algorithm of Dahlhaus and Karpinski [28], who present an algorithm which runs in $O(\log^2 n)$ time and uses $O(n^8)$ processors. The algorithm for obtaining the SPEO is presented in Section 2.6.

3. It was shown earlier, by Dahlhaus and Karpinski [26] that maximum matching [49] in chordal graphs is as hard as bipartite graph matching. This implies that maximum

matching in chordal graphs is in Random NC. An $O(\log^2 n)$ time algorithm which uses $O(n^8)$ processors on a strongly chordal graph was presented by [26]. We present an $O(\log^2 n)$ time algorithm which uses $O((n+m)^{3/2}/\log^2 n + M(n))$ processors on a CRCW-PRAM for determining the maximum matching in strongly chordal graphs. This result is presented in Chapter 2.

4. From discussions in the earlier subsections it can be clearly seen that chordal graphs have advantages over arbitrary graphs. Dahlhaus and Karpinski [27] present an $O(\log^3 n)$ time parallel algorithm which uses $O(nm)$ processors to convert an arbitrary graph into a chordal graph by adding a minimal set of edges. We use the algorithm of Dahlhaus and Karpinski to construct strongly chordal, ptolemaic, and block graphs from arbitrary graphs. This result is significant in the sense that we are able to build acyclic databases given cyclic ones, and is discussed in Chapter 3.

5. We present fast parallel and linear-time sequential algorithms for domination problems on strongly chordal graphs which have applications in computer networks. A set of vertices $D \subseteq V$ is a *1-dominating set* for a graph $G = (V, E)$ if every vertex in $V - D$ is adjacent to a vertex in D. The domination set D is *connected* if the graph induced by vertices in D is connected, *independent* if the vertices in D are independent, and *total* if every vertex in V is adjacent to a vertex in D. The *domination problem* or *1-domination problem* is to determine the minimum cardinality set D. Gerard Chang [20] gave a linear time sequential algorithm for the k-domination problems on a strongly chordal graph given a *simple vertex elimination ordering* and without taking powers of the graph. Farber [38] presented a linear time sequential algorithm for

the minimum weight domination and minimum weight independent domination given *strongly perfect elimination ordering* of the vertices of the strongly chordal graph.

Given a strongly chordal graph G with n vertices and m edges, we present sequential algorithms with $O(n+m)$ time complexity and fast parallel algorithms with $O(\log^2 n)$ time complexity using $O(n+m)$ processors on a CRCW-PRAM model for the following problems:

(1) Dominating set problem

(2) Domatic number problem, i.e., determine the maximum integer K and disjoint vertex sets $V_1, V_2, ..., V_K$ such that each V_i is a dominating set.

(3) Connected domination problem

(4) Total domination problem

We know of no parallel algorithms for domination problems on strongly chordal graphs. Domination and other related problems are solved in Chapter 4.

6. The query implication problem $(Q_1 \rightarrow Q_2)$ on two queries Q_1 and Q_2 is to determine whether the data retrieved by the query Q_1 is always a subset of the data retrieved by Q_2. The query implication problem has applications in the areas of computational geometry, distributed databases, and others. We study the general implication problem in which all six comparison operators: $=, \neq, <, >, \leq, \geq$, as well as conjunctions and disjunctions, are allowed. It is shown here that the general implication problem is not even in NP and in fact complete in $\Pi^p{}_2$. In the simple case where the

comparison operator is only '=', we show that the implication problem is *NP*-Complete. We define a class of queries called 'acyclic queries' and show the existence of polynomial-time algorithms for the implication problems which are shown to be *NP*-Complete.

We use the above results, to estimate the time-complexity of determining whether two update transactions consisting of *insert* and *delete* operations are equivalent. Conjunctive queries arise in the area of query optimization in relational data bases [68]. We show that the testing implication of two conjunctive queries with inequalities is $\Pi^P{}_2$-Complete. The query implication problem is discussed in Chapter 5.

7. Given a set of functional dependencies Σ and a single dependency σ, we show that the algorithm to test whether Σ implies σ is log-space complete in P. The functional dependencies Σ are represented as a directed hypergraph H_Σ [8]. We first present a parallel algorithm which solves the above implication problem using P processors on an EREW-PRAM in $O(e/P + n.logP)$ time and on a CRCW-PRAM in $O(e/P + n)$ time, where e and n are the number of arcs and nodes of the graph H_Σ. For graphs H_Σ with fixed degree and diameter, we show that the closure $H_\Sigma{}^+$ can be computed in NC. We present NC algorithms to obtain a non-redundant and an LR-Minimum cover for the set of functional dependencies Σ. All our algorithms on a n-node directed hypergraph with fixed degree and diameter can be implemented to run in $O(\log^2 n)$ time with $M(n)$ processors on a CREW-PRAM model, where $M(n)$ is the cost of multiplying two binary matrices. The algorithms are efficient based on the *transitive*

closure bottleneck phenomenon [56], that is, any improvement in the time and processor complexity of the transitive closure algorithm will result in an improvement by the same amount for the algorithms presented here. Algorithms for directed hypergraph or functional dependency manipulations are presented in Chapter 6.

8. We present a parallel algorithm to obtain a set of points in a rectangular parallelepiped (range-search) in $O(logn)$ time, with only $(2.\log^2 n - 10.logn + 14)$, on ann EREW-PRAM, where processors are allowed to communicate through messages. We also present a non-trivial implementation technique on the hypercube parallel architecture with which the above time and processor bound can be achieved without any communication overhead. A parallel algorithm for range searching is developed here using the concept of distributed data structures. We use the range tree proposed by Bentley as our data structure to be distributed. Our algorithm can easily be generalized for the case of a d-dimensional range search. Range search has important applications in the areas of databases and computational geometry. The range searching problem is discussed in chapter 7.

We conclude this thesis in Chapter 8. and mention some open problems.

Chapter Two

PARALLEL ALGORITHMS FOR RECOGNIZING A CLASS OF CHORDAL GRAPHS

1. INTRODUCTION

The central theme of this chapter is to provide characterizations for strongly chordal, ptolemaic, and block graphs in terms of the intersection graph of its maximal cliques. Using the correspondence between the maximal cliques of the above graphs and acyclic hypergraphs we develop characterization theorems for them. We first show that the intersection graph of a strongly chordal graph is chordal. Now, since a block graph is a ptolemaic graph which is a strongly chordal graph, the chordality property of the intersection graph holds for these graphs also. It turns out that this necessary condition is crucial for the purposes of recognition and computation on strongly chordal graphs. In section 4, we present an $O(\log^2 n)$ time parallel algorithm which uses $O(n+m)$ processors on a CRCW-PRAM to construct the intersection graph. The above time and processor complexities are achieved by the use of an important combinatorial lemma on chordal graphs by Fulkerson and Gross [42] who show that the sum of the sizes of edge labels of the intersection graph of a chordal graph is at most $O(n+m)$.

There has been a number of parallel chordal graph recognition algorithms [19, 34, 52, 73]. The most efficient recognition algorithm in terms of the number of processors was developed by Philip Klein [60]. In [60] a variety of problems including PEO (*Perfect Elimination Ordering*), finding maximal cliques, and coloring was

solved in NC^2 using a linear number of processors (linear in the number of vertices and edges of the graph). Dahlhaus and Karpinski [28] were the first to obtain an NC^2 algorithm to recognize and determine the strong vertex elimination ordering of strongly chordal graphs. The algorithm of Dahlhaus and Karpinski for computing the SPEO of a strongly chordal graph runs in $O(\log^2 n)$ time and uses $O(n^8)$ processors. We present an algorithm for computing the SPEO and it runs in $O(\log^2 n)$ time and uses $O((n+m)^{3/2}/\log^2 n) + M(n))$ processors on a CRCW-PRAM. This algorithm is presented in Section 5. The material contained in this chapter is a completely revised and expanded version of Radhakrishnan and Iyengar [82, 79].

2. PRELIMINARIES -- NOTATIONS AND DEFINITIONS

We adopt terminologies and state theorems from [3] to give the relationships between the chordality of a graph G and acyclicity of the hypergraph H. Duke [32] gives a good survey of the various cycles in hypergraphs. We assume G to be a class of simple, loopless, undirected graphs and H to be undirected, reduced, and conformal hypergraphs. The number of vertices is n and the number of edges is m.

Definition 2.2.1 [Graph Chordality and Elimination Orderings]

A *chordal* graph [47] is a graph in which every cycle with at least four distinct nodes has a chord. A vertex v is *simplical* if the graph induced by v and its neighbors is a clique. An ordering of the vertices $v_1, v_2, ..., v_n$ with v_i simplical in the graph induced by $\{v_i, v_{i+1}, ..., v_n\}$ for all i is called a *perfect elimination scheme.* A graph is chordal iff it has a perfect elimination scheme (or ordering) (PEO) [47].

A *strongly chordal* graph [37] is a chordal graph that in every even cycle with at least six nodes contains a strong chord (i.e., a chord joining two nodes with an odd distance in the cycle). A perfect elimination ordering $<$ of the vertices $v_1, ..., v_n$ is a strong perfect elimination ordering (SPEO) if and only if for any x, y, x', y', s.t. (x, y), $(x\ y')$, $(y, x') \in E$ and $x < x', y < y'$, we have $(x', y') \in E$. Also, Farber showed that a graph G is strongly chordal if and only if G has an SPEO.

A *ptolemaic* graph G [53] is a strongly chordal graph and each 5-cycle of G has at least three diagonals, or each cycle of G of length greater than or equal to 5 has a pair of diagonals which cross one another.

A *block* graph is a ptolemaic graph in which each biconnected component is a complete subgraph [50].

Figure 2.1 and Figure 2.2 give examples of chordal and strongly chordal graph with PEO and SPEO numbering, respectively.

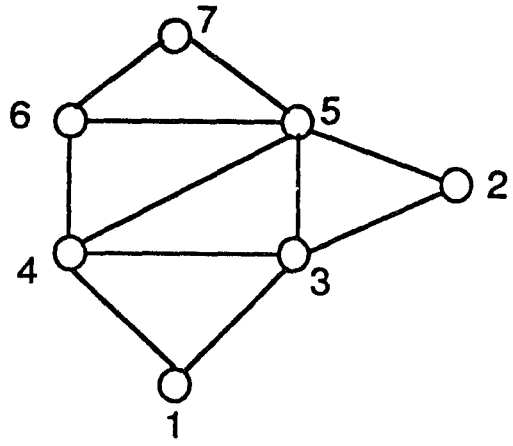

Figure 2.1: A chordal graph with PEO numbering of the vertices

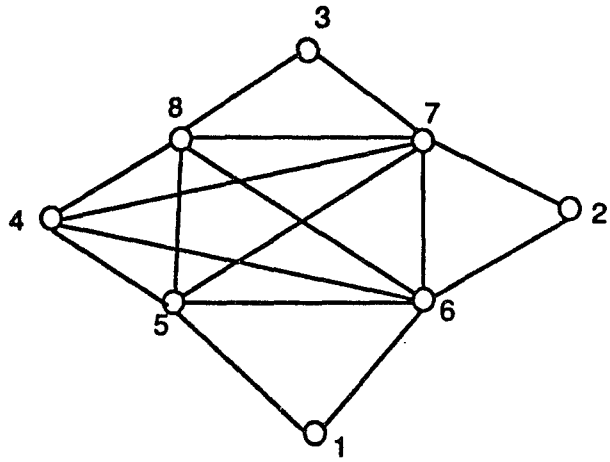

Figure 2.2: A strongly chordal graph with SPEO numbering of the vertices

Definition 2.2.2 [Intersection Graph]: Let H be the maximal cliques $\{S_1, S_2, ..., S_r\}$, $1 \leq r \leq n$, of a chordal graph G. We can treat H as a hypergraph by treating each maximal clique in H as a hyperedge. The *intersection graph* $I(H)$ of H is a graph containing edges $S_i \in H$ as nodes and edges (S_i, S_j) labeled $l(S_i \cap S_j)$, if $S_i \cap S_j \neq \emptyset$.■

Figure 2.3 gives an example of the intersection graph of the graph in Figure 2.2.

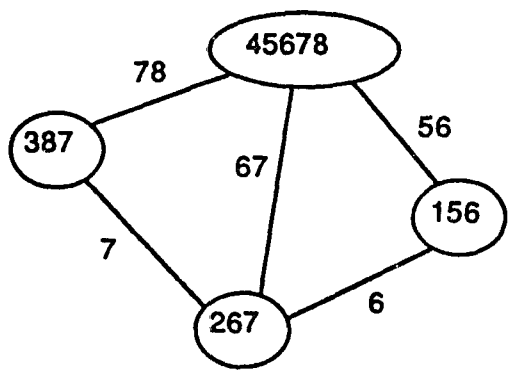

Figure 2.3: The intersection graph of the maximal cliques of the graph in Figure 2.2.

Definition 2.2.3 [Hypergraph Acyclicity]:

The following definitions for hypergraph acyclicity are taken from [36].

A hypergraph H is *Berge-acyclic* if it does not contain the following sequence $(S_1, x_1, S_2, x_2, ..., S_m, x_m, S_{m+1})$ satisfying the following conditions:

(i) $x_1, ..., x_m$ are distinct nodes of H;

(ii) $S_1, ..., S_m$ are distinct edges of H, and $S_{m+1} = S_1$;

(iii) $m \geq 2$, that is, there are at least 2 edges involved; and

(iv) x_i is in S_i and S_{i+1} $(1 \leq i \leq m)$.

Figure 2.4 gives an example of a block graph together with its intersection graph formed by the maximal cliques (hyperedges) which is Berge-acyclic.

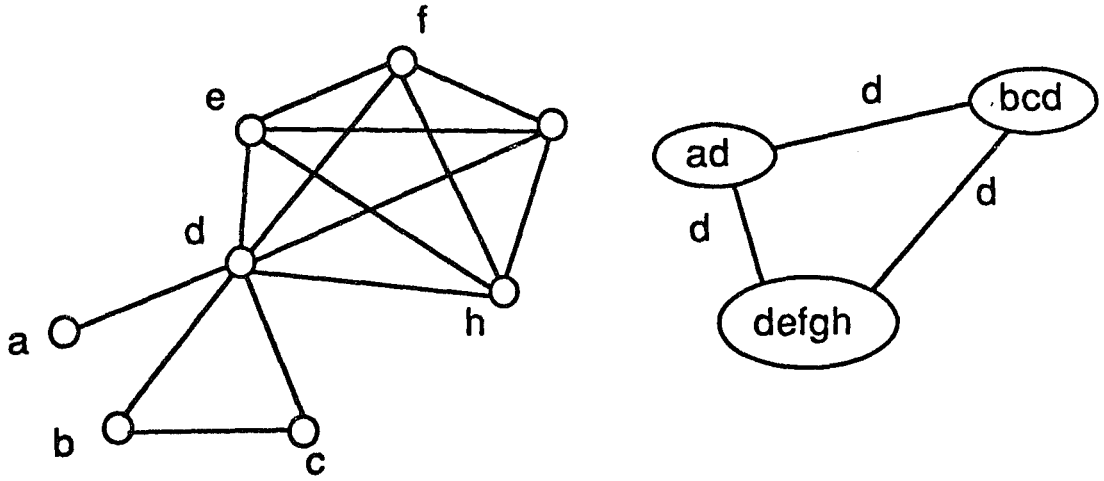

Figure 2.4: A block graph together with its intersection graph of the maximal cliques.

A hypergraph H is *γ-acyclic* if it does not contain the following sequence $(S_1, x_1, S_2, x_2, ..., S_m, x_m, S_{m+1})$ satisfying the following conditions:

(i) $x_1, ..., x_m$ are distinct nodes of H;

(ii) $S_1, ..., S_m$ are distinct edges of H, and $S_{m+1} = S_1$;

(iii) $m \geq 3$, that is, there are at least 3 edges involved;

(iv) x_i is in S_i and S_{i+1} $(1 \leq i \leq m)$; and

(v) If $1 \leq i \leq m$, then x_i is in no S_j except S_i and S_{i+1}.

Figure 2.5 gives an example of a ptolemaic graph together with its intersection graph formed by the maximal cliques (hyperedges) which is γ-acyclic.

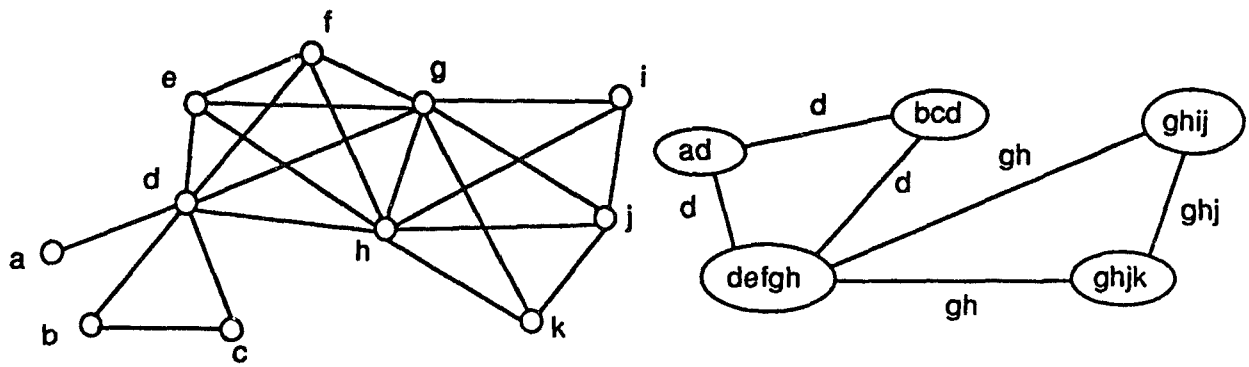

Figure 2.5: A ptolemaic graph together with its intersection graph of the maximal cliques.

A hypergraph H is β-*acyclic* if it does not contain the following sequence $(S_1, x_1, S_2, x_2, ..., S_m, x_m, S_{m+1})$ satisfying the following conditions:

(i) $x_1, ..., x_m$ are distinct nodes of H;

(ii) $S_1, ..., S_m$ are distinct edges of H, and $S_{m+1} = S_1$;

(iii) $m \geq 3$, that is, there are at least 3 edges involved; and

(iv) x_i is in S_i and S_{i+1} $(1 \leq i \leq m)$ and in no other S_j.■

We will now state some theorems which gives the relationship between chordality of graphs and acyclicity and shows how the various forms of acyclicities are related. The following theorem was proved in [3].

Theorem 2.2.1: A graph G is chordal, strongly chordal, ptolemaic, or block if and only if the hypergraph H is α-, β-, γ, or *Berge*-acyclic, respectively.■

Fagin [36] proved the following hierarchy of acyclicities.

Theorem 2.2.2: The following implication Berge-acyclicity ⇒ γ-acyclicity ⇒ β-acyclicity ⇒ α-acyclicity and none of the reverse implication holds.■

The following result of Klein [60] is a very useful one and used often in this thesis.

Theorem 2.2.3: Recognition of a chordal graph, determining the PEO, maximal cliques, maximum independent set, and coloring of a chordal graph can all be done in $O(\log^2 n)$ time using $O(n+m)$ processors on a CRCW-PRAM.■

3. CHARACTERIZATIONS OF ACYCLICITIES

In this section we provide certain characterizations of acyclicities based on the intersection graph of the maximal cliques of the graph G. These characterizations are used in deriving efficient recognition algorithms in section 4. First, we begin by

proving the following necessary condition.

Lemma 2.3.1: The *intersection* graph $I(H)$ for a set of maximal cliques H of the graph G is chordal, if G is block, ptolemaic, or strongly chordal.

Proof: When graph G is strongly chordal we know from Theorem 2.2.1 that H is β-acyclic. We will show by contradiction that $I(H)$ is chordal. Let H be β-acyclic and $I(H)$ not chordal. Consider the sequence $(S_1, x_1, S_2, x_2, S_3, x_3, S_4, x_4, S_1)$ in $I(H)$. The nodes x_1, x_2, x_3, and x_4 are all distinct (otherwise there would be a chord connecting opposite vertices). The sequence is β-cycle, which means that H contains a β-cycle, which is a contradiction. Therefore, $I(H)$ is chordal when G is strongly chordal. >From Theorem 2.2.2, we see that $I(H)$ is chordal when G is ptolemaic or a block.■

We will now, present a characterization for the recognition of Berge-acyclicity.

Theorem 2.3.2: Let $I(H)$ be the intersection graph with labels $|S_i \cap S_j| < 2$, for all nodes S_i, S_j in $I(H)$. The hypergraph H is Berge-acyclic if and only if $I(H)$ is chordal and every triangle (S_i, S_j, S_k) in $I(H)$ satisfies the condition $S_i \cap S_j = r$, $S_j \cap S_k = r$, $S_k \cap S_i = r$ and $|r| = 1$.

Proof: (*IF*) Let us assume that H is Berge-acyclic. It is clear from Lemma 2.3.1 that $I(H)$ is chordal. Consider the triangle (S_i, S_j, S_k) in $I(H)$. We have $S_i \cap S_j = r$, and $|r| = 1$, for otherwise we have a Berge-cycle (S_i, A, S_j, r, S_i) when $S_i \cap S_j = Ar$ (Definition 2.2.3). Thus, $|r_i| = 1, i = 1, 2, 3$. If $S_i \cap S_j = r_1, S_j \cap S_k = r_2, S_k \cap S_i = r_3$, then, we clearly have a Berge-cycle. The case where $S_i \cap S_j = r_1, S_j \cap S_k = r_1$,

and $S_k \cap S_i = r_2$ does not exists. Thus, every triangle in $I(H)$ satisfies the condition of the theorem.

(*ONLY-IF*) Let us assume that $I(H)$ is chordal and every triangle (S_i, S_j, S_k) satisfies the condition of the theorem. It is easy to see that the triangle (S_i, S_j, S_k) is Berge-acyclic. We have to show that any cycle of length $k > 3$ is not a Berge-cycle. Consider a sequence $(S_1, r_1, S_2, r_2, S_3, r_3, S_4, r_4, S_1)$ in $I(H)$. Now, $|r_i| = 1$, $i = 1, 2, 3, 4$, $r_1 = r_2$, and $r_3 = r_4$ from the assumptions about each triangle in $I(H)$ and chordality of $I(H)$. Therefore, the above sequence is not a Berge-cycle. In fact, is it easy to see that S_i, $i = 1, 2, 3, 4$ in $I(H)$ forms a complete sub-graph due to the following. >From the assumptions: $r_1 = r_2$ and let $S_1 \cap S_3 = r_1$ (chordality assumption). Now, $r_1 = r_2 = r_3 = r_4$, which implies, $S_1 \cap S_2 \cap S_3 \cap S_4 \neq \emptyset = r_1$.■

Corollary 2.3.3: Let $I(H)$ be a complete graph. H is Berge-acyclic if and only if for all S_i and S_j in $I(H)$, $S_i \cap S_j = r$ and $|r| = 1$.

Proof: See the proof of Theorem 2.3.2.■

>From the above characterization it can be seen that, testing coro. 2.3.3 on every maximal clique of $I(H)$ would be a sufficient test for the recognition of Berge-acyclicity. We develop the following characterization for the recognition of γ-acyclicity.

Theorem 2.3.4: The hypergraph H is γ-acyclic if and only if $I(H)$ is chordal and every triangle (S_i, S_j, S_k) in $I(H)$ satisfies the condition $S_p \cap S_q = r_1$, $S_q \cap S_r = r_1$, $S_r \cap S_p = r_2$ for some $p \neq q \neq r$ in $[i, j, k]$.

Proof: (*IF*) Let us assume that H is γ-acyclic. It is clear from Lemma 2.3.1 that $I(H)$ is chordal. Consider the triangle (S_i, S_j, S_k) in $I(H)$. If $S_i \cap S_j = r_1$, $S_j \cap S_k = r_2$, and $S_k \cap S_i = r_3$, then we have a γ-cycle $(S_i, r_1, S_j, r_2, S_k, r_3, S_i)$, a contradiction to the assumption. For the case, $S_i \cap S_j = r_1$, $S_j \cap S_k = r_1$, $S_k \cap S_i = r_1$, and $|r_1| = 1$, H is Berge-acyclic (Theorem 2.3.2), which implies H is γ-acyclic (Theorem 2.2.2). Clearly, when $|r_1| > 1$, there is no γ-cycle. For the case, $S_i \cap S_j = r_1$, $S_j \cap S_k = r_1$, and $S_k \cap S_i = r_2$ the graph is still γ-acyclic. Hence, every triangle satisfies the condition of the Theorem.

(*ONLY-IF*) Let us assume that $I(H)$ is chordal and every triangle (S_i, S_j, S_k) satisfies the condition of the theorem. We can easily show that there is no γ-cycle of length $k > 3$ in $I(H)$. Let $(S_1, r_1, S_2, r_2, S_3, r_3, S_4, r_4, S_1)$ be a sequence in $I(H)$. The sequence is not a γ-cycle, since $r_1 \cap r_2 \neq \emptyset$ and $r_2 \cap r_3 \neq \emptyset$ from assumptions about chordality of $I(H)$ and condition on each triangle. In fact, it can be shown that any cycle of length k in $I(H)$ is part of a complete sub-graph on k vertices. ■

We will now derive some properties of β-acyclic hypergraphs.

Lemma 2.3.5: If H is β-acyclic, then, every triangle (S_i, S_j, S_k) in $I(H)$ satisfies the condition $S_p \cap S_q \subseteq S_p \cap S_r$ and $S_p \cap S_q \subseteq S_q \cap S_r$ for some $p \neq q \neq r$ in $[i, j, k]$.

Proof: Let $S_q \cap S_q = r_1$, $S_q \cap S_r = r_2$, $S_r \cap S_q = r_3$, and H be β-acyclic. If $r_1 \cap r_2 \cap r_3 = \emptyset$, then, we have a β-cycle, a contradiction to the assumption. If $r_1 \cap r_2 \cap r_3 = x$ and $x \subset r_i$, $i = 1, 2, 3$, then we have a β-cycle by Definition 2.4, a contradiction. The case where, $(r_1 = r_2) \cap r_3 = \emptyset$ does not exist, and either $r_3 = r_1$ or the contain-

ment stated in the Lemma holds.■

Definition 2.3.1 [36]: A triangle (S_1, S_2, S_3) *begins a* β-*cycle* in $I(H)$, if it satisfies the following condition:

Let $X = S_1 \cap S_2 \cap S_3$, and let $S'_i = S_i - X$, for $i = 1, 2, 3$. Let $T = \{E \in I(H)$: $(E = S_1)$ or $(E = S_3)$ or $(X \subset E$ and $E \cap S'_2 = \emptyset)\}$. Let $T' = \{E - X : E \in T\}$. Now, (S_1, S_2, S_3) begins a β-cycle in $I(H)$ if and only if S'_1 and S'_3 are in the same component of T'. It should be clear that the operation to test whether a triangle begins a β-cycle in $I(H)$ is examining a complete subgraph of $I(H)$.■

Lemma 2.3.6: Let $I(H)$ be a *complete* graph satisfying the condition of Lemma 2.3.5. The graph H is β-acyclic if and only if an arbitrary chosen triangle (S_i, S_j, S_k) in $I(H)$ does not begin a β-cycle.

Proof: (*IF*) Let us assume H to be β-acyclic. Clearly, every triangle in $I(H)$ satisfies the condition of Lemma 2.3.5. Also, an earlier result of Fagin [36] tells us that no triangle in $I(H)$ begins a β-cycle.

(*ONLY-IF*) Let $I(H)$ satisfy the conditions of Lemma 2.3.5 and an arbitrarily triangle in $I(H)$ does not begin a β-cycle. Now, we have to show that no other triangle in $I(H)$ begins a β-cycle and hence H is β-acyclic. Let (S_1, S_2, S_3, S_4) be the vertices of the graph $I(H)$. Let $S_1 \cap S_2 \cap S_3 \cap S_4 = x \neq \emptyset$. Let our arbitrary triangle be (S_1, S_2, S_3). The operation in Definition 2.3.1 to test if the triangle begins a β-cycle completely disconnects the graph $I(H)$. Let (S_2, S_3, S_4) be another triangle and we will show that it cannot begin a β-cycle. Note that, $S_1 \cap S_2 \cap S_3 = x_1 \supseteq x$, $S_2 \cap S_3 \cap S_4$

$= x_2 \supseteq x$, and $x_1 \cap x_2 \neq \emptyset$. Hence, the operation in Definition 2.3.1, when testing the triangle (S_2, S_3, S_4), would also completely disconnect the graph, justifying that it does not begin a β-cycle. Thus, no triangle in $I(H)$ begins a β-cycle, hence H is β-acyclic. ■

Theorem 2.3.7: Let $I(H)$ be the intersection graph and let $I^*(H) = \{I^1(H), I^2(H), ..., I^k(H)\}$, $1 \leq k \leq n$, where $I^j(H)$ is a maximal clique of $I(H)$. Now, H is β-acyclic if and only if each $I^j(H)$ satisfies the condition of Lemma 2.3.6.

Proof: The proof follows from Lemma 2.3.6.■

4. ALGORITHMS AND COMPLEXITY

We present efficient recognition algorithms based on the characterizations developed for various acyclicities in section 3. We now state a combinatorial lemma.

Lemma 2.4.1 (see [60]) : Let S_v be the set of all maximal cliques of a chordal graph containing vertex v, $\Sigma_v \ |S_v| = O(n+m)$.■

>From Lemma 2.4.1 we can see that the sum of the sizes of all edge labels of the intersection graph $I(H)$ for a set of maximal cliques H of a chordal graph is $O(n+m)$. We now present a method to construct the intersection graph efficiently.

Given a Chordal graph G with n vertices and m edges its maximal cliques H can be determined in $O(\log^2 n)$ time with $O(n+m)$ processors on a CRCW-PRAM [60]. Let the maximal cliques be represented as a vertex-clique incidence lists M.

Let $M(i)$ correspond to the incidence list of vertex v_i. The intersection graph $I(H)$ can be constructed as follows. Allocate processors $P^r_{i,1}, P^r_{i,1}, ..., P^r_{i,|M(r)|^2}$ to the incidence list of vertex v_r. The processor $P^r_{i,j}$ stores the following information: (a). the vertex v_r, (b). the clique number S_i, (c). the adjacent clique number $S_j \neq S_i$. The total number of processors is $O(n+m)$, since the sum of the sizes of all edge labels is $O(n+m)$. Arrange all the processors sorted first by the field (b) and then by the field (c). Now, from the sorted order the edges and its labels from each clique S_i can be easily determined. The sorting can be done in $O(logn)$ time with $O(n+m)$ processors using the sorting algorithm of Cole [24]. Thus, the intersection graph $I(H)$ can be constructed in $O(logn)$ time with $O(n+m)$ processors.

Theorem 2.4.2: Given a clique-vertex incidence list M for a set of maximal cliques H of G the intersection graph $I(H)$ with labeled edges can be determined in $O(logn)$ time using $O(n+m)$ processors on a CRCW-PRAM model.■

The following Lemma by Fisher [41] establishes an upper bound on the number of triangles of graph G with n vertices and m edges. This Lemma would be useful in deriving the complexity estimates for the recognition algorithms.

Lemma 2.4.3: The upper bound on the number of triangles of a graph with n vertices and m edges is $O(m^{3/2})$.■

We will now present methods to recognize various hypergraph acyclicities. >From Lemma 2.3.1, for a hypergraph to be Berge-, γ-, or β-acyclic the intersection

graph of the hyperedges must be chordal. We will assume that as a preprocessing step, the intersection graph $I(H)$ has been tested for chordality, the nodes are assigned PEO numbers, and the maximal cliques of $I(H)$ have been determined in $O(\log^2 n)$ time with $O(n+m)$ processors on a CRCW-PRAM using the algorithm of Klein [60]. We have the following theorem for Berge-acyclicity recognition complexity.

Theorem 2.4.4: Let H be the maximal cliques of a chordal graph G with n vertices and m edges and let $I(H)$ be chordal. Berge-acyclicity of H can be recognized in time $O(\log^2 n)$ with $O(n+m)$ processors on a CRCW-PRAM model.

Proof: Check the condition of Coro. 2.3.3 on each maximal clique of $I(H)$. Since the total number of edges is at most $O(n+m)$ (Lemma 2.4.1), checking all the maximal cliques can be done in constant time with $O(n+m)$ processors on a CRCW-PRAM.■

For recognizing γ-acyclicity we process each clique $I^j(H)$ of $I(H)$ as follows. Choose the edge label l_j in $I^j(H)$ whose size is minimum. This can be done for all maximal cliques in time $O(\log n)$ with $O(n+m)$ processors. Remove edges from $I^j(H)$ whose label is l_j (*removal operation*). This operation is clearly takes $O(1)$ time with $O(n+m)$ processors from Lemma 2.4.1. We will now show that if the degree of each node in $I^j(H)$ is less than two after the removal operation, then $I^j(H)$ satisfies Theorem 2.3.4 and hence the hypergraph formed using the maximal cliques H is γ-acyclic. Let us assume that $I^j(H)$ satisfies Theorem 2.3.4 and after the removal of edges which has the label l_j, let S_j be the node with degree two, i.e., with edges (S_j, S_p) and (S_j, S_q). Now, the edge labels of (S_j, S_p) and (S_j, S_q) should be the same for otherwise we have a triangle (S_j, S_p, S_q) which does not satisfy Theorem

2.3.4, a contradiction. The degree of each node in $I^j(H)$ after the removal operation can be determined in $O(1)$ time with $O(n+m)$ processors, thus, γ-acyclicity can be recognized in $O(\log^2 n)$ time with $O(n+m)$ processors on a CRCW-PRAM.

Theorem 2.4.5: Let H be the maximal cliques of a chordal graph G with n vertices and m edges and let $I(H)$ be chordal. γ-acyclicity can be recognized in $O(\log^2 n)$ with $O(n+m)$ processors on a CRCW-PRAM model. ■

For recognizing β-acyclicities using the characterization in Lemma 2.3.5, we have to necessarily process each triangle. Since the intersection graph has at most $O(n+m)$ edges the number of triangles is $O((n+m)^{3/2})$ from Lemma 2.4.3. With $P = \text{Min}(|S_i \cap S_j|, |S_j \cap S_k|, |S_k \cap S_i|)$ the triangle (S_i, S_j, S_k) can be checked to see if it satisfies the condition of Lemma 2.3.5 in $O(1)$ time by indexing into an array storing edge labels. In fact, any set of t triangles each of which contains at least one edge whose label size is less than or equal to P, can be processed in $O(1)$ time using $P \cdot t$ processors. Now, $P \cdot t \leq O((n+m)^{3/2})$ from Lemma 2.4.1 and Lemma 2.4.3. Now it can be easily seen that with $O((n+m)^{3/2})$ processors all triangles in $I(H)$ can be processed. The condition of Lemma 2.3.6 is checked as follows. For each maximal clique $I^j(H)$ in $I(H)$, arbitrarily choose a triangle (S_i, S_j, S_k). Assume edge (S_i, S_j) is contained (label is contained) in edges (S_j, S_k) and (S_k, S_i). Now for all edges (S_t, S_j) which contains (S_i, S_j), remove node S_t from $I^j(H)$. All edges (S_r, S_s) whose edge labels are contained in (S_i, S_j) are removed from $I^j(H)$. Now, check to see if S_i and S_k are connected in $I^j(H)$ using the connected components algorithm [87]. If S_j and S_k are connected then, triangle (S_i, S_j, S_k) begins a β-cycle and H is

β-cyclic. If the sum of the sizes of all edge labels in $I^j(H)$ is T, then all the above operations can be done in $O(logT)$ time using $O(T)$ processors. Hence, all the maximal cliques of $I(H)$ can be processed in $O(logn)$ time using $O(n+m)$ processors.

Theorem 2.4.6: Let H be the maximal cliques of the chordal graph G with n vertices and m edges and let $I(H)$ be chordal. β-acyclicity can be recognized in $O(\log^2 n)$ time with $O((n+m)^{3/2}/\log^2 n)$ processors on a CRCW-PRAM.

Proof: Follows from the discussion above.■

5. ALGORITHM FOR COMPUTING STRONGLY PERFECT VERTEX ELIMINATION SCHEMES

We use the characterization of Dahlhaus and Karpinski [26] for strongly perfect elimination schemes in terms of maximal cliques of the graph and develop a fast parallel algorithm for computing such a scheme. Using the same characterization Dahlhaus and Karpinski developed an NC^2 algorithm using $O(n^8)$ processors. Our use of the intersection graph of the maximal cliques of a strongly chordal graph reduces the processor bound by a great extent. We will now state some definitions and present the characterization for strongly perfect elimination schemes.

Definition 2.5.1 [26]: Choose a clique m and say $x <_m' y$ if and only if there is a *chain* from x to m via y, such that the sequence $x_0 = x, x_1, ..., x_i = y, x_{i+1}, ..., x_k = m$ with $x_{i-1} \cap x_i$ and $x_i \cap x_{i+1}$ are incomparable by inclusion.■

Definition 2.5.2 [26]: For two cliques x and y we say $x <_m{}^L y$ if and only if there exists a clique z such that $\emptyset \neq x \cap y \subset y \cap z$.■

Lemma 2.5.1 [26]: Let $<_m$ be the transitive closure of $<_m{}' \cup <_m{}^L$. The ordering satisfying $<_m$ is a strongly perfect elimination ordering of the maximal cliques of the strongly chordal graph.■

We now present an algorithm for computing the strongly perfect elimination ordering of the vertices of a strongly chordal graph.

Algorithm SPEO;
Input: A strongly chordal graph G with n-vertices and m-edges.
Output: An ordering of vertices of G satisfying strongly perfect elimination ordering.

Begin
1. Determine the maximal cliques H of G.
2. Construct the intersection graph $I(H)$.
3. Obtain a PEO numbering of the chordal intersection graph $I(H)$.
4. Determine the maximal cliques R of $I(H)$.
5. Order the maximal cliques in R as $R_1, ..., R_k$ such that R_i contains a vertex whose PEO number is less than all vertices in R_j, for $i < j \leq k$.
6. **For** each clique R_i of R **Do-in-parallel**
   Begin
7. **For** each triangle (x, y, z) in R_i **Do-in-parallel**
   Begin
8. Determine the 'containment edge', i.e., (x, y) such that $x \cap y \subseteq y \cap z$, and $x \cap y \subseteq x \cap z$.
9. Add arcs (x, z) and (y, z) in the directed graph $G(R_i)$.
   End;
10. Compute strong components of $G(R_i)$, arbitrarily order members of each strong components and starting with a component with indegree zero construct a directed path containing all vertices in $G(R_i)$.
   End;
11. Merge all the paths computed in step 10 after deleting vertices H_l in R_j if is in some R_i with $i < j$. Call this $<_m$, the ordering of cliques of G satisfying Lemma 2.5.1.
12. Remove vertices v from maximal cliques H_j of G if it is in some H_i with $i < j$ in the ordering $<_m$.

13. Order vertices of G in each clique H_i based on the PEO numbers assigned to them (lower to higher).

End.

We will now show that the above algorithm correctly computes the SPEO on vertices of a graph G which is strongly chordal. >From Lemma 2.3.1 it is clear that the graph $I(H)$ of a strongly chordal graph is chordal. In Step 5 the ordering $R_1, ..., R_k$ of the maximal cliques defines the $<_m'$ ordering as follows. We will assume that R_i is the union of all vertices in the maximal cliques of G contained in R_i, since R_i is a maximal clique of $I(H)$. Let $K_{i-1} = R_{i-1} - \{R_{i-1} \cap R_i\}$, $K_{i+1} = R_{i+1} - \{R_{i+1} \cap R_i\}$, $K_i = R_i - \{K_{i-1} \cup K_{i+1}\}$. For $l_{i-1} \in K_{i-1}$, $l_i \in K_i$, $l_{i+1} \in K_{i+1}$, clearly, $l_{i-1} \cap l_i$ and $l_i \cap l_{i+1}$ are incomparable by inclusion. Hence, using the ordering of the R_i's of $I(H)$ the $<_m'$ ordering of the maximal cliques of the graph G is determined. Now, we are to order the vertices in each R_i satisfying the $<_m^L$ ordering. Note that since G is strongly chordal, every triangle in $I(H)$ contains a 'containment edge'. For a given triangle (x, y, z) the ordering done in Steps 7-9 clearly satisfies the $<_m^L$ ordering. Steps 10-12 performs the transitive closure of the orderings $<_m'$ and $<_m^L$.

The implementation of the algorithm SPEO is done as follows. We will first state the following remark without proof.

REMARK 2.5.2: The following problem can be solved in $O(\log P)$ time using P processors on CRCW-PRAM. Let M be an array of size P containing 0's and 1's. At the end of the execution of the algorithm an array element M_i contains a 1 if and only

if M_i contained a 1 initially and all $M_j = 0$ for $1 \leq j < i \leq P$.

Steps 1-4 can be implemented in $O(\log^2 n)$ time using $O(n+m)$ processors on a CRCW-PRAM. Step 5 is determined by finding the minimum vertex number in each R_i and sorting the R_i's based on the minimum vertex numbers. The operation can be done in $O(\log n)$ time using $O(n+m)$ processors. For sorting the parallel sorting algorithm of Cole [24] can be used.

>From Lemma 2.3.7 for a graph with n-vertices and m-edges there are at most $O((n+m)^{3/2})$ triangles. Using an argument similar to the one given for the recognition complexity in Section 3, Steps 6-10 can be implemented in $O(\log^2 n)$ time with $O((n+m)^{3/2}/\log^2 n)$ processors. Step 10 can be implemented in $O(\log n)$ time using $M(n)$ processors [56]. Steps 11-12 can be implemented in $O(\log n)$ time with $O(n+m)$ processors from REMARK 2.5.2. Step 13 can be done in $O(\log n)$ time with $O(n)$ processors using integer sorting algorithm of Cole [24]. We now state the following theorem.

Theorem 2.5.3: The strongly perfect elimination ordering of the vertices of a strongly chordal graph G with n-vertices and m-edges can be determined in $O(\log^2 n)$ time with $O((n+m)^{3/2}/\log^2 n + M(n))$ processors on a CRCW-PRAM.■

6. MAXIMUM MATCHING IN STRONGLY CHORDAL GRAPHS

In this section we will present a result for finding the maximum matching in strongly chordal graphs. A *matching* is a subset of edges in which no two edges are

adjacent. A *maximal matching* is a matching to which no edge in the graph can be added. A maximal matching with largest number of edges is called the *maximum matching*. A matching is a *perfect matching* is all vertices are covered by the edges in the matching. Determining a maximum matching in general graphs is in randomized NC [57]. It was shown by Dahlhaus and Karpinski [26] that maximum matching in chordal graphs is as hard as matching in bi-partite graphs which is as hard as general graphs. Rabin and Vazirani [78] and Mulmuley et. al. [71] have shown that is a graph has a unique perfect matching then the matching can be computed in $O(\log^2 n)$ time. It is known that maximum matching is NC-reducible to perfect matching, hence maximum matching can be solved in $O(\log^2 n)$ time for strongly chordal graphs if it is shown to have a unique prefect matching. It was shown by Dahlhaus and Karpinski [26] that there is a unique perfect matching for strongly chordal graphs.

The actual algorithm for finding a matching involves determining edges in the following manner. Let $<$ be the strongly perfect elimination ordering of the vertices of the strongly chordal graph. For every pair of edges (u_1, u_2), (v_1, v_2), with $(u_1, v_1) \in E$ (the edge set) and $u_1 < v_2$ and $v_1 < u_2$ take (u_1, v_1) and (u_2, v_2) into the matching. The proof of the above method is presented in [26]. Since the computing of strongly perfect elimination ordering is necessary for computing the maximum matching we state the main result.

Theorem 2.6.1: The maximum matching of a strongly chordal graph G with n vertices and m edges can be computed in $O(\log^2 n)$ time using $O((n+m)^{3/2}/\log^2 n + M(n))$ processors on a CRCW-PRAM.

Proof: The proof follows from Theorem 2.5.3 and above discussion.■

7. CONCLUSION

The recognition results obtained in this chapter are summarized in the following table.

Graph G (n vertices, m edges)	Hypergraph H, the maximal cliques of G	Previous recognition complexity		Our recognition complexity	
		Time	Processor	Time	Processor
Chordal	α-acyclicity	$O(\log^2 n)$ [21]	$O(n+m)$ [21]	-	-
Strongly Chordal	β-acyclicity	$O(\log^2 n)$ [8]	$O(n^4)$ [8]	$O(\log^2 n)$	$O((n+m)^{3/2}/\log^2 n)$
Ptolemaic	γ-acyclicity	-	-	$O(\log^2 n)$	$O(n+m)$
Block	Berge-acyclicity	-	-	$O(\log^2 n)$	$O(n+m)$

Table 1. - Time and processor complexity for recognition of the above graphs.

Chapter Three

PARALLEL ALGORITHMS FOR MINIMAL CONSTRUCTION OF A CLASS OF CHORDAL GRAPHS

1. INTRODUCTION

This chapter presents parallel algorithms for the construction of strongly chordal (β-acyclic hypergraph), ptolemaic (γ-acyclic hypergraph), and block (Berge-acyclic hypergraph) graphs given an arbitrary graph by adding a minimal set of edges. The parallel algorithm of Dahlhaus and Karpinski [27] determines the minimal set of edges needed to construct a chordal graph from an arbitrary n-vertex m-edge graph in $O(\log^3 n)$ time with $O(nm)$ processors on a CRCW-PRAM. Construction of a chordal graph from an arbitrary graph is also referred to as determining the *minimal elimination ordering* or *minimal fill-in.* It is well known that minimum fill-in is NP-complete [92]. In this chapter we outline a method to obtain a *minimal strong elimination ordering* (MSEO) given an arbitrary graph.

Determining the MSEO has several advantages. Clearly, problems which are NP-complete (see Chapter 1) on chordal graphs can now be solved in polynomial time on the minimally constructed strongly chordal graph. Also, determining the MSEO is equivalent to converting an arbitrary (0, 1)-matrix into a *totally balanced* matrix by

using minimal fill-in's [4]. In relational database theory it was shown that β-acyclic schemes satisfied desirable properties which the α-acyclic schemes did not [11]. So determining the MSEO can be thought of as designing β-acyclic relational databases from cyclic ones by adding a minimal set of attributes in each schema. A new elimination ordering of vertices called a *doubly perfect elimination ordering (DPEO)* for a ptolemaic graph or γ-acyclic hypergraph is defined. Using DPEO we present a parallel algorithm to find the minimal set of edges needed to make an arbitrary graph a ptolemaic graph.

2. MINIMAL CONSTRUCTION OF STRONGLY PERFECT ELIMINATION ORDERING (MSEO)

For the sake of completeness we present the following definitions again.

Definition 3.2.1 (*Perfect Elimination Ordering*): A vertex v is *simplical* if the graph induced by v and its neighbors is a clique. An ordering of the vertices $v_1, v_2, ..., v_n$ with v_i simplical in the graph induced by $\{v_i, v_{i+1}, ..., v_n\}$ for all i is called a *perfect elimination scheme or ordering*(PEO).■

Definition 3.2.2 (*Strongly Perfect Elimination Ordering*): A perfect elimination ordering (<) of the vertices $v_1, ..., v_n$ is a *strongly perfect elimination ordering* (SPEO) if and only if for any x, y, x', y', such that $(x, y), (x, y'), (x', y) \in E$ and $x < x', y < y'$

we have $(x', y') \in E$. The edge (x', y') is called the *deficient edge*. We say that the edge (x, y) *creates* (x', y').■

Our method of constructing an MSEO will be to first construct an MEO using the algorithm of Dahlhaus and Karpinski [27] and then each edge will be examined to determine the "deficient edge" as defined in Definition 3.2.2. The following lemma proves that given a strongly chordal graph G with PEO (<), the ordering < is not an SPEO ordering.

Lemma 3.2.3: A PEO (<) of a strongly chordal graph need not be an SPEO.

Proof: Counterexample. See Figure 3.2.1.■

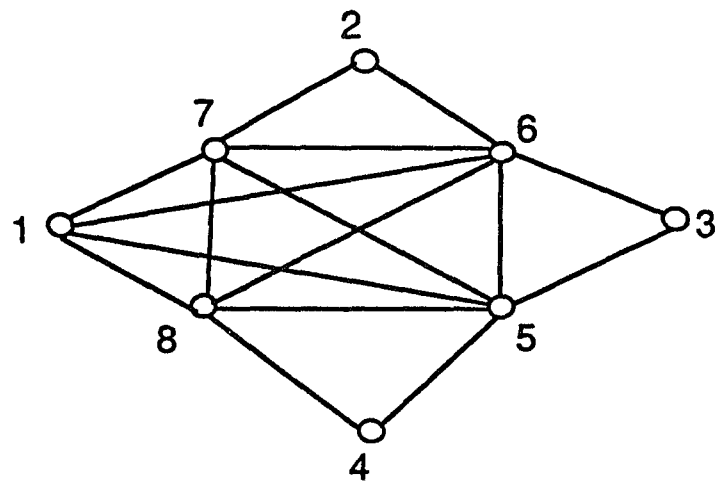

Figure 3.2.1: A strongly chordal graph with PEO numbering which is not an SPEO.

Algorithm Construct_MSEO;
Input : A Graph G with n-vertices and m-edges.
Output : A strongly chordal graph G_{sc}.

Begin
1. Obtain G_c the chordal graph of G.
2. Check if G_c is strongly chordal and if so output G_c; **Stop.**
3. Obtain PEO (<) of the graph G_c.
4. $V_{sc} \leftarrow V_c$; $E_{sc} \leftarrow E_c$.
5. Process each edge (x, y) of E_c **in parallel** as follows
6. **If** (x', y) and $(x, y') \in E_c$ with $x < x'$ and $y < y'$ **then**
7. ADD edge (x', y') to E_{sc}.
End.

Lemma 3.2.4: Let < be the PEO ordering of the graph G which is minimally constructed into an SPEO (G'). For each edge (x', y') in $G'-G$ there exists an edge (x, y) in G which defines (x', y') in G'.

Proof: Let G be chordal and not strongly chordal. Let us assume that an edge (x'', y'') $\in G'-G$ creates (x', y'). We will show that there exists an edge $(x, y) \in G$ which created (x', y'). Let (p, q) be the edge that creates (x'', y''). This implies $p < x''$, $q < y''$, and (p, y''), $(q, x'') \in G'$. Now, since G is chordal either (p, x'') or $(q, y'') \in G'$. Also from assumption $y'' < y'$ and $x'' < x'$. This implies $(x=p, y=y')$ or $(x=q, y=y'')$ will create (x', y'), hence, the Lemma.■

Figure 3.2.2 gives an example of a chordal graph with PEO numbering of the vertices and the dotted lines correspond to the "deficient edges" determined the PEO ordering.

Note that the graph in Figure 3.2.2 is a strongly chordal graph when the dotted edges are included.

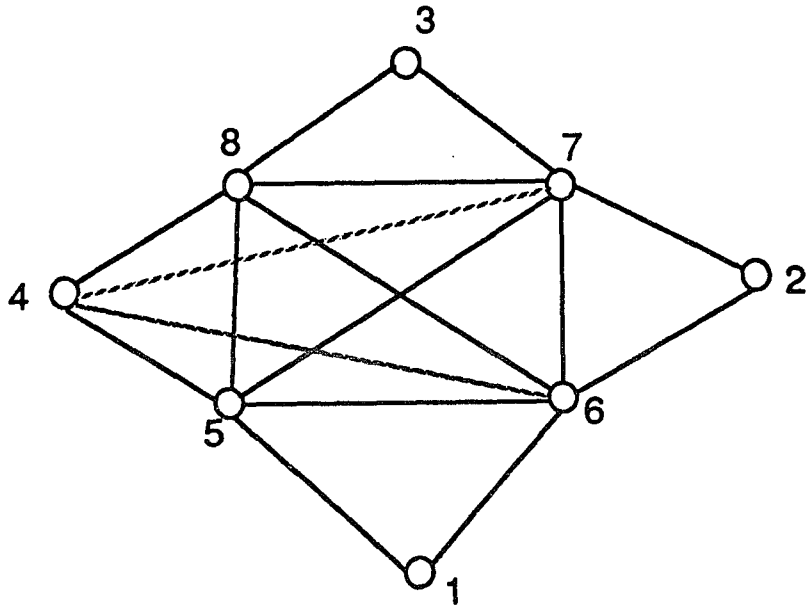

Figure 3.2.2: A chordal graph with minimal edges (dotted) making it strongly chordal.

Theorem 3.2.5: Given an arbitrary graph G with n-vertices and m-edges algorithm Construct_MSEO correctly determines the minimal set of edges needed to make G strongly chordal.

Proof: Since every strongly chordal graph is also a chordal graph, Step 1. determines the minimal set of edges needed to make an arbitrary graph chordal. In Lemma 3.2.3 it was shown that a PEO of a strongly chordal graph need not be an SPEO. In Step 2. we check if the graph obtained at the end of Step 1. is strongly chordal and if so we stop. If the graph obtained is not strongly chordal, then, we determine the a PEO of the graph and minimally transform the PEO into an SPEO by adding the deficient

edges. This is done in Steps 5-7. From Lemma 3.2.4 it is clear that all the deficient edges will be determined. Hence, the theorem.■

Theorem 3.2.6: The complexity of the algorithm Construct_MSEO which is to be implemented on a CRCW-PRAM is listed as follows:

(a) $O(\log^3 n)$ time and $O(nm)$ processors when the input graph is an arbitrary graph.

(b) $O(\log^2 n)$ time and $O((n+m)^{3/2}/\log^2 n)$ processors when the input graph is a chordal graph.

(c) $O(\log^2 n)$ time and $O(n+m)$ processors when the input graph is a chordal graph and not a strongly chordal graph.

(d) $O(1)$ time and $O(n+m)$ processors when the input graph is a chordal graph with PEO and not a strongly chordal graph.

Proof: Step 1. can be executed using the algorithm of Dahlhaus and Karpinski [27] in $O(\log^2 n)$ time using $O(nm)$ processors. Step 2. can be executed in $O(\log^2 n)$ time using $O((n+m)^{3/2}/\log^2 n)$ processors using the algorithm presented in Chapter 2. Step 3. can be executed in $O(\log^2 n)$ time using the algorithm of Klein [60]. Steps 4-7 can be executed in $O(1)$ time using $O(n+m)$ processors. The complexities given in (a)-(d) follows directly from above.■

3. MINIMAL CONSTRUCTION DOUBLY PERFECT ELIMINATION ORDERING (MDEO)

In this section we introduce a new elimination ordering of the vertices of the ptolemaic graph. Using this elimination ordering we determine the minimal set of edges needed to make an arbitrary graph a ptolemaic graph.

Definition 3.2.7: A strongly perfect elimination ordering (<) is called a *doubly perfect elimination ordering* if and only if for each vertex p and edges (p, y'), $(p, x') \in E$, and (x, y), (x, y'), $(x', y) \in E$ with $x < x', y < y'$, we have (x', y'), (x, x'), $(y, y') \in E$.

Theorem 3.2.8: A graph G is ptolemaic if and only if it has a *doubly perfect elimination ordering* (DPEO).

Proof: There are two parts to this proof.

(if-part): Let G be ptolemaic and we will show it has a DPEO. Since, G is ptolemaic it has an SPEO. This implies for (x, y), (x, y'), $(x', y) \in E$ with $x < x', y < x'$ we have $(x', y') \in E$. Now, since G is chordal we have either (x', x) or $(y, y') \in E$. Assume we have $(x', x) \in E$ and $(y', y) \notin E$. There can exists a vertex p such that (p, x') and $(p, y') \in E$. The graph induced by vertices (x, x', y, y', p) is not ptolemaic, contradicting our assumption. This completes the if-part.

(only-if): Let < be the DPEO of a graph G. We will show that the graph G is ptole-

maic. Since a DPEO is also a SPEO, the graph G is strongly chordal. For each p with (x, y), (x', x), (y', y), (x', y'), (p, x'), (p, y') such that $x < x'$ and $y < y'$ the graph induced by vertices (p, x, y, x', y') is ptolemaic.■

Algorithm Construct_MDEO;
Input : A Graph G with n-vertices and m-edges.
Output : A Ptolemaic graph G_p.

Begin
1. Obtain G_{sc} the strongly chordal graph of G.
2. Check if G_{sc} is ptolemaic and if so output G_{sc}; **Stop.**
3. $V_p \leftarrow V_{sc}$; $E_p \leftarrow E_{sc}$.
4. Process each edge (x, y) of E_{sc} **in parallel** as follows
5. **If** $(x', y), (x, y'), (R, x'), (R, y') \in E_{sc}$ with $x < x'$ and $y < y'$ for some R with $R > x$ or $R > y$ and $(R, y), (R, x) \notin E_p$ **Then**
6. ADD edges $(x, x'), (y, y')$ to E_p.
End.

Figure 3.2.3 gives an example of a strongly chordal graph which is made a ptolemaic graph by minimal addition of edges (dotted).

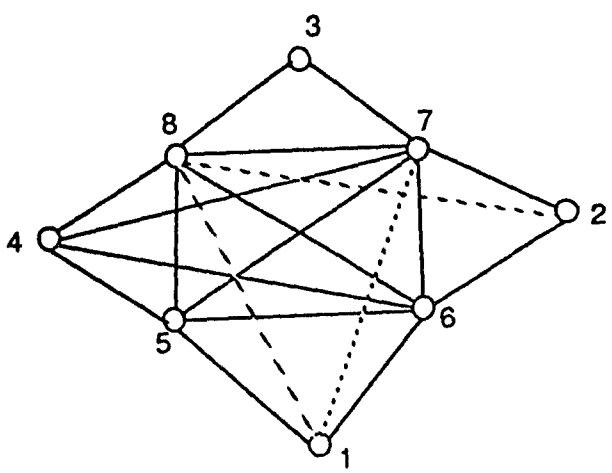

Figure 3.2.3: A strongly chordal graph with minimal edges (dotted) making it a ptolemaic graph.

Theorem 3.2.9: Given an arbitrary graph G with n-vertices and m-edges algorithm Construct_MDEO correctly determines the minimal set of edges needed to make G ptolemaic. The algorithm Construct_MDEO which runs in CRCW-PRAM has the following complexity.

(a) $O(\log^3 n)$ time and $O(nm)$ processors when the input graph is an arbitrary graph.

(b) $O(\log^2 n$ time and $O((n+m)^{3/2}/\log^2 n + M(n))$ processors when the input graph is a strongly chordal graph.

(c) $O(1)$ time and $O(n+m)$ processors when the input graph is strongly chordal and its SPEO is given.

Proof: The correctness of the algorithm is similar to the one presented for Theorem 3.2.5. Step 2. can be checked in $O(\log^2 n)$ time using $O(n+m)$ processors using the algorithm presented in Chapter 2. The SPEO of a strongly chordal graph can be obtained in $O(\log^2 n)$ time using $O((n+m)^{3/2}/\log^2 n + M(n))$ processors (Chapter 2). Steps 3-6 can be executed in $O(1)$ time using $O(n+m)$ processors.■

4. MINIMAL CONSTRUCTION OF BLOCK GRAPHS

We have seen earlier that block graphs are biconnected components and each block of the biconnected components is a completely connected subgraph. Tarjan and Vishkin [91] present a parallel algorithm to determine all the blocks of a graph in

$O(logn)$ time using $O(n+m)$ processors on a CRCW-PRAM model. Now, the construction of block graphs can be easily done using the following simple algorithm.

Algorithm Construct_Block_Graph;
Input : A Graph G with n-vertices and m-edges.
Output : A Block graph G_b.

Begin
1. Determine all the blocks of the graph G.
2. Completely connect the vertices in each block.

End.

Figure 3.2.3 gives an example of a graph whose biconnected components are completely connected making it a block graph.

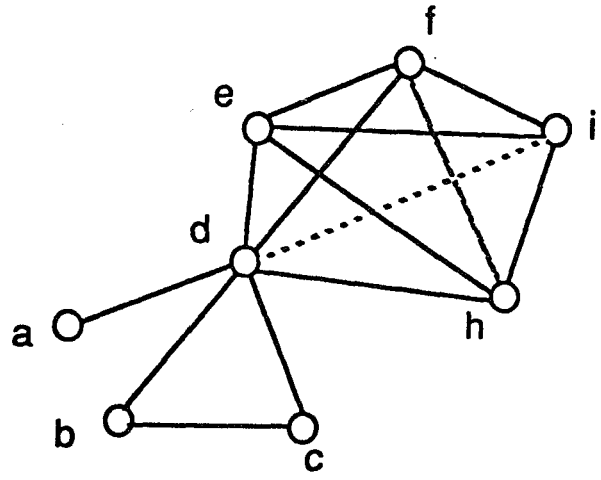

Figure 3.2.3: A graph with minimal edges (dotted) making it a block graph.

Theorem 3.2.10: Given an arbitrary graph G with n-vertices and m-edges algorithm Construct_Block_Graph correctly determines the minimal set of edges needed to

make G a block graph. The algorithm Construct_Block_Graph which runs in CRCW-PRAM has a time complexity of $O(logn)$ time and uses $O(n+m)$ processors.

Proof: The correctness of the algorithm can be easily verified. Step 1. of the above algorithm takes $O(logn)$ time and uses $O(n+m)$ processors. Step 2. can be done using pointer jumping techniques to create the adjacency list.■

5. CONCLUSION

The results obtained in this chapter are summarized in the following table. All algorithms are designed for a CRCW-PRAM.

Input Graph G (n vertices, m edges)	Output Graph	Algorithmic complexity	
		Time	Processor
Arbitrary	Chordal	$O(\log^3 n)$	$O(nm)$
Arbitrary	Strongly Chordal	$O(\log^3 n)$	$O(nm)$
Arbitrary	Ptolemaic	$O(\log^3 n)$	$O(nm)$
Arbitrary	Block	$O(\log n)$	$O(n+m)$
Chordal (not strongly chordal)	Strongly Chordal	$O(\log^2 n)$	$O(n+m)$
Chordal with PEO and not strongly chordal	Strongly Chordal	$O(1)$	$O(n+m)$
Chordal	Strongly Chordal	$O(\log^2 n)$	$O((n+m)^{3/2}/\log^2 n)$
Strongly Chordal with SPEO	Ptolemaic	$O(1)$	$O(n+m)$
Strongly Chordal	Ptolemaic	$O(\log^2 n)$	$O((n+m)^{3/2}/\log^2 n + M(n))$

Chapter Four

EFFICIENT PARALLEL ALGORITHMS FOR DOMINATION PROBLEMS ON STRONGLY CHORDAL GRAPHS

1. INTRODUCTION

The main goal of this chapter is to develop linear-time sequential and efficient parallel algorithms for various domination problems on strongly chordal graphs. A dominating set of a graph $G = (V, E)$ is a set of vertices D such that for every vertex x in V there exists some vertex y in D such that (x, y) is in E. A dominating set is connected if the subgraph $G[D]$ induced by D is connected, total if $G[D]$ has no isolated vertex, and independent if vertices in D are pairwise non-adjacent in G. Given a strongly chordal graph G with n vertices and m edges, we present sequential algorithms with $O(n+m)$ time complexity and fast parallel algorithms with $O(\log^2 n)$ time complexity using $O(n+m)$ processors on a CRCW-PRAM model for the following problems:

(1) Dominating set

(2) Domatic number, i.e., determine the maximum integer K and disjoint vertex sets $V_1, V_2, ..., V_K$ such that each V_i is a dominating set

(3) Connected dominating set

(4) Total dominating set

It has been shown earlier that directed-path graphs, ptolemaic graphs, block graphs, interval graphs, and threshold graphs are also strongly chordal, hence, our results extend to those graphs also.

The domination problem is used in facility location problems, where each vertex represents a customer and a potential site for a facility. A feasible solution corresponds to a set of facilities located at $D \subseteq V$ such that every customer is adjacent to some facility. An optimal solution is a minimum cardinality set of facilities with this property.

The domination problem is NP-complete for undirected path graphs [63], split graphs [62], bipartite graphs [62] and 2-CUBS [25]. Polynomial time algorithms for the domination problem are available in trees [63], series-parallel graphs [63], permutation graphs [25], interval graphs [59] and strongly chordal graphs [20].

Several variations of the domination problem have been studied in the past. A k-dominating set of a graph $G = (V, E)$ is a set of vertices D such that for every vertex x in V there exists some vertex y in D satisfying $d(x, y) \leq k$. Usually, "domination" is used for 1-domination. A dominating set D of G is *independent* if the vertices of D are pairwise non-adjacent, *connected* if the subgraph $G[D]$ induced by D

is connected, and *total* if $G[D]$ has no isolated vertex.

Domination problems for a special class of graphs called the strongly chordal graph [37] or sun-free chordal graph [20] are investigated here. An undirected graph is *chordal* if every cycle of length at least four contains a chord, i.e., an edge between two vertices that are not consecutive in the cycle. Chordal graphs are an important subclass of perfect graphs [47]. A *strongly chordal* graph [37] is a chordal graph that in every even cycle with at least six nodes contains a strong chord (i.e., a chord joining two nodes with an odd distance in the cycle).

Chang [20] presented a linear time algorithms for the k-domination problems in strongly chordal graph given the simple elimination ordering of its vertices. In [4] an $O(n^3)$ algorithm on an n vertex strongly chordal graph for obtaining the simple elimination ordering is presented. Farber [38] presented a linear time sequential algorithm for the weighted 1-domination problem and weighted independent 1-domination problem given the strongly perfect elimination ordering of the vertices of the strongly chordal graph.

Recently, He and Yesha [51] presented an efficient parallel algorithm for the k-domination problem on trees. Bertossi and Moretti [16] developed parallel algorithms for several weighted domination problems on circular-arc graphs. Parallel

algorithms on chordal graphs have attracted some attention recently. Philip Klein [60] presents efficient parallel algorithms on chordal graphs. Klein presents parallel algorithms to determine the maximal cliques and maximum independent set of a chordal graph in $O(\log^2 n)$ time using $O(n+m)$ processors on a CRCW-PRAM model. We use the results of Klein to obtain efficient parallel algorithms for the domination problems and its variants.

Our method of solving the domination problems involves determining the structure of the intersection graph of the maximal cliques of a strongly chordal graph. By the use of an intersection graph we are able to obtain a dominating set which is connected and hence a total dominating set.

It was shown by D'Atri and Moscarini [3] that a ptolemaic graph and a block graph are also strongly chordal. It is also known that directed-path graphs, interval graphs, and threshold graphs are also strongly chordal, hence our results extends to those graphs also.

The model of computation used in our parallel algorithms is the parallel random access machine or PRAM in which all processors have access to a common memory and run synchronously. Both simultaneous reading as well as writing on common memory locations are allowed. Furthermore, in the case of simultaneous writing, an

arbitrary processor succeeds in writing [56].

The rest of the chapter is organized along the following lines. Section 2 presents sequential and parallel algorithms for the construction of the intersection graph of the maximal cliques of a chordal graph. We derive several properties of the intersection graph of a strongly chordal graph. In Section 3 we derive sequential and parallel algorithms for the domination problems based on the properties derived in Section 2. We summarize in Section 4. The results obtained in this chapter appear in [Radhakrishnan and Iyengar] [83].

2. THE INTERSECTION GRAPH OF A STRONGLY CHORDAL GRAPH

In this section, we will define the intersection graph $I(H)$ of the maximal cliques H of a strongly chordal graph G and present a linear time sequential and efficient parallel algorithm to construct the intersection graph. We next derive several properties of the intersection graph of the strongly chordal graph.

Definition 4.2.1 [Intersection Graph]: Let H be the maximal cliques $\{S_1, S_2, ..., S_r\}$, $1 \leq r \leq n$, of a chordal graph G. We can treat H as a hypergraph by treating each maximal clique in H as a hyperedge. The *intersection graph* $I(H)$ of H is a graph containing edges $S_i \in H$ as nodes and edges (S_i, S_j) labeled $l(S_i \cap S_j)$, if $S_i \cap S_j \neq$

Ø. The size of an edge label $l_{ij} = S_i \cap S_j$ is $|S_i \cap S_j|$.■

The following equivalence was proved by D'Atri and Moscarini [3].

Theorem 4.2.1: A graph G is chordal, strongly chordal, ptolemaic, or block if and only if the hypergraph formed using the maximal cliques H of G is α-acyclic, β-acyclic, γ-acyclic, or Berge-acyclic, respectively.■

Definition 4.2.2 [Hypergraph Acyclicity]:

A hypergraph H is β-*acyclic* [36] if it does not contain the following sequence $(S_1, x_1, S_2, x_2, ..., S_m, x_m, S_{m+1})$ satisfying the following conditions:

(i) $x_1, ..., x_m$ are distinct nodes of H;

(ii) $S_1, ..., S_m$ are distinct edges of H, and $S_{m+1} = S_1$;

(iii) $m \geq 3$, that is, there are at least 3 edges involved; and

(iv) x_i is in S_i and S_{i+1} $(1 \leq i \leq m)$ and in no other S_j.■

Fagin [36] proved the following hierarchy of concepts (no reverse implication holds)

Berge-acyclicity ⇒ γ-acyclicity ⇒ β-acyclicity ⇒ α-acyclicity.

Gavril [45] provided a linear time sequential algorithm to determine the maximal cliques of a chordal graph. Since a strongly chordal graph is a chordal graph, we can

use the same algorithm to determine the maximal cliques of a strongly chordal graph in linear time. Klein [60] presented a parallel algorithm to determine the maximal cliques and a minimum clique cover of a chordal graph in $O(\log^2 n)$ time with $O(n+m)$ processors on a CRCW-PRAM. The following combinatorial lemma holds for chordal graphs (see [60]).

Lemma 4.2.2: Let S_v be the set of all maximal cliques of a chordal graph containing vertex v. $\Sigma_v |S_v| = O(n+m)$.■

We now present a linear time algorithm for constructing the intersection graph $I(H)$ given the maximal cliques H represented as a vertex-clique incidence lists.

Algorithm Construct_Intersection_Graph;

Input: Maximal cliques H represented as vertex-clique incidence list M.
$M(i)$ is the list for clique S_i.

Output: Intersection graph $I(H)$ represented as an incidence list L.
$L(i)$ is the pairs $(i_1, l_1), ..., (i_k, l_k)$, where $l_j = S_i \cap S_j \neq \emptyset$.

Data structures:

(1) An $n \times n$ uninitialized matrix R with columns and rows labeled with clique numbers.

(2) $R(i, j)$ points to a set which stores $S_i \cap S_j \neq \emptyset$.

(3) A 'Flag' field in each $R(i, j)$ which is uninitialized.

(4) A multi-set $P(i)$ for each clique S_i. $P(i).j$ means that the jth element of $P(i)$, and $S_i \cap S_{P(i).j} \neq \emptyset$.

Begin

1. **For** $i := 1$ **to** n **do**
2. **For** $j := 1$ **to** $|M(i)|$ **do**
3. **For** $k := 1$ **to** $|M(i)|$ **do**
4. **If** $(j \neq k)$ **then**
   Begin
5. Add vertex v_i into the set in $R(j, k)$;
6. Add the number k into the multi-set $P(j)$;
7. Flag$(j, k) := 0$;
   End;
8. **For** $i := 1$ **to** Number_of_cliques **do**
9. **For** $j := 1$ **to** $|P(i)|$ **do**
10. **If** Flag$(i, P(i).j) = 0$ **then**
    Begin
11. Add the set in $R(i, P(i).j)$ into $L(i)$;
12. Flag$(i, P(i).j) = 1$;
    End;

End.

We will show that the above algorithm has a linear time complexity. First we make the observation from Lemma 4.2.2 that the sum of the sizes of all edge labels of the intersection graph $I(H)$ is $O(n + m)$. The number of times Step 5 would be executed

is clearly $2.(n+m)$. Also the sum of the sizes of all $P(i)$'s is $2.(n+m)$. Thus, Step 10 would be executed at most $2.(n+m)$ times; hence, the time-complexity of the algorithm Construct_Intersection_Graph is $O(n+m)$.

Given a Chordal graph G with n vertices and m edges its maximal cliques H can be determined in $O(\log^2 n)$ time with $O(n+m)$ processors on a CRCW-PRAM [60]. Let the maximal cliques be represented as a vertex-clique incidence lists M. Let $M(i)$ correspond to the incidence list of vertex v_i. The intersection graph $I(H)$ can be constructed as follows. Allocate processors $P^r{}_{i,1}, P^r{}_{i,1}, \ldots, P^r{}_{i,|M(r)|^2}$ to the incidence list of vertex v_r. The processor $P^r{}_{i,j}$ stores the following information: (a). the vertex v_r, (b). the clique number S_i, (c). the adjacent clique number $S_j \neq S_i$. The total number of processors is $O(n+m)$, since the sum of the sizes of all edge labels is $O(n+m)$. Arrange all the processors sorted first by the field (b) and then by the field (c). Now, from the sorted order the edges and its labels from each clique S_i can be easily determined. The sorting can be done in $O(logn)$ time with $O(n+m)$ processors using the integer sorting algorithm of Reif [85]. Thus, the intersection graph $I(H)$ can be constructed in $O(logn)$ time with $O(n+m)$ processors.

2.1. Properties of the intersection graph of a strongly chordal graph

We first derive two necessary conditions for strongly chordal graphs in terms of the intersection graph of its maximal cliques.

Theorem 4.2.3: The *intersection* graph $I(H)$ for a set of maximal cliques H of the graph G is chordal, if G is strongly chordal.

Proof: When graph G is strongly chordal we know from Theorem 4.2.1 that H is β-acyclic. We will show by contradiction that $I(H)$ is chordal. Let H be β-acyclic and $I(H)$ not chordal. Consider the sequence $(S_1, x_1, S_2, x_2, S_3, x_3, S_4, x_4, S_1)$ in $I(H)$. The nodes x_1, x_2, x_3, and x_4 are all distinct (otherwise there would be a chord connecting opposite vertices). The sequence is a β-cycle, which means that H contains a β-cycle, a contradiction. Therefore, $I(H)$ is chordal when G strongly chordal.■

Theorem 4.2.4: If H is β-acyclic, then every triangle (S_i, S_j, S_k) in $I(H)$ satisfies the condition $S_p \cap S_q \subseteq S_p \cap S_r$ and $S_p \cap S_q \subseteq S_q \cap S_r$ for some $p \neq q \neq r$ in $[i, j, k]$.

Proof: Let $S_q \cap S_q = r_1$, $S_q \cap S_r = r_2$, $S_r \cap S_q = r_3$, and H be β-acyclic. If $r_1 \cap r_2 \cap r_3 = \emptyset$, then we have a β-cycle, a contradiction to the assumption. If $r_1 \cap r_2 \cap r_3 = x$ and $x \subset r_i$, $i = 1, 2, 3$, then we have a β-cycle by Definition 4.2.2, a contradiction. The case where $(r_1 = r_2) \cap r_3 = \emptyset$ does not exist, and either $r_3 = r_1$ or the containment stated in the Theorem holds.■

The other properties of the intersection graph are stated in the following propositions.

Proposition 4.2.5: Let H_i^* be a maximal clique of $I(H)$ of a strongly chordal graph. Let l_i be an edge label whose size is minimum among all edge labels in H_i^*. The

label l_i is contained in each vertex of H_i^*.

Proof: Since l_i is the edge label whose size is minimum it is contained in every other edge in H_i^* from Theorem 4.2.4. This implies l_i is in every vertex of H_i^* (i.e., adjacent to all vertices of G in H_i^*).■

Proposition 4.2.6: Let H_i^* and H_j^* be two maximal cliques of $I(H)$ of a strongly chordal graph with minimum edge labels l_i and l_j, respectively. We have $l_i \cap l_j = \emptyset$.

Proof: If $l_i \cap l_j \neq \emptyset$, then, every vertex in H_j^* is connected to every vertex in H_i^* (from Proposition 4.2.5) which implies $H_i^* = H_j^*$, a contradiction.■

Proposition 4.2.7: Let $I(H)$ be a non-trivial intersection graph of a connected strongly chordal graph. At least one vertex of each maximal clique H_i^* (the vertex in H_i^* is a maximal clique of the graph G) is in another maximal clique H_j^* of $I(H)$.

Proof: Since G is connected the above trivially holds.■

The following theorem shows that the set of vertices obtained from each minimum edge label forms the dominating set. The dominating set thus obtained is a connected one and hence a total dominating set.

Theorem 4.2.8: Let $D = \{a_1, a_2, ..., a_k\}$ be a vertex chosen from each of the minimum edge labels $l_1, l_2, ..., l_k$ of each of the clique in the *minimum clique cover* (a

minimum set of cliques which covers all vertices) of the intersection graph $I(H)$ of a strongly chordal graph G. Set D is a minimum dominating set for G which is also connected if G is connected, hence a total dominating set.

Proof: Clearly, D is a dominating set for G from Proposition 4.2.5 and Proposition 4.2.6 and the fact that each clique in the minimum clique cover contains a unique vertex. It is connected from Proposition 4.2.7. A dominating set which is connected does not have any isolated vertices, hence it is a total dominating set.■

It was shown by Farber [38] that the domatic number of a strongly chordal graph is the minimum degree of a vertex of the graph plus one. The *domatic number* is clearly, *minimum*($|l_1|$, ..., $|l_k|$), where l_i is the minimum edge label of the maximal clique H_i^* of $I(H)$ of a strongly chordal graph.

3. DOMINATION PROBLEMS -- SEQUENTIAL AND PARALLEL ALGORITHMS

From Theorem 4.2.8 it is clear that domination problems on strongly chordal graphs can be solved by determining the minimum edge label of each of the clique of the minimum clique cover of the intersection graph $I(H)$ of the strongly chordal graph G. We present the following algorithm to determine the minimum edge labels.

Algorithm Domination;
Input: A strongly chordal graph $G=(V, E)$ with n vertices and m edges.

Output: A minimum cardinality dominating set $D \subseteq V$.

Begin
1. Compute the maximal cliques H of G.
2. Construct the intersection graph $I(H)$.
3. Compute the minimum clique cover H^* of $I(H)$
4. From each clique $H_i^* \in H^*$ choose an edge with minimum edge label size.
5. Choose a vertex from each such edge and add it to D.

End.

The correctness of the above algorithm follows from Theorem 4.2.8. We will now estimate the sequential time complexity of the above algorithm. Since G and $I(H)$ are both chordal (Theorem 4.2.3), using Gavril's [45] algorithm Steps 1 and 3 can be executed in $O(n+m)$ time. Step 5 takes $O(n)$ time once the minimum edges are chosen. The minimum size edges can be chosen in $O(n+m)$ time given the maximal cliques H^*. From the discussion in Section 2 the intersection graph $I(H)$ can be constructed in $O(n+m)$ time. We have the following theorem.

Theorem 4.3.1: The dominating set for a strongly chordal graph with n-vertices and m-edges can be computed in $O(n+m)$ sequential time.■

It can be clearly seen that the algorithm Domination is in NC since Steps 1-5 are all in NC from the discussions in Section 2. We have the following theorem.

Theorem 4.3.2: The dominating set for a strongly chordal graph with n-vertices and m-edges can be computed in $O(\log^2 n)$ parallel time with $O(n+m)$ processors on a

CRCW-PRAM.■

It is clear that the k-domination problem on a strongly chordal graph can be solved by solving the 1-domination problem on the kth power of the strongly chordal graph. The kth *power* of a graph $G = (V, E)$ is the graph $G^k = (V, E^k)$ with $(x, y) \in E^k$ if and only if $1 \leq d_G(x,y) \leq k$. Lubiw [67] proved that powers of a strongly chordal are strongly chordal. Now, the kth power of a graph can be easily obtained in $O(logn)$ time using $O(n^3)$ processors on a CRCW-PRAM and in $O(n^3)$ time sequentially.

4. CONCLUSION

We summarize the results obtained in this chapter in the following table. All the parallel algorithms are designed for a CRCW-PRAM model and n and m are the number of vertices and edges of the input strongly chordal graph, respectively.

Problem	Previous Result [4]	Our Result	
	Sequential time	Sequential time	Parallel time-processor
1-domination	$O(n^3)$	$O(n+m)$	$O(\log^2 n)$-$O(n+m)$
1-connected domination	$O(n^3)$	$O(n+m)$	$O(\log^2 n)$-$O(n+m)$
1-total domination	$O(n^3)$	$O(n+m)$	$O(\log^2 n)$-$O(n+m)$
k-domination	$O(n^3)$	$O(n^3)$	$O(\log^2 n)$-$O(n^3)$
k-connected domination	$O(n^3)$	$O(n^3)$	$O(\log^2 n)$-$O(n^3)$
k-total domination	-	$O(n^3)$	$O(\log^2 n)$-$O(n^3)$
1-Domatic Number	-	$O(n+m)$	$O(\log^2 n)$-$O(n+m)$
k-Domatic Number	-	$O(n^3)$	$O(\log^2 n)$;$O(n^3)$

Table 1: Sequential time complexity, parallel time and processor complexity of various domination problems on strongly chordal graphs.

There are several open problems. It is not yet know whether linear-time sequential algorithm (except with preprocessing cost of $O(n^3)$) exists for k-domination problems on strongly chordal graphs. Parallel algorithms for weighted domination problems are also open.

Chapter Five

THE COMPLEXITY OF PROCESSING IMPLICATION ON QUERIES AND CHORDAL GRAPHS

1. INTRODUCTION

The query implication problem ($Q_1 \rightarrow Q_2$) on two queries Q_1 and Q_2 is to determine whether the data retrieved by the query Q_1 is always a subset of the data retrieved by Q_2. The query implication problem has applications in the areas of computational geometry, distributed databases, and others. In this chapter we study the general implication problem in which all six comparison operators: $=, \neq, <, >, \leq, \geq$, as well as conjunctions and disjunctions, are allowed. It is shown here that the general implication problem is not even in *NP* and in fact complete in Π^p_2. In the simple case where the comparison operator is only '=', we show that the implication problem is *NP*-Complete. We define a class of queries called 'acyclic queries or chordal' and show the existence of polynomial-time algorithms for the implication problems which are shown to be *NP*-Complete. The results contained in this chapter appear in [Radhakrishnan and Iyengar [80]].

We use the above results to estimate the time-complexity of determining whether two update transactions consisting of *insert* and *delete* operations are equivalent.

Conjunctive queries arise in the area of query optimization in relational databases [68]. We show that the testing implication of two conjunctive queries with inequalities is $\Pi^P{}_2$-Complete.

A *query condition* is a conjunction of *literals* and each literal is of the form '$A\ \Theta\ X$', where A is an attribute, X is an attribute or a constant and Θ is one of the six comparison operators $=, \neq, <, \leq, >, \geq$. A query condition on a d-dimensional cartesian space specifies a set of points to be retrieved. The *query implication problem* $Q_1 \rightarrow Q_2$, is to determine if any point in the d-dimensional cartesian space that satisfies Q_1 will also satisfy the query set Q_2, or in otherwords testing if the points retrieved by query Q_1 is always a subset of the points retrieved by query Q_2. Assuming that the set of points in the d-dimensional cartesian space is finite, we can solve the implication problem by actually retrieving points that satisfy Q_1 and check each point retrieved to see if it satisfies Q_2. So, there is a polynomial time algorithm which will solve the implication problem for a finite set of points in the cartesian space. We are interested in determining the complexity of solving the implication problem without any knowledge of the point space.

A detailed study of the query implication problem was presented by Xian-He et. al [90]. They showed that the general implication problem is *NP*-Hard by transforming the satisfiability problem into the general implication problem. They present a

polynomial time algorithm to solve the implication problem when disjunctions and the '$\neq$' operator are not allowed. We show that the general implication problem is Π^P_2-Complete. This does not mean that the transformation to the satisfiability problem in [90] is incorrect. It only means that the transformation to satisfiability problem could have been very well done using literals of the form '$A = c$' or '$A \neq c$', where c is a constant. Our work does a "fine-grain" estimation of the complexities of the implication problem by considering different allowable comparison operators and literals.

The problem of determining the equivalence of update transactions consisting of *insert*, *delete*, and *modify* operations on a relational database was presented by Abiteboul and Vianu [2]. Determining transaction equivalence is useful in optimizing a given transaction. The delete condition and modify condition were simple selections containing literals '$A = c$' and '$A \neq c$', where c is a constant. Finding the complexities of the query implication problem based on query conditions is useful in determining the complexity of the update transaction equivalence. We consider this problem in Section 5.

Relational database query containment, equivalence, and minimization problems have received considerable attention (see [61]). The equivalence problem is that of finding whether two queries always retrieve the same data. The containment problem is to determine whether the results of one query are always a subset of the results of

another query. The minimization problem is the problem of finding a query equivalent to the given one and having the fewest number of joins. For arbitrary relational calculus or relational algebra queries these problems are undecidable. For more details see [61]. A class of queries called conjunctive queries with inequalities were studied by Klug [61]. It allows queries of the kind "get all students whose GPA is greater than 3.5." Estimating the complexity of the implication problem would help in estimating the complexity of problems involving conjunctive queries with inequalities. We study this problem in Section 4.

The problem of relations with *null-values* in relational databases has received considerable attention [66]. Null-values are values present, but currently not known. A relation with null values represents data with incomplete information. An incomplete information database represents sets of possible worlds [1]. Data complexity of a query is defined as a function of the database size. Abiteboul, Kanellakis, and Grahne [1] determine the data complexity of querying and the complexity of containment test of one query in the other in the case of incomplete information database. The incomplete information database they consider are very general in that they allow conditions on null values. Our results on the query implication problem are obtained by transforming the query containment problem on incomplete databases with conditions on nulls to the query implication problem. This is discussed in Section 2.

In Section 3 we consider a class of query conditions called the 'acyclic' ones and show the existence of a polynomial time algorithm for implication problems which are shown to be *NP*-Complete. This is done along the lines of Goodman and Shmueli [48] who show that certain *NP*-Complete problems can be done in polynomial time when the input is restricted to tree schemas.

Finally, to end the introduction we note that the following *derivability problem* the is same as the query implication problem which arises in distributed databases [90]. Given a query Q and a set of f stored fragments $T_1, T_2, ..., T_f$, can the query Q be computed from the f-fragments? If Q and $T_1, T_2, ..., T_f$ are reducible to boolean expressions i.e., query conditions, then the derivability problem becomes the query implication problem.

2. THE COMPLEXITY OF QUERY IMPLICATION PROBLEM

First we define different types of query conditions and present the complexity of the implication problem of two sets of query conditions belonging to each of the different types.

Definition: A query condition (QC) is a conjunction of literals. Literals are of the form Z1 Θ Z2, where both Z1 and Z2 are not constants and Θ is a comparison operator. The conditions are on the attributes of the relation $R(X_1, ..., X_n)$. Based on the

allowable comparison operator in the query condition, various types of query conditions are defined as follows:

(i) QC1: $\Theta = \{=\}$; Z2 is a constant.

(ii) QC2: $\Theta = \{=, \neq\}$; Z2 is a constant.

(iii) QC3: $\Theta = \{=\}$; Z2 can be a variable or a constant.

(iv) QC4: $\Theta = \{=, \neq\}$; Z2 can be a variable or a constant.

(v) QC5: $\Theta = \{=, <, <=\}$; Z2 is a constant.

(vi) QC6: $\Theta = \{=, <, <=\}$; Z2 is a variable or a constant.■

A set of query conditions C represents the set of all possible tuples to be retrieved. An instance of a set of query conditions C is a set of tuples which satisfies the query conditions in C. If the query conditions C are of type QC1, then C represents a single instance. If the query conditions C are of type QC2, then C represents more than one instance. So, in general the query conditions can be thought of as representing a set of possible worlds (instances). Codd-tables were used to represent a set of instances of a relation with null values. A Codd-table is a relation of constants and distinct variables called nulls, which stands for values present, but unknown. Given a set of query conditions C in QC1, we can construct the Codd-table as follows. Create a table T with columns $X_1, X_2, ..., X_n$, the attributes of the relation R. For each query condition C_i in C, create row i in T. If there is a literal $X_j = c$, where

c is a constant, then place c in the column j of row i. For the rest of the columns in row i place unique variables not placed in the table so far. The resulting table is clearly a Codd-table.

Abiteboul, Kennellakis, and Grahne [1] extended the Codd-table T to contain conditions on the variables in the Codd-table. The conditions associated with the table T in two ways:

(i) A global condition Φ associated with the entire table, and

(ii) a local condition $\phi(t)$ associated with each tuple t in T.

The global conditions are of type QC3 or QC4. A table T with both global and local condition is called a *c-table*. A table T with only a global condition is called a *g-table*. When the Φ of the g-table is of type QC3, then the g-table is called an *e-table*. A g-table T with global condition Φ of type QC4 containing literals with only '$\neq$' operators is called an *i-table*. We will now formally define an instance of an x-table (x is Codd-,c-,g-,e-, or i-).

An *valuation* σ is a function from variables and constants to constants, such that $\sigma(c) = c$ for each constant. A valuation σ naturally extends to a tuple t of a table T, producing a complete tuple or a fact, and to the entire table T, producing a relation. Let Φ and $\phi(t)$ be the global and local condition associated with the table T and for each tuple t of T, respectively. We say that σ *satisfies* Φ and $\phi(t)$ if its assignment of

constants to variables makes formulas Φ and $\phi(t)$ true. The conditional table represents a set of instances $I = \{R \mid$ there is a valuation σ satisfying Φ, such that relation R consists of those tuples $\sigma(t)$ for which σ satisfies $\phi(t)\}$.

If I_0 and I are two sets of instances, the *containment problem* (CONT) is to check whether one is contained in the other. The following Theorem on the containment of instances represented by various tables was given in [1].

Theorem 5.2.1: Let I_0 and I be the two sets of instances. We have,

(i) CONT(I_0, I) is polynomial time if I_0 and I are represented by a Codd-table.

(ii) CONT(I_0, I) is NP-Complete if I_0 and I are represented by an e-table.

(iii) CONT(I_0, I) is Π_2^P-Complete if I_0 and I are represented by a c-table, g-table, or an i-table. ■

Theorem 5.2.2: Let C_1 and C_2 be two sets of query conditions of type r. The complexity of implication testing is as follows:

(i) Polynomial time if r is QC1.

(ii) NP-Complete if r is QC2.

(iii) NP-Complete if r is QC3.

(iv) Π_2^P-Complete if r is QC4.

(v) Polynomial time if r is QC5.

Proof: We will prove each one of the above cases as follows.

(i). From a previous discussion it can be easily shown that a Codd-table is equivalent to a set of query conditions in QC1. Hence from Theorem 5.2.1 the implication of two sets of query conditions in QC1 can be tested in polynomial time.

(ii). We will first show that it is NP-Complete to test the implication of two DNF expressions. We then give the transformation of a DNF expression to a set of query conditions in QC2 and thus show that it is NP-Complete to test the equivalence of two sets of query conditions in QC3. Let F_1 and F_2 be two DNF expressions. We have to test the implication $F_1 \rightarrow F_2$ for tautalogy. $F_1 \rightarrow F_2$ is a tautalogy iff $NOT(F_1) \vee F_2$ is tautalogy. For $NOT(F_1) \vee F_2$ to be a tautalogy either F_2 is a tautalogy or F_1 is a non-tautalogy. Checking whether F_1 is is a non-tautalogy is NP-Complete [44]. Now, for each DNF expression F_i we generate a set of query conditions C_i in QC2 as follows. Let F_{ij} be a disjunct in the expression F_i. The query condition C_{ij} in C_i is formed by placing a literal $X_i = c_i$ in C_{ij} for each literal X_i in F_{ij}. For the literal $\overline{X_i}$ in F_{ij} place the literal $X_i \neq c_i$ in C_{ij}. The resulting set of query conditions is clearly in QC2, hence it is NP-Complete to test the implication of two sets of query conditions in QC2.

(iii). We will show that an e-table can be transformed into a set of query conditions in QC3 and thus show from Theorem 5.2.1 that testing implication of two sets of query

conditions in QC3 is NP-Complete. Let us assume that the columns of the e-table T are labeled with $X_1, X_2, ..., X_n$. For each row T_i of the e-table a query condition C_i is formed as follows. If T_{ij} (the jth column of the ith row of T) is equal to M, then form a conjunct $X_j = M$ in C_i. The resulting set of query conditions are clearly in QC3. From Theorem 5.2.1 we conclude that the testing implication of two sets of query conditions in QC3 is NP-Complete.

(iv). The Π_2^P-completeness result for testing two sets of query conditions in QC4 follows from the transformation of a g-table to a set of query conditions in QC4 similar the above and by using Theorem 5.2.1.

(v). The implication of two sets of query conditions in QC5 can be tested in polynomial time by syntactic checks after minimization of each query condition. Klug [61] presents a simple method to minimize query conditions which are in QC5.■

3. ACYCLIC QUERY CONDITIONS

Goodman and Shmueli [48] have shown that the satisfiability of a k-CNF expression with m-clauses can be determined in $O(m.(k.2^k)^2)$ time for acyclic formulas. Acyclic formulas are defined as follows. A boolean formula F is in *k-Conjunctive Normal Form* (CNF) with m-clauses if $F = F_1 \wedge ... \wedge F_m$, where each conjunct F_i is of the form $(l_1 \vee ... \vee l_k)$ and each literal l_i is either a boolean variable x_j or its negation $\overline{x_j}$. With each $F = F_1 \wedge ... \wedge F_m$ we associate a schema $C = (S_1, ..., S_m)$, by

defining

$S_i = \{x_j \mid x_j \text{ or } \overline{x_j} \text{ appears in } F_i\}$. C is a *acyclic schema* if the graph $G = (V, E)$ with $V = \{x_1, ..., x_p\}$, where x_i is in some S_j, $1 \leq j \leq m$ and $E = \{(x_i, x_p) \mid x_i \text{ and } x_p$ are in some S_j, $1 \leq j \leq m\}$, is chordal and conformal [36]. In fact, there is a linear time sequential algorithm to determine if a given schema is acyclic [92]. The schema C is *reduced* if $S_i \subseteq S_j$ implies $i = j$. A formula F with schema C is acyclic (resp. reduced) if the schema C is acyclic (resp. reduced).

Using the construction of a boolean formula in Theorem 5.2.2 we can naturally extend the acyclic formula to *acyclic query conditions* for query conditions in QC2. We say a query condition set QC1 with boolean formula F_1 acyclic if the $\overline{F_1}$ is an acyclic formula. For two sets of acyclic query conditions Q_1 and Q_2 with formulas F_1 and F_2, we will show that the implication problem $Q_1 \rightarrow Q_2$ is in polynomial time as follows. In order to show $Q_1 \rightarrow Q_2$ it is suffices to show that $F_1 \rightarrow F_2$ or $\overline{F_2} \rightarrow \overline{F_1}$. Consider that $\overline{F_2}$ does not have more than m-clauses with at most k literals per clause. The truth-assignments for $\overline{F_2}$ can be obtained in $O(m.(k.2^k)^2)$ time [48]. We can similarly generate truth-assignments for $\overline{F_2}$ and in polynomial time we can check whether truth-assignments of $\overline{F_2}$ are a subset of the assignments of $\overline{F_1}$

Theorem 5.3.1: The implication problem for two sets of acyclic queries in DC2 can

be solved in polynomial time.

Proof: Follows from above discussion.■

It can be easily shown by construction that the above boolean expression can be constructed for a query set in QC3 similar to the one for QC2 and hence Theorem 5.3.1 will hold for query sets in QC3.

4. CONJUNCTIVE QUERIES WITH INEQUALITIES

Klug [61] considered a class of conjunctive tableau queries which contained inequality conditions on the attributes of the relations and showed that the containment problem is in Π_2^P. It was left open to determine if the containment is Π_2^P-Complete. For a subclass of conjunctive queries with inequalities called the left- or right- semi-interval queries Klug showed that the containment test is in NP. In this section we first show that it is NP-Complete to test the containment of left- or right- semi-interval tableau queries by transforming the problem to the containment problem of e-tables. The containment problem of conjunctive tableau queries with inequalities is shown to be Π_2^P-Complete by transforming the problem to the containment problem of g-tables.

A *conjunctive query* Q can be specified as follows: (1) a set $X_Q = \{x_1, x_2, ..., x_p\}$ of *distinguished variables*, the sequence $<x_1, ..., x_p>$ being called the *summary row*, or

just the summary; (2) a set $Y_Q = \{y_1, ..., y_q\}$ of *nondistinguished* (existentially quantified) *variables*; (3) a set $C_Q = \{c_1, ... c_r\}$ of distinct *conjuncts*, each conjunct c_i being an atomic formula of the form $R(z_1, ... z_m)$, where R is a relation name and each z_i is a variable (either distinguished or nondistinguished); and (4) a set $L_Q = \{l_1, ..., l_u\}$ of *inequalities*, each inequality l_i being an atomic formula of the form $z_1 \theta z_2$, where z_1 and z_2 are either variables or constants (but not both are constants) and θ is $=, <, \leq$. A conjunctive query with inequalities are all of the form $x \theta y$ is called a *left semiinterval query* (right semiinterval) if x (y) is a variable and y (x) is a constant.

We will now show that containment of test for left (right) semiinterval queries is NP-Complete. Consider the query of the following form.

$$Q_1: \{x_1, y_1, z_1: (\exists x_1, y_1, z_1)(R_1(x_1, y_1) \& R_2(y_1, z_1) \& x_1 = c_1 \& y_1 = c_2)\}.$$

The query Q_1 is a left (right) semiinterval query. The query Q_1 is equivalent to the query Q_2.

$$Q_2: \{R_1.x_1, R_1.y_1.R_2.z_1: (\exists R_1.x_1, R_1.y_1, R_2.y_1, R_2.z_1)$$
$$(R_1R_2(R_1.x_1, R_1.y_1, R_2.y_1, R_2.z_1) \&$$
$$R_1.y_1 = R_2.y_1 \& R_1.x_1 = c_1 \& R_1.y_1 = c_2)\}$$

The query Q_2 is formed after taking the join of relations R_1 and R_2 and adding an

equality condition on attributes common to R_1 and R_2. The query Q_2 is not a left (right) semiinterval query, but given the queries in the of form Q_2 we can form an equivalent query which is a left (right) semiinterval query. Given an e-table a set of queries of the form Q_2 can be formed which can then be converted into a set of left (right) semiinterval queries. Now, using Theorem 5.2.1 we conclude that testing the containment of two sets of left (right) semiinterval queries is NP-Complete.

Consider the following query Q_1 with the inequalities containing an $\neq$ operator. Assume the domains of the attributes of all relations are integers.

$$Q_1\colon \{x, y, z \cdots : \exists (x, y, ..)(R(x, y, \cdots) \,\&\, x \neq y \,\&\, z \leq c_1 \,\&\, ...)\}$$

The query Q_1 is equivalent to $Q_2 \cup Q_3$, where

$$Q_2\colon \{x, y, z \cdots : \exists (x, y, ..)(R(x, y, \cdots) \,\&\, x < y \,\&\, z \leq c_1 \,\&\, ...)\}$$

$$Q_3\colon \{x, y, z \cdots : \exists (x, y, ..)(R(x, y, \cdots) \,\&\, y < x \,\&\, z \leq c_1 \,\&\, ...)\}$$

Clearly, the g-table can be represented as a set of queries like Q_1, which can be further decomposed into queries like Q_2 and Q_3. Thus, from Theorem 5.2.1 testing containment of two sets of conjunctive queries containing inequalities is Π_2^P-Complete. It is now easy to show that testing equivalence of two sets of query conditions in QC6 is Π_2^P-Complete.

Theorem 5.4.1: Let Q_1 and Q_2 be the sets of conjunctive queries with inequalities.

(i) The containment test is NP-Complete if the conjunctive queries are left (right) semiinterval queries.

(ii) The containment test is Π_2^P-Complete for conjunctive queries with inequalities.

Proof: Follows from discussion above.■

Theorem 5.4.2: Let C_1 and C_2 be two sets of query conditions in QC6. It is Π_2^P-Complete to test if C_1 is equivalent to C_2.

Proof: Can be easily shown using Theorem 5.4.1.■

5. EQUIVALENCE OF UPDATE TRANSACTIONS

A transaction t consists of update operators (*insert, delete, and modify*) which access tuples in $T(U)$, where U is the set of all attributes forming the database scheme. The semantics of the update operators are described as follows.

(1) *insert* - Insert tuples in $T(U)$.

(2) *delete* - Delete tuples satisfying condition C.

(3) *modify* - Modify tuples satisfying condition C to tuples C'. (Note that *modify* is equivalent to delete tuples satisfying C and insert tuples of the form C').

A delete operation is *valid* if there exist, tuples in the database D in state S $(D(S))$ matching the delete condition. Similarly an insert operation is valid if it inserts tuples

which are not already in $D(S)$. A transaction is valid if consists of valid delete and insert operations. For ease of presentation we do not consider the modify action and consider transactions with delete and insert operations only.

An update transaction described above changes the state of the database from S to S'. In fact, every update operator changes the state of the database. We say a transaction is *cyclic* if there exists at least one sequence of actions A_i, A_{i+1}, ..., A_k such that after execution of the action A_k we obtain the database state S' which was the state of the database before the execution of A_i. A transaction which is not cyclic is termed *acyclic*. In this paper we consider only transactions which are *acyclic*. The following lemma shows that in polynomial-time we can decide whether a transaction is *cyclic* or *acyclic*.

Lemma 5.5.1 [2]: The determination whether an update transaction is cyclic or acyclic can be done in polynomial time.■

We shall proceed to define transaction equivalence. The state of the database is determined by the set of tuples in $T(U)$. Two transactions t_i and t_j are *equivalent* if and only if they leave the database in the same final state. Testing whether the final states are equivalent amounts to testing whether at the end of both the transactions the database consists of the same set of tuples. This task is very expensive if we assume

an infinite database. Here, we use a simple but efficient technique for finding equivalence of two acyclic update transactions.

Creation of partitions containing tuples

For each transaction t_i we create two partitions, the *delete* (D_i) partition and the *insert* partition (I_i). The delete partition consists of tuples which are deleted and the insert partition consists of tuples which are inserted. When a delete action is processed we place tuples from the database and the insert partition which satisfies the delete condition in D_i. Tuples which have to be inserted are placed in I_i.

Lemma 5.5.2: If t_i is an acyclic and valid transaction, then $D_i \cap I_i = \emptyset$.

Proof: We will prove by contradiction. Assume there exists a tuple $C_j \in D_i \cap I_i$. This is possible only if we have deleted tuple C_j first and then added to the insert partition the tuple C_j, as described in the creation process of the delete and insert partition. Since C_j is disjoint (i.e., there is no other tuple C_i in D_i or I_i such that $C_i = C_j$) from all other tuples, the delete and insert operations can be thought of as operations occurring next to each other in the transaction t_i. We now have $D(S) \rightarrow_{delete(C_j)} D(S') \rightarrow_{insert(C_j)} D(S)$. But, this is cyclic and contrary to our assumption of the acyclicity of the transaction t_i. Hence the Lemma.■

We will now proceed to show that transaction equivalence is equal to testing for equivalences of partitions.

Theorem 5.5.3: Two acyclic, valid transactions t_i and t_j are equivalent if and only if $D_i = D_j$ and $I_i = I_j$, where the D's are the tuples in the delete partitions and the I's are the tuples in the insert partitions obtained by the creation process described above.

Proof:

(*If-part*): Assume t_i and t_j are equivalent, then $D(S) \rightarrow_{t_i} D(S')$ and $D(S) \rightarrow_{t_j} D(S')$. Now for transaction t_i, $D(S') = \{D(S) - D_i\} \cup I_i$. Since the final database states of both transaction are the same we have $\{D(S) - D_i\} \cup I_i = \{D(S) - D_j\} \cup I_j$. Using Lemma 5.5.2 we can clearly see that $D_i = D_j$ and $I_i = I_j$.

(*Only-if-part*): Assume $D_i = D_j$ and $I_i = I_j$, we have for transaction t_i the final state

$D(S') = \{D(S) - D_i\} \cup I_i$

$= \{D(S) - D_j\} \cup I_i$

$= D(S')$ which is the final state for transaction t_j.

Hence, transactions t_i and t_j are equivalent.■

Creation of partitions consisting of delete conditions and inserted tuples

For each transaction t_i we create two partitions, the delete ($\{C_i\}$) partition and the insert (I_i) partition. When a delete action of t_i is processed we delete tuples in I_i

satisfying the delete condition and place the delete condition in $\{C_i\}$. When we process an insert action we place the tuple to be inserted in I_i.

Theorem 5.5.4: Two acyclic, valid transactions t_i and t_j are equivalent iff the sets of delete conditions $\{C_i\}$ and $\{C_j\}$ are equivalent and $I_i = I_j$. Here the $\{C_i\}$'s correspond to the set of delete conditions and I's correspond to the tuples in the insert partition obtained by the creation process described above.

Proof: Similar to the proof of Theorem 5.5.3.■

Now, from Theorem 5.5.4, we can see that the complexity of update transaction equivalence is determined by the complexity of testing the equivalence of two sets of delete conditions. Hence, if we consider the query conditions as delete conditions the complexity results in Theorem 5.2.2 will hold for update transaction equivalence also.

6. CONCLUSION

We have shown that the general query implication problem is not even in NP but complete in Π^p_2. We have estimated the complexities of implication problems by restricting the allowable comparison operators and variables and constants that can be compared. We have shown that the implication problem on queries which are 'acyclic' can be solved in polynomial-time using the technique in [48]. As applications we have estimated the complexity of equivalence testing of two update transactions

and containment of conjunctive queries with inequalities arising in relational databases. The higher-order complexities of the implication problem only warns about the time involved in solving the problem but does not preclude the existence of any algorithm to solve the problem. It would be interesting and useful to design an algorithm to the solve the general implication problem or restricted problems.

Chapter Six

EFFICIENT PARALLEL ALGORITHMS FOR THE MANIPULATION OF DIRECTED HYPERGRAPHS

1. INTRODUCTION

The main focus of this chapter would be to develop efficient parallel algorithms for the manipulation of directed hypergraphs. Directed hypergraphs are generalization of directed graphs in which an arc (i, j) may involve more than two nodes i.e., $|i| \geq 1$ and $|j| \geq 1$. If $|j| > 1$, we call j as a compound node and there are arcs from node i to all the components of j $(j_1 \cup \ldots \cup j_k = j)$. Directed hypergraphs are viewed as structures to represent functional dependencies, where an arc (i, j) is interpreted as i "functionally determines" j. Using the well defined manipulations defined on functional dependencies we would like to manipulate directed hypergraphs. Hence our focus would be to manipulate functional dependencies using directed hypergraphs. In the next paragraphs we would mention the main results obtained in this chapter and give an overview of the manipulations that can be done to a directed hypergraph. Previous results and approaches are also discussed. The results presented in this chapter appear in [Radhakrishnan and Iyengar [81]].

MAIN RESULTS:

Given a set of functional dependencies Σ and a single dependency σ, we show that the algorithm to test whether Σ implies σ is log-space complete in P. The above implication problem is the *membership-test* for functional dependencies or interpreted as finding a directed path between two nodes in the directed hypergraph. The functional dependencies Σ are represented as a directed hypergraph H_Σ [8]. We first present a parallel algorithm which solves the above implication problem using P processors on an EREW-PRAM in $O(e/P + n.logP)$ time and on an CRCW-PRAM in $O(e/P + n)$ time, where e and n are the number of arcs and nodes of the graph H_Σ. For graphs H_Σ with fixed degree and diameter, we show that the closure H_Σ^+ can be computed in NC. The closure operation is finding all the possible arcs in a directed hypergraph. We present NC algorithms to obtain a non-redundant and an LR-Minimum cover for the set of functional dependencies Σ. All our algorithms on an n-node directed hypergraph with fixed degree and diameter can be implemented to run in $O(\log^2 n)$ time with $M(n)$ processors on an CREW-PRAM model, where $M(n)$ is the cost of multiplying two binary matrices. The algorithms are efficient based on the *transitive closure bottleneck* phenomenon [56]; that is, any improvement in the time and processor complexity of the transitive closure algorithm will result in an improvement by the same amount for the algorithms presented here.

MANIPULATION OF FUNCTIONAL DEPENDENCIES (DIRECTED HYPER-GRAPHS) AN INTRODUCTION

Functional dependencies (FDs) and their manipulation plays a decisive role in the design, use, and maintenance of relational databases. The elimination of data redundancy and the enhancement of data reliability can be done by imposing restrictions on the data. Functional dependencies provide a way to impose restrictions on data and prior knowledge about them are useful in designing better relational databases [68, 93].

Given a set of attributes T: $A_1, A_2, ..., A_k$, a *relation scheme* $R(T_1)$ is a subset of attributes T_1 in T. A *relation* R over the scheme $R(T_1)$ is the subset of the cartesian product $\mathrm{DOM}(A_1) \times \mathrm{DOM}(A_2) \times ... \times \mathrm{DOM}(A_r)$, where $A_1, ..., A_r$ are the attributes in T_1. An element of the cartesian product is called a *tuple.* A *functional dependency* $X \rightarrow Y$ (where $X, Y \subseteq T_1$) holds in R iff, given two tuples t_1 and t_2 of R, $t_1.X = t_2.X$ implies $t_1.Y = t_2.Y$. Given a set of FDs Σ, it is important to determine those functional dependencies which are not explicitly expressed but derived from those contained in Σ. Such a derivation is possible using Armstrong's sound and complete set of axioms (see [68, 93]). The Armstrong's axioms are as follows.

Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$.

Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.

The manipulation of Σ involves the following.

(i) (Membership-Test): Given a set of dependencies Σ and a dependency σ, find whether Σ implies σ using the Armstrong's axioms.

(ii) (Closure-Finding): Determine Σ^+ the closure of Σ consisting of all dependencies that can be derived from Σ using the Armstrong's axioms.

(iii) (Minimal Key-Finding):Finding a minimal set $X \subseteq T$ of attributes, such that $X \rightarrow T$ is a member of Σ^+. The attribute set X is called the *minimal key* of the relational scheme $R(T)$.

(iv) (LR-Minimal cover): Finding a set of dependencies Σ_r from Σ such that $\Sigma_r{}^+ = \Sigma^+$ with the following properties.

(a) For any dependency σ in Σ_r, $(\Sigma_r - \sigma)^+ \neq \Sigma^+$.

(b) We say an attribute A in X of the dependency $X \rightarrow Y$ as *extraneous* if $X - A \rightarrow Y$ is in Σ^+. No dependency in Σ_r has extraneous attributes on its left side as well as its right side.

The set Σ_r is called the LR-Minimal cover for Σ.

For discussion about LR-Minimum and the advantages of manipulating the given set of dependencies Σ (see [68, 93]).

Several data structures and sequential algorithms for representation and manipulation of functional dependencies have been proposed in the past [8, 30, 43]. A new graph-theoretic approach which leads to efficient algorithms for manipulation and representation of FDs were introduced in Ausiello et. al [8]. In this approach, the given set of FDs were represented as a directed hypergraph and known graph algorithms like the transitive closure, transitive reduction, and finding *strong connected components*† were modified for manipulating FDs. Using the algorithms of Maier [68], an LR-Minimum is obtained; and with the LR-Minimum set of FDs, the synthesis algorithm [68] can be applied to get the relational schemes. Several theoretical issues based on directed hypergraphs were discussed in [9]. The algorithms of Diederich and Milton [30] for computing minimal covers and synthesizing relations into third normal form do not try to achieve a reduction in the computational complexity of the algorithms in [68]. They present interesting insights into the manipulation algorithms of [68] and suggest techniques for enhancement of those algorithms. For example, in standard methods for synthesizing relations, most dependencies have to be checked a second time for redundancy after grouping dependencies with equivalent left-hand sides. Using the method of Diederich and Milton the dependencies can be characterized in such a way they are checked only once. At the present time we do not know of any parallel algorithm for manipulating functional

† A set of nodes are in a strongly connected component if there are paths from every node to every other node in the strong component.

dependencies.

We will show by a simple reduction technique that the FD-Membership problem is P-Complete (Section 3). Using the directed hypergraphs [8] as the representation scheme for the given set of FDs, we derive parallel manipulation algorithms. Our algorithms unlike the algorithms of Ausiello et. al [8], are highly suitable for parallelization. Our characterization of the FD-manipulations in terms of directed hypergraph representing the FDs are simpler compared to the ones presented in [8]. The algorithms for manipulating the functional dependencies use algorithms for computing transitive closure, transitive reduction, and strongly connected components. In order to construct efficient parallel algorithms for computing transitive reduction and strongly connected components it will be necessary to avoid the use of matrix powering or transitive closure as a subroutine; our inability to do so is sometimes called the *transitive closure bottleneck* [56]. The FD-manipulation algorithms necessarily have to use the transitive closure algorithm as a subroutine and hence, it is also affected by the transitive closure bottleneck phenomenon. We will show in this chapter that our parallel FD-manipulation algorithms are efficient based on the transitive closure bottleneck phenomenon. This is done by showing that all operations other than those involving the transitive closure as a subroutine take $O(logn)$ time with processors at most equal to the size of the directed hypergraph H_Σ representing a set of functional dependencies Σ. First we present a parallel algorithm to obtain the closure H^+ of the

directed hypergraph H. We show that our closure algorithm is in NC for fixed degree and diameter graph H (Section 4). Section 5. presents algorithms to obtain a non-redundant and a LR-Minimum cover and it is also in NC for fixed degree and diameter graph H. From the LR-Minimal cover a minimal key can be easily determined.

2. PRELIMINARIES - DEFINITIONS AND NOTATIONS

Definition 6.1.1 (Directed Hypergraph): A directed hypergraph $H = (V, E)$ consists of nodes and arcs as follows.

nodes: The node set V consists of simple and compound nodes. A compound node j has components $j_1, j_2, ..., j_r, r > 1$ and each j_k is a simple node. A simple node is a node with only one component.

Arcs: The arc set E has the following arcs:

(i) arcs (i, j) from one simple node to another,

(ii) arcs $(j, j_1), ..., (j, j_r)$ from each compound node to its components.

(iii) arcs (i, j) from node i to compound node j if and only if there are arcs (i, j_1), ..., (i, j_r), where $j_1, ..., j_r$ are the components of compound node j. If such an i exists we say that node j is *satisfied* by node i.■

We say that there is a *path* from node i to node j, written $<i, j>$, if and only if there are paths $<i, k>$ and $<k, j>$. Also, there is a path from node i to a compound node

j, if and only if there are paths $<i, j_1>$, ..., $<i, j_r>$, where j_1, ..., j_r are all the components of compound node j.

Definition 6.1.2 (Hypergraph Accessibility Problem (HGAP)):

Given a directed hypergraph $H = (V, E)$, and two distinguished nodes $i, j \in V$, does there exists a path $<i, j>$ in H.■

The above HGAP problem on a n-node directed graph containing only simple nodes, can be solved in $O(\log^2 n)$ time with $M(n)$ processors, where $M(n)$ is the cost of multiplying two binary matrices [56].

We will assume that the set Σ is in *reduced form* as follows

(a) there exist no two FDs $X \rightarrow Y$ and $X' \rightarrow Y'$ such that $X = X'$, and

(b) for all FDs $X \rightarrow Y, X \cap Y = \emptyset$.

Let the given set of FDs Σ be in reduced form and represented by a directed hypergraph H_Σ as follows. For each FD $X \rightarrow Y$ create a compound node X and simple nodes X_1, ... X_r and arcs (X, Y), (X, X_1), ..., (X, X_r) in H_Σ. The nodes X_1, ..., X_r are components of node X. We will denote $n = |\Sigma| = |\Sigma_l| + |\Sigma_r|$ the sum of the length of the strings of attributes appearing on the left (right) side of the dependencies. Also, $e = ||\Sigma||$ will denote the number of FDs in Σ. We will use the notation H instead of H_Σ when the context is clear and call H as a graph instead of a directed hypergraph.

Proposition 6.1.0: Let the given set of functional dependencies (FDs) be represented by a directed hypergraph $H = (V, E)$. The FD-Membership test on the dependency $X \rightarrow Y$ is equivalent to the HGAP instance from node X to node Y.■

Example 1: See Figure 6.1. for a set of FDs and its corresponding directed hypergraph.

$A \rightarrow F$
$A \rightarrow C$
$A \rightarrow B$
$C \rightarrow D$
$FBD \rightarrow H$
$BD \rightarrow I$

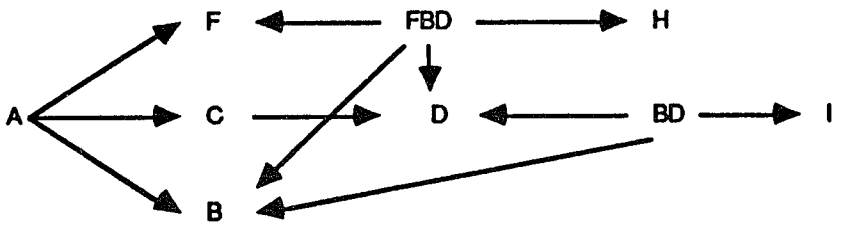

Figure 6.1: A set of FDs and the directed hypergraph corresponding to it (from Ausiello et. al [1]).

Definition 6.1.3: We say a directed hypergraph $H = (V, E)$ *generates* a set of functional dependencies Σ, when, for each arc (X, Y) in E a functional dependency $X \rightarrow Y$ is generated.■

3. THE *P*-COMPLETENESS RESULT AND A PARALLEL ALGORITHM

In this section, we show that the *monotone circuit value problem* is *log-space* reducible to HGAP and thus establish HGAP is *P*-Complete. The monotone circuit

value problem is P-Complete (see [56]).

Definition 6.2.1 (Monotone Circuit Value Problem):

Given a finite set of g gates; for $1 \leq j \leq g$, gate j is either an input (0 or 1), an AND-gate AND($i_{j,1}$, $i_{j,2}$, ..., $i_{j,k(j)}$), or an OR-gate OR($i_{j,1}$, $i_{j,2}$, ..., $i_{j,k(j)}$), where $1 \leq i_{j,1}$, $i_{j,2}$, ..., $i_{j,k(j)} < j$, what is the value of the expression represented by gate g.

Lemma 6.2.1 (see [56]): The monotone circuit value problem is log-space complete in P.■

Theorem 6.2: The HGAP is log-space complete in P.

Proof: We show by the following construction that the monotone circuit value problem is log-space reducible to the HGAP. Consider the case where all the gates have two inputs for the sake of ease in presentation. We construct the following directed hypergraph H. For an AND-gate $g_i = g_j \wedge g_k$, create a compound node g_i with two components g_{i1} and g_{i2}. Add arcs (g_j, g_{i1}) and (g_k, g_{i2}). For an OR-gate $g_i = g_j \vee g_k$, add arcs (g_j, g_{i1}), (g_j, g_{i2}), (g_k, g_{i1}), and (g_k, g_{i2}). We can easily show by induction that on an input 1 at gate g_i, an output of 1 is obtained at gate g; if and only if there exists a directed path from node g_i to node g in H. The construction of H can be done in log-space. Hence the theorem.■

Coro 6.2.1: The FD-membership test is log-space complete in P.

Proof: Follows directly from Proposition 6.1 and Theorem 6.2.■

The negative result in Theorem 6.2. only tells us that HGAP is resistant to high-degree parallelism. We present a simple sequential algorithm for the HGAP which runs in time $O(e+n)$, where e and n are the number of edges and vertices of the graph H. A parallel version of the sequential algorithm runs in time $O(e/P + n.logP)$ with P processors on an EREW-PRAM and in time $O(e/P + n)$ with P processors on an CRCW-PRAM. The technique used in the following algorithm is similar in spirit to the one presented for the monotone circuit value problem by Vitter and Simmons [95].

(* Initially all vertices are marked "not visited." *)

Algorithm HGAP (x, y)

Begin

1. Starting from x determine all the k vertices that can be reached from x by using transitive closure; Mark all the k vertices "visited" including x.
2. If y is one of the k vertices, then RETURN ('Found'); STOP.
3. If either $k = 0$ or there is no arc (x, p) such that vertex p is a component of some compound node, then RETURN ('Nil').
4. For each "unvisited" compound node j, such that there is at least one arc (x, j_r), where j_r is a component of the node j, Do
   Begin
5. If node j is *satisfied* by x, then
   Begin
6. ADD arc (x, j)
7. HGAP (j, y)
   End
   End

End.

It can be easily seen that if there should exist a path $<x, y>$ and, has not been determined at the end of Step 3., then there exists at least one compound node in H which is satisfied by x. The algorithm HGAP presented above can be parallelized in several ways. Each of the steps 1-7 can be parallelized. Step 4-7 is executed sequentially and the processors are assigned to keep track if node j is *satisfied* by node x. Each of the P processors are assigned to check the presence of the arc from x to the Pth component of j. Once each processor determines the presence/absence of arcs assigned to it, the time taken to check if node j is satisfied by x is $O(logP)$ using binary-tree communication scheme among P processors. Essentially, we are computing AND of P binary values. On a CRCW-PRAM, we can determine the AND of P binary values using P processors in constant time.

Theorem 6.3: The HGAP can be solved using P processors on an EREW-PRAM in $O(e/P + n.logP)$ time or on a CRCW-PRAM in $O(e/P + n)$ time.

Proof: Follows from the discussion above.■

4. CLOSURE OF A DIRECTED HYPERGRAPH

Computing the closure of a directed hypergraph H is finding all the possible arcs in the graph H. The closure of the graph H is the transitive closure on directed graphs when H contains only simple nodes. Since, finding whether there exists a path

$<X, Y>$ is P-Complete, determining the closure is also P-Complete. In this section, we present a parallel algorithm whose execution time is dependent on the *diameter* and the *degree* of the graph H. The diameter of the graph H is the maximum distance between any two nodes in H. The degree of the graph H is the degree of a node having maximum number of arcs going out. For graphs with fixed diameter and degree algorithm, we show that the closure can be computed in NC. Having determined the closure the HGAP problem can be solved in constant time. In terms of the functional dependencies Σ, the degree of a node X in H_Σ is the number of FDs in Σ whose left hand side is in X or equal to X. The distance between two nodes X and Y in H_Σ is the number of dependencies in Σ which have to be applied before X determines Y. In the worst case the maximum distance and degree can both be equal to the number of FDs in Σ.

Algorithm H-Graph-Closure

Begin

1. Perform transitive closure on H. Here we find all simple nodes that can be reached from node i and add arcs to nodes reachable from i.
2. **Do** Steps 3-9 **Until** no new arcs are added
3. For each node i **In-Parallel**
4. If there are arcs $(i, j_1), ..., (i, j_r)$, where $j_1, .., j_r$ are *all* the components of compound node j, then
   Begin
5. Add the arc (i, j) in H
6. For all arcs (j, k) add arcs (i, k)
   End
7. For each node i **In-Parallel**

Begin

8. Let $\mathbf{J} = \{j^1, ..., j^k\}$ be the compound nodes such that there is an arc (i, j_p), where j_p is a component of node $j \in \mathbf{J}$ AND for all components p of j, there are arcs only from compound nodes in **J**.
9. Add arcs (i, j) for each $j \in \mathbf{J}$ and arcs (i, k) such that node k is adjacent to some compound node in **J**

End

End.

Theorem 6.4: The algorithm H-Graph-Closure correctly determines the closure of directed hypergraph H in $O(\log^2 n + MAX(diameter(H), degree(H))*logn)$ with $O(M(n))$ processors on a CREW-PRAM.

Proof: It is straightforward to understand Steps 1-6 of the algorithm H-Graph-Closure. Step 8. performs the closure operation with respect to a node i. Let us assume that there should exist an arc (i, j) in the closure of H and not in H after the execution of Step 6. The absence of the arc (i, j) implies that there exists a compound node k in the directed path between node i to node j. Assume there is an arc (k, j), if it is absent, we have the case described above. The arc (i, k) is absent, otherwise, we would have the arc (i, j). For the arc (i, k) to be present, there should be arcs from node i to every component of k. In Steps 7-9 determines the arcs from node i to the components of k. It can be easily shown, that if there should exists an arc (i, j) in the closure and has not been determined at the end of Step 6., then there exists at least one compound node k in **J** of Step 8.

In Figure 6.2. and Figure 6.3. we have depicted the worst-case scenario in terms of the number of iterations of Steps 3-9 before the arc (i, j) is determined. We can easily show that at most MAX(diameter(H), degree(H)) iterations of Steps 3-9 would be necessary to determine the closure. Step 1. takes $O(\log^2 n)$ time to determine the closure with $O(M(n))$ processors (see [56]). Each of the Steps 3-9 can be executed in $O(\log n)$ time with $O(n+e)$ processors using suitable matrix structures on a CRCW-PRAM.■

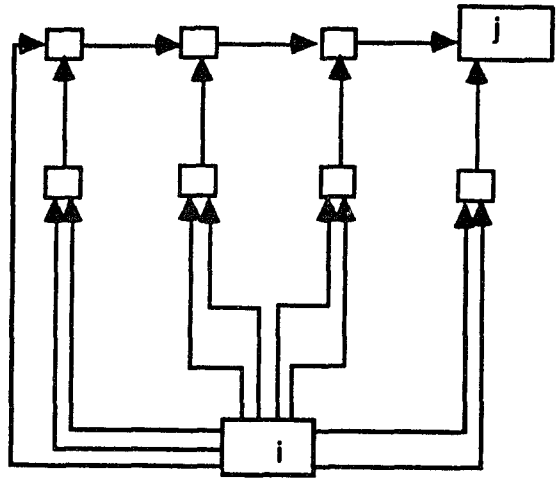

Figure 6.2: The number of iterations of steps 3-9 in algorithm H-Graph-Closure to determine the arc (i, j) is at most equal to the degree of the above graph. The graph above consists of compound nodes with two components each.

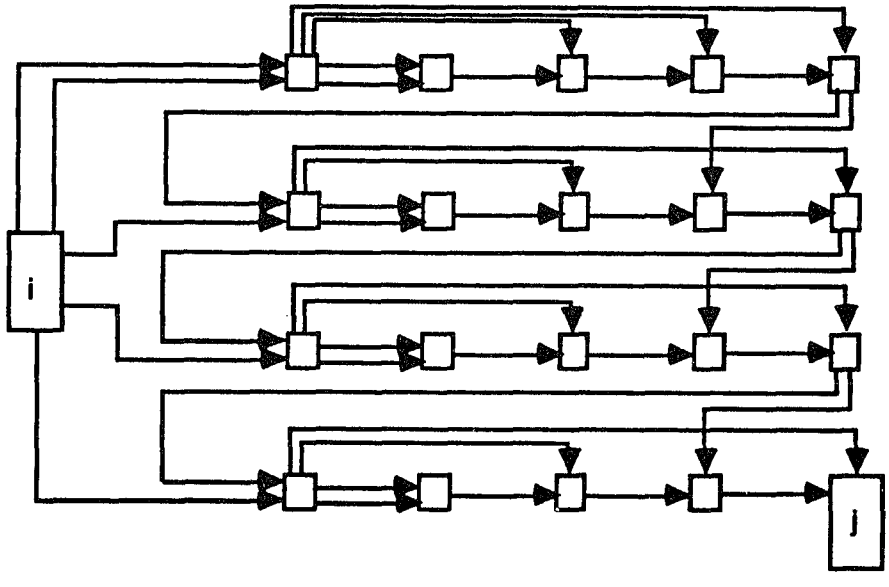

Figure 6.3: The number of iterations of steps 3-9 needed by algorithm H-Graph-Closure to determine the arc (i, j) is at most equal to the diameter of the above graph.

Example 2: The closure of the graph in Figure 6.1. is given in Figure 6.4.

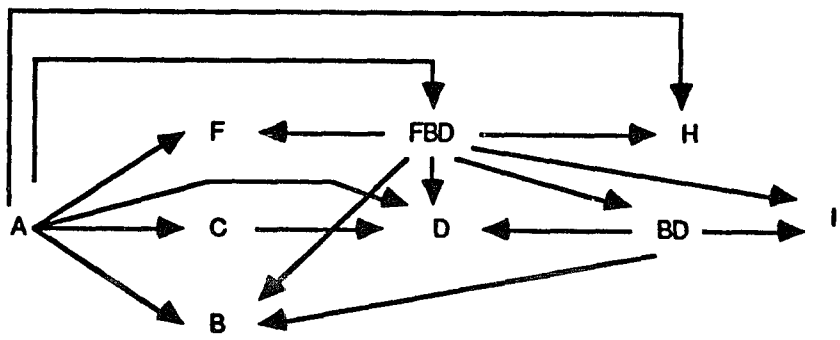

Figure 6.4: The closure of the graph in Figure 6.1.

5. NON-REDUNDANT AND MINIMUM DIRECTED HYPERGRAPHS

Given a set of functional dependencies Σ, we present algorithms to obtain a non-redundant and a minimum cover as defined by Maier [68]. Since, the FD-membership test is P-Complete, algorithms for determining a non-redundant and minimum cover are also P-Complete. In the previous section we presented the closure algorithm which was shown to be in NC for fixed diameter and degree graph H. We will now define several terminologies.

Left reduction involves removal of "extraneous" attributes from X in each of the

dependencies $X \rightarrow Y$ in Σ. Given the set Σ, an attribute B is *extraneous* in $X \rightarrow Y$ if $X = ZB$, $X \neq Z$, and $A \in Z_{\Sigma}^{+}$. There are two kinds of extraneous attributes. If $X = ZB$, $X \neq Z$, and $B \in Z_{\Sigma}^{+}$, then B is called an *implied* extraneous attribute and all other extraneous attributes are *non-implied* extraneous attributes.

We say two attribute sets P and Q are equivalent in Σ written $P \equiv Q$, if $P \rightarrow Q$ and $Q \rightarrow P$ are in Σ^{+}. Let $X \rightarrow Y$ be a dependency with $X \cap Y = \emptyset$ and let $X_1, X_2, \ldots X_l$ be some subsets of X such that $(X_1 \equiv Y_1)$, $(X_2 \equiv Y_2)$, ..., $(X_l \equiv Y_m)$ with $Y_1 \cup Y_2 \cup \ldots \cup Y_m = Y$. The dependency $X \rightarrow Y$ above is trivial and Y is *textually contained* in X.

A *non-redundant* cover for Σ is the set Σ_r in which all dependencies σ in Σ_r, when removed is not in the closure Σ_r. If we assume that the right hand side of each dependency in Σ_r is a single attribute, then a *minimal cover* for Σ_r can be obtained by removing both the implied and non-implied extraneous attributes from the left hand side of each dependency in Σ_r [93]. A *minimum cover* is a minimal cover with a minimum number of functional dependencies than any other equivalent set. A minimal cover is a minimum cover which does not contain some two dependencies $X \rightarrow A$ and $Y \rightarrow B$, such that X and Y are equivalent.

We will give definitions for non-redundant, minimal, and minimum directed hypergraph H.

A hypergraph H is non-redundant if it does not contain any *redundant arcs*. An arc (i, j) is *redundant* in H if

(i) there are arcs (i, k) and (k, j) in H^+, or

(ii) the node j is textually contained in node i.

Condition (ii) identifies arcs which generate trivial dependencies. A non-redundant hypergraph is minimal if each compound node does not contain any implied extraneous attributes. The non-implied extraneous attributes are removed when redundant arcs satisfying condition (i) is removed. A minimal hypergraph is a minimum one if there are no two arcs (I_1, J) and (I_2, K) in H, where I_1 and I_2 are nodes in the same strongly connected component I with J and K in different strong components. From a minimum hypergraph a minimum set of FDs can be easily generated.

The following algorithm obtains a minimum directed hypergraph H_m from the graph H_Σ for a given set of dependencies Σ.

Algorithm Minimize

Begin

1. Let H_1 be the graph H_Σ with arcs to compound nodes present in H_Σ^+.
2. Compute the strongly connected components of the graph H_1.
3. For each component i **do in parallel**
   Begin
4. If more than two arcs from the component i to the component j, then REMOVE all except one from H_1.
5. Choose a node X as a representative of component i.
6. For all arcs (Y, Z), such that Y is in i and Z not in i, REMOVE (Y, Z) and add (X, Z).
7. Remove nodes j from component i which is textually contained to some node k in component i.
   End.
8. Process the acyclic graph formed by the strong components as follows: Mark the arc (I, J) from strong component I to component J for deletion when representative node j of J is textually contained in representative node i of I.
9. Transitively reduce the acyclic graph formed by the strong components and mark arcs to be deleted.
10. Remove implied extraneous attributes from each compound node i.
11. Remove arcs marked for deletion in Step 8 and Step 9.
12. For each one of the strong-components form a Hamiltonian-Circuit with the nodes in the component.
13. Remove redundant arcs formed due to the Hamiltonian-Circuits by Transitively reducing the graph H_1.
14. For each arc (i, j), where j is a compound node, **do in parallel**
    Begin
16. Add arcs $(i, j_1), ..., (i, j_r)$, where $j_1, ..., j_r$ are components of compound node j.
17. Add arcs $(j, j_1), ..., (j, ..., j_r)$.
    End
18. Transitively reduce the resulting graph and remove the arcs to the compound nodes.

End.

The Steps 1-6 are easy to understand. In Step 7 node j is removed from strong component I if it is textually contained in node k in the same component I. Since j and k

are in the same strong component, if j is textually contained in k, then k is also textually contained in j. Hence, to avoid deleting both j and k from component I, ranks are assigned to each node in component I and j is deleted from component I iff j is textually contained in k and rank(j) < rank(k). The textual containment of two nodes in the same component can be tested as follows. Let j and k be two nodes in the component I with rank(j) < rank(k). Let (I, I_1), ..., (I, I_l) be the arcs from strong component I to strong components I_1, ..., I_l, respectively. For all strong components I_m, $1 \leq m \leq l$, we do the following. If $k_m \subset k$ is in node I_m, then for all nodes j_m in I_m with $j_m \subset j$ remove j_m from j. If j becomes empty, then j is textually contained in k, otherwise it is not.

In Step 8 we delete the arc from strong component I to strong component J when some node j in J is textually contained in some node i in I. In fact, if j and i are representative nodes chosen in Step 5, arc (I, J) can be removed if j is textually contained in i. Now, the test is carried out as follows. Let I_1, ..., I_l be the components such that there are arcs (I, I_1), ..., (I, I_l), $1 \leq m \leq l$, and the arcs from strong component J are to only the strong components I_1, ..., I_l. We perform the operation described previously on the node $j \in J$, if j becomes empty, the arc (I, J) is redundant, i.e., j is textually contained in node $i \in I$, otherwise, arc (I, J) is not redundant. The arcs which are found redundant in this step are removed in Step 11 as they are required to determine and remove implied extraneous attributes.

The removal of implied extraneous attribute from every compound node i in component I is done as follows. Let $I_1, I_2, ..., I_l$ be the strong components which have arcs from compound node I containing i with no I_k having an arc from any of I_j, $1 \leq j \leq l$ and each I_k contains a node $z \subset i$. If each I_k, $1 \leq k \leq l$ is reduced, i.e., no compound node $z \subset i$ in I_k contains extraneous attributes, then pick a node z from each one of the I_k's. The union of all z's gives the compound node i without any implied extraneous attributes. The arcs to the compound nodes are redundant and Steps 14-18 performs the right reduction operation.

The minimal key for a set of dependencies Σ by taking the union of representatives of the strong components with indegree zero in the minimum hypergraph H_m of the graph H_Σ.

Example 3: For an illustration of the algorithm Minimize see Figures 6.5(a) - 6.5(d).

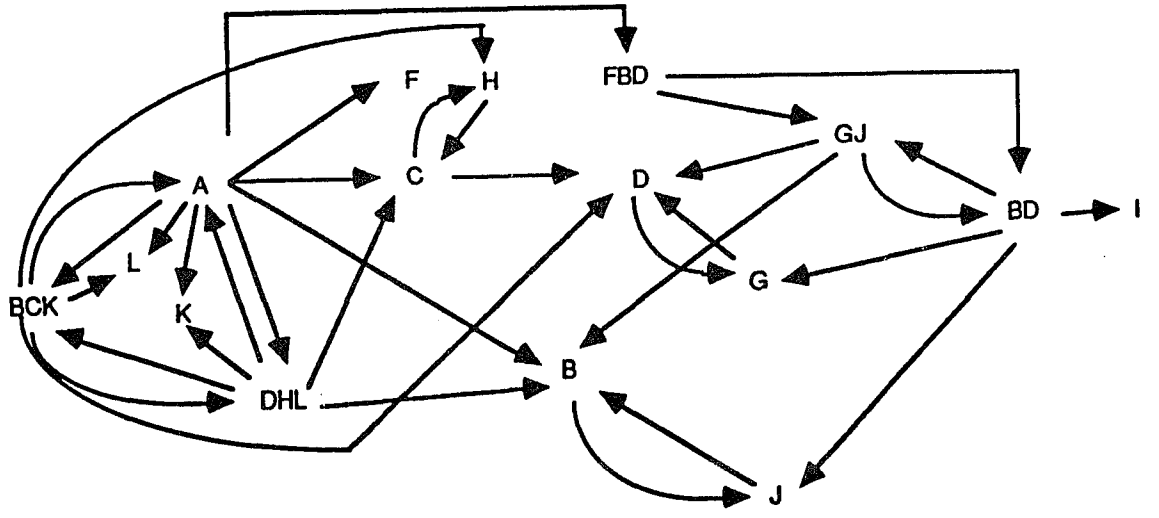

Figure 6.5(a): The graph H_1 formed in Step 1 of the algorithm Minimize. The arcs to the compound nodes have been added after the closure of the original graph is determined. The arcs from a compound node to its simple components have been removed to lessen the cluttering of the Figure.

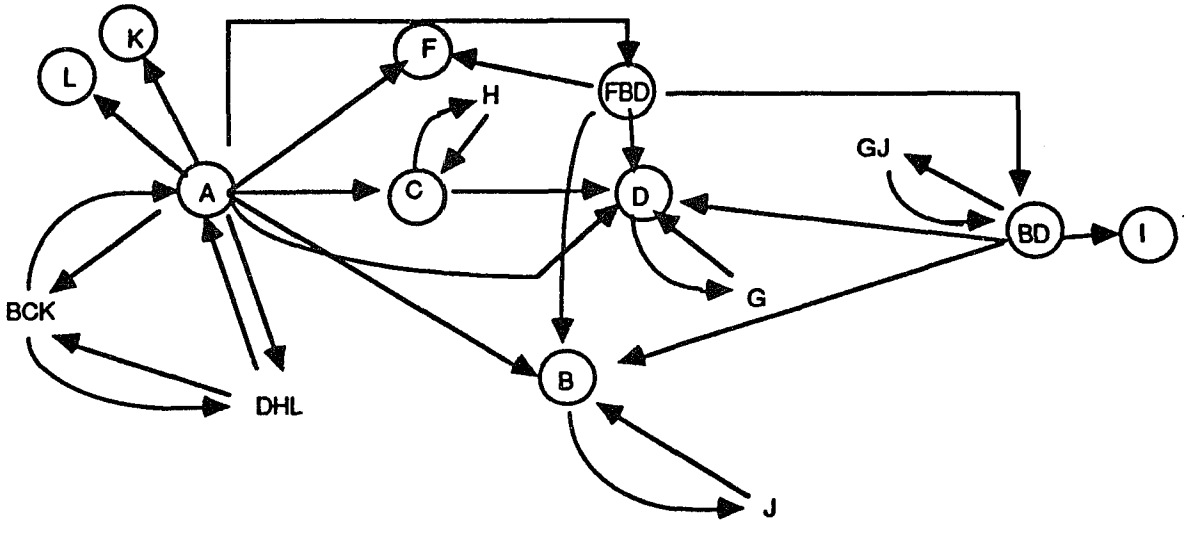

Figure 6.5(b): The graph H_1 after the execution of steps 1-6 of the algorithm Minimize on the graph in Figure 6.5(a). The strong components can be clearly seen and the representatives are marked with circles around them. In Step 7, node *GJ* would be determined to be textually contained in *BD* and would be removed.

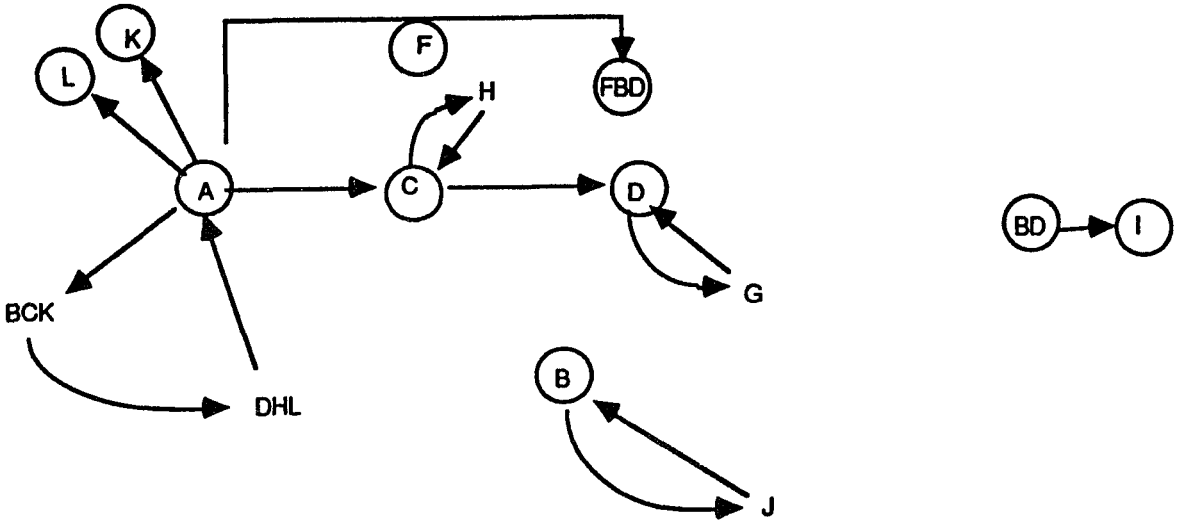

Figure 6.5(c): The graph H_1 at the end of Step 13. of the algorithm Minimize.

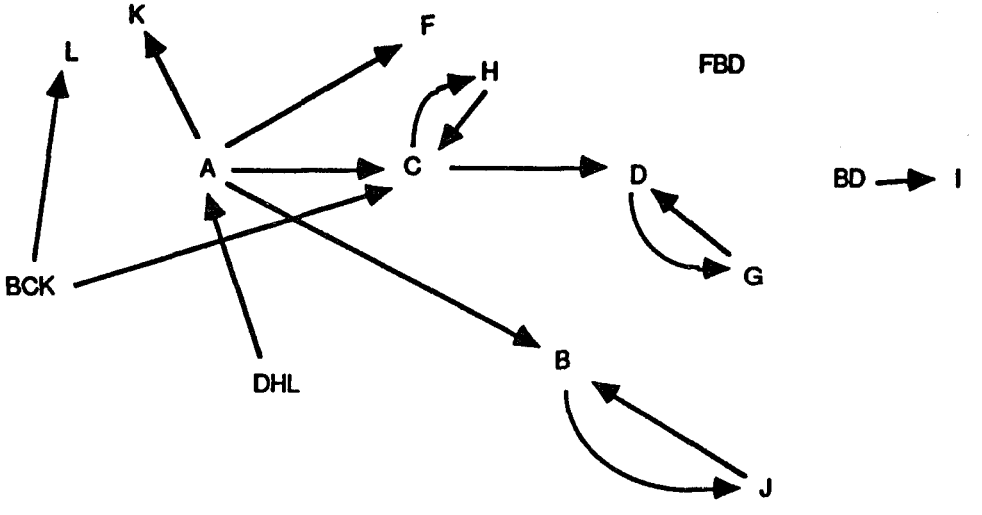

Figure 6.5(d): The minimum directed hypergraph of the graph H_1 in Figure 6.5(a) after the completion of the algorithm Minimize.

Lemma 6.4.1: Given directed hypergraph H the algorithm Minimize correctly obtains the minimum directed hypergraph H_m.

Proof: We will first show that the graph H_m does not contain any redundant arcs.

Case (i): Let (i, j) be the redundant arc and let i and j be in different strong components determined in Step 2. Since (i, j) is redundant there can exist a node k such that there are arcs (i, k) and (k, j). If node k is in a different strong component, then the transitive reduction of the acyclic graph formed by the strong components would delete the arc (i, j) in Step 9. If node k is in the component of i or j, then Steps 3-6 would delete the arc (i, j). Also the arc (i, j) is redundant if node j is textually contained in node i and it is removed in step 8. Hence the arc (i, j) is not redundant.

Case (ii): Let the arc (i, j) be redundant and let nodes i and j be in the same strong component determined in Step 2. Since (i, j) is redundant there can exist a node k, and arcs (i, k) and (k, j). Node k cannot be in a different component, since all nodes reachable by the node i are reachable by k using arcs (i, k) and (k, j). If node k is in the same component as nodes i and j, then the Hamiltonian circuit formed for each strong-component would delete arc (i, j) in Step 12. The arc (i, j) is redundant if node j is textually contained in node i. In Step 7 node j is removed. Hence the arc (i, j) is not redundant.

Case (iii): The construction of an Hamiltonian circuit in Step 12, creates redundant arcs as shown in Fig. 4.2(a). A transitive reduction on the entire graph H_1 in Step 13,

removes redundant arcs.

Case (iv): The arc (i, j), where j is a compound node; is redundant when there are arcs (i, j_1), ..., (i, j_l), where j_1, ..., j_l are *some* or *all* of the components of compound node j and each arc (i, j_k), where $1 \leq k \leq l$, is redundant. In Steps 14-18 the arc (i, j) is removed and arcs (i, j_1), ..., (i, j_r) are added, where j_1, ..., j_r are *all* of the components of j. The resulting graph is reduced using transitive reduction.

From the above we can clearly infer that H_m is non-redundant. It is minimal since in Step 10 the implied extraneous attributes are removed from each compound node and there are no arcs to compound nodes. Also, for two equivalent nodes X and Y there is only one arc from the component containing both X and Y and hence H_m is a minimum one.■

The presence of the arcs to compound nodes helps to treat the graph H as a directed digraph which makes the application of parallel transitive closure algorithms possible. Given H_1 all the steps in the algorithm except for the Steps 7,8 and 10 can be implemented using parallel transitive closure algorithms in $O(M(n))$ time with $O(n + e)$ processors on a CREW-PRAM.

The implementations of Steps 7, 8, and 10 are given in the following. We would first present a method to determine textual containment of two nodes in H_1. We will

assume that Steps 1-6 have been executed on the graph H_1. We will show how to test the textual containment of two nodes in the same strong component first. We construct the following data structure for the graph H_1. For each compound node J, let $j_1, j_2, ..., j_J$ be the nodes such that each $j_p \subset J$. The set $S(J)$ consists of ordered pairs (j_c, c), where j_c is the union of all j_p's in strong component c. Keep all the $S(J)$ sets sorted first on the component number containing J and next the rank compound node J within the component. For each ordered pair (j_c, c) in $S(J)$ we have a list of pointers $L(j_c, c)$. Each pointer $l_{jc} \in L(j_c, c)$ points to an ordered pair in the set $S(K)$ and K is in the same component as J with rank(K) > rank(J). In order to check the textual containment of node J in node K collect the pointers $l_{j1}, l_{j2}, ..., l_{jr}$, where each l_{ji} points to an ordered pair in $S(K)$. Now if $j_1 \cup j_2 \cup ... \cup j_r \supseteq J$, then J is textually contained in node K.

The sum of the sizes of the $S(J)$'s and the pointers in $L(j_c, c)$'s are both at most $O(n+e)$. The $S(J)$ sets can be ordered as required above in time $O(logn)$ time using $O(n)$ processors on a CREW-PRAM. We will show that the $S(J)$ sets can be constructed in $O(logn)$ time using $O(n+e)$ processors. The sets $S(J)$'s can be constructed by sorting the j_i's $\subset J$ based on the strong component number they are in and then taking a union of j_i's which are in the same component c to form the ordered pair (j_c, c). The sorting and union operation using recursive doubling techniques can

be done in $O(logn)$ time with $O(n+e)$ processors on a CREW-PRAM. Note that the j_i's $\subset J$ can be obtained during the construction of the graph H_1. The collecting of pointers which point to the ordered pair in $S(K)$ can all be done by sorting the pointers based on the list they point to and the test of textual containment can all be done in $O(logn)$ total time for all compound nodes in the graph H_1 using $O(n+e)$ processors on a CREW-PRAM.

Step 8 of the algorithm can be implemented as described above. The most time consuming step in the entire algorithm is Step 10. Removal of implied extraneous attributes can be compared with the problem of finding the minimal key. In fact, given a minimum directed hypergraph H_m, the minimal key is the union of the representatives contained in the strong components whose indegree is zero. The problem of left-reducing a dependency $X \rightarrow Y$ is finding a minimal key in the graph $H_m(X)$, where $H_m(X)$ is the graph induced by nodes $Z \subset X$ in H_m. Now, the removal of implied extraneous attributes from a compound node X is done by finding all strong components containing a node $Z \subset X$ and assuming all such Z's do not have implied extraneous attributes we pick Z's from components which do not have an incoming arc. The union of all such Z's gives the node X without any implied extraneous attributes. The time taken to do this is dependent on the diameter of the graph H_1 and can be easily done in $O(logn)$ time for a constant diameter graph H_1 with $O(n+e)$ processors on a CREW-PRAM. nodes.

Theorem 6.5: A minimum directed graph H_m of the directed hypergraph H_Σ for a given set of functional dependencies Σ can be obtained using algorithm Minimize in $O(\log^2 n + MAX(degree(H_\Sigma), diameter(H_\Sigma))*logn)$ using $O(M(n))$ processors on a CREW-PRAM.

Proof: The correctness of the algorithm Minimize follows from Lemma 6.4.1. Step 1 takes $O(\log^2 n + MAX(degree(H_\Sigma), diameter(H_\Sigma))*logn)$ using $O(M(n))$ processors (Theorem 6.4). Steps 7 and 8 take $O(logn)$ time and uses $O(n+e)$ processors from the above discussion. Also, Step 10 can be implemented in $O(diameter(H_\Sigma).logn)$ with $O(n+e)$ processors as discussed above. All other steps can be done in $O(\log^2 n)$ time using $O(M(n))$ processors, since they all use transitive closure algorithms as a subroutine. All the steps require the CREW-PRAM model.■

From Theorem 6.5 we can see that for fixed degree and diameter graphs H_Σ the complexity of the algorithm Minimize is $O(\log^2 n)$ and uses $O(M(n))$ processors. Hence it is optimal based on the transitive closure bottleneck phenomenon.

6. CONCLUSION

Parallel algorithms for the manipulation of directed hypergraphs were presented in the chapter. It was first shown that manipulation of directed hypergraphs in inherently sequential. A parallel algorithm for the directed hypergraph reachability

problem is presented. Algorithms for finding the closure and determining the minimum equivalent directed hypergraph were also presented. All algorithms are shown suffer from the transitive closure bottleneck phenomenon. The algorithms were discussed in the context of manipulation of functional dependencies in relational databases.

Chapter Seven

PARALLEL ALGORITHMS FOR MULTI-DIMENSIONAL RANGE SEARCH

1. INTRODUCTION

In this chapter we present a parallel algorithm to obtain a set of points in a rectangular parallelepiped (range-search) in $O(logn)$ time, with only $(2.\log^2 n - 10.logn + 14)$, on an EREW-PRAM, where processors are allowed to communicate through messages. We also present a non-trivial implementation technique on the hypercube parallel architecture with which the above time and processor bound can be achieved without any communication overhead. A parallel algorithm for range searching is developed here using the concept of distributed data structures. We use the range tree proposed by Bentley [12] as our data structure to be distributed. Our algorithm can easily be generalized for the case of d-dimensional range search. Range search has important applications in the areas of databases and computational geometry. The results presented in this chapter appear in [Radhakrishnan, Iyengar, Subbiah [84]].

2. RANGE SEARCH

Let S be a set of n d-dimensional points in R^d. A range query q is a d-range which is the cartesian product of d intervals. The output of the query is all points in S

that lie within q. In the case of two dimensions the 2-range is a rectangle and for more than three dimensions the d-range is a hyperrectangle. Thus, the answer to the query q is a set of all points in S that are inside the rectangle or hyperrectangle as the case may be. Range search has several applications including databases and computational geometry [77]. The range search is equivalent to database *selection* operator on a relation.

A considerable amount of work has been devoted to the range search problem [13, 54]. Bentley [12] gives a thorough overview of various multi-dimensional and range searching problems. Several data structures and algorithms for range searching have been proposed and each has trade-offs between storage and time complexity. These structures include k-D-Tree, multidimensional trie, super-B-tree etc. Bentley and Maurer [14] have shown the lower bound on the time complexity of range search on a set of n d-dimensional data to be $(d.logn)$. With the overlapping-ranges data structure [14] the time bound of $O(d.logn)$ can be obtained at the expense of very high storage cost which is $O(n^d)$. Most practical algorithms use a storage cost of $O(n\log^{d-1}n)$ to obtain a time bound of $O(\log^{d-1}n)$ [77]. Layered Range tree data structure [77] a variant of range tree has the above storage and time complexity; a reduction of $O(logn)$ factor in storage and time complexity of the range tree. Chazelle [23] using the concept of filtering search reduced the storage cost to $O(n.\log^{d-1}n/loglogn)$ while retaining the time complexity.

Recently, there has been a growing interest in developing parallel algorithms for problems in databases and computational geometry. This interest has been enhanced due to the availability of more feasible parallel architectures like the hypercube and the mesh of processors. Baru and Frieder [10] have developed novel algorithms for the execution of relational database operations on a hypercube parallel machine. Algorithms for the execution of the relational *join* operator on the hypercube machine were also given by Omiecinski and Tien [74]. A number of parallel algorithms for computational geometry problems can be found in [7, 29, 69, 55, 89].

More recently, Katz and Volper [58] developed a parallel algorithm for retrieving the sum of values in a region on a two dimensional grid in $O(logn)$ time with $O(n^{1/3})$ processors. In this chapter, we present a parallel algorithm for the range search problem using the range-tree as our data structure. In particular, we show that the 2-dimensional and 3-dimensional range search can be effected in $O(logn)$ time with $3/2.logn - 1$ $(2.\log^2 n - 10.logn + 14)$ processors respectively, on an EREW-PRAM. The retrieval of the sum of the values can also be done in the above time and processor bounds.

One of the keys to efficient parallel searching is the distribution of the data points to be searched. To achieve such an efficiency we use the concept of *distributed data structures*. By *distributed* data structures we mean a typically large data structure,

such as a B-Tree, K-d tree, Range tree and others, that is logically a single entity but that has been distributed over several independent processor stores. This concept is not new and frequently arises in the area of distributed data bases. Ellis [35] developed a distributed version of Extendible Hashing for database searching. Distributed data structures of scientific calculation and processing of sets were introduced in [86, 72] respectively. One of the fundamental advantages of the concept of distributed data structure is that processors are assigned to data statically and overheads due to dynamic allocation are avoided. Also the concept of parallel processing of a single data structure have occurred in other forms such as concurrent access to a data structure and issues relating to concurrency control [65].

The parallel model of computation used in this chapter is the EREW-PRAM (Exclusive-Read-Exclusive-Write Parallel Random Access Model). Here no two processors are to simultaneously allowed to read or write to the same memory location [56]. Processors communicate through messages and it takes unit time to send/receive a message from/to an adjacent processor. Also in one unit of time a RAM instruction can be executed by a processor. For example, it takes $O(logn)$ time for a single processor to search a sorted list of n elements and the in the same time broadcasting of a message on a n leaf binary-tree of processors can be completed.

3. THE RANGE TREE DATA STRUCTURE

The range tree was first introduced by Bentley after which several variants were proposed. We will first introduce the 2-dimensional range tree. The generalization to d dimensions can be easily visualized. Let S be the set of n 2-dimensional points. First sort the n points based on the value of the x-coordinate. Imagine each point p as an interval $[x_i, x_i]$, where the first and second components are $B[p]$ (begin point) and $E[p]$ (end point). Now, the range-tree corresponding to the first dimension is a rooted binary-tree whose leaves contain the n points sorted and placed from left to right as intervals. An interior node v and its left (v_1) and right (v_2) children has an associated interval with $B[v] = B[v_1]$ and $E[v] = E[v_2]$. Now the second dimensional coordinates i.e., the y-coordinates are stored in the tree as follows. For each interval $l = (B[v], E[v])$ belonging to the node v in the tree, the y-coordinates of the points which project onto the interval l are stored as a binary-tree and the node v points to the root of the binary tree. Figures 7.2.a and 7.2.b show a set of points in the plane and its corresponding range tree, respectively.

X	1	2	3	4	5	6	7	8	...	16
Y	9	13	12	17	14	6	10	16	...	2

Figure 7.2.a - A set of points in the plane.

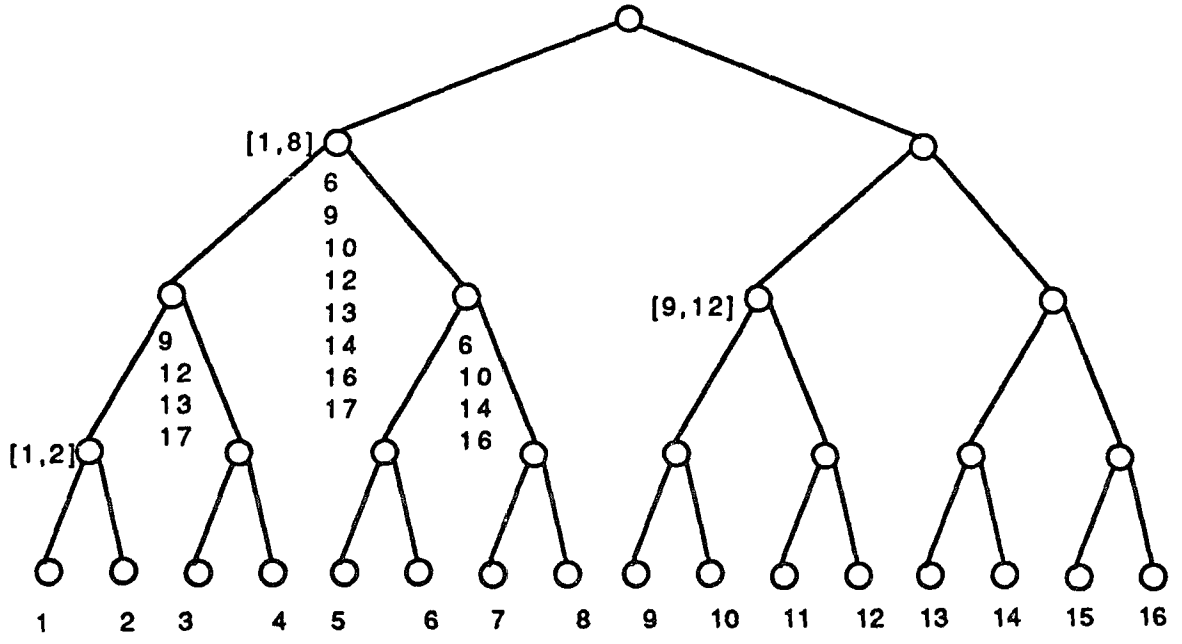

Figure 7.2.b - The range tree corresponding to points in Figure 7.2.a.

For the case where each point in the plane represents a value and the range query is to sum the values in a specified region we need to store the values S_v at each node v of the range tree as follows. Let t_v be the binary tree corresponding to the y-coordinates at node v. Let ${t_v}^b$ and ${t_v}^e$ represent the left-most and the right-most leaves of the binary tree t_v. The value S_v stored at node v is the sum of the values of the points whose x-coordinates and y-coordinates lie in the interval $(B[v], E[v])$ and $({t_v}^b, {t_v}^e)$, respectively. It is not difficult to show that the d-dimensional range tree can be constructed in parallel in $O(d.logn)$ time using $O(n)$ processors by the use of Cole's [24] parallel sorting algorithm on a EREW-PRAM.

We present some properties of the range tree from [77].

Proposition 7.2.1: The number of nodes selected in the range tree during the range search on any dimension is at most $2.logn - 2$ and there are not more than two nodes selected from each level of the range tree.■

Proposition 7.2.2: Range searching of an n-point d-dimensional file can be effected by an algorithm in time $O((log\ n)^d)$ using the range-tree technique.■

4. RANGE TREE DISTRIBUTION AND PARALLEL ALGORITHM

The key to the success of any parallel algorithm for range searching is determined by the type of data distribution. With $O(n)$ processors effective searches can be made, but, having such large number of processors is highly impractical. In this section, with range tree as the data structure, we present a simple data distribution scheme with which $O(logn)$ search time using $(2.\log^2 n - 10.logn + 14)$ processors is effected for the case of 3-dimensional data points. The technique we describe can easily be extended to the case of d-dimensions. We will assume that the root and the leaves of the tree are at heights h $(n = 2^h)$ and 1, respectively. We will assume that data values in each dimension are unique (in case some data values are the same, we can perturbate them slightly and, this commonly done (see [55]).

3.1 Estimation of processor and time-complexity

We now estimate the number of processors required to search in parallel for the case when $d = 2$ and $d = 3$.

In Proposition 7.2.1 we note that at most $2.logn - 2$ range tree nodes are selected for any range query on a single dimension. This tells us that with $2.logn - 2$ processors we can search the next dimension in parallel. Now, the time-complexity is given by the following simple equation:

$$Q(1, n) = O(logn)$$

$$Q(2, n) = Q(1, n) + O(logn) = O(logn)$$

Here $Q(1, n)$ is the time taken to search the range tree in dimension 1. Let us say that another $2.logn - 2$ processors are available at each of the selected nodes during the processing of the dimension i. The next dimension $i+1$ can also be processed in parallel. Generalizing this scheme to d-dimensions we can see that the time-complexity is now given by the equation:

$$Q(1, n) = O(logn)$$

$$Q(d, n) = Q(d-1, n) + O(logn) = O(d.logn)$$

The total number of processors ($P(d, n)$) required to search a range tree storing n d-dimensional points and achieve the above time-complexity is given by the equation:

$$P(1, n) = 1$$

$$P(2, n) = 2logn - 2$$

$$P(d, n) = P(d-1, n)\,(2logn - 2) = O(\log^{d-1} n)$$

A simple observation that at most 2 nodes are selected at each of the heights from $h - 2$ to 1 (Proposition 7.2.1) helps to reduce the above loose processor bound to a great extent. The number of data points belonging to dimension i stored at node v at height r in the $i - 1$-dimension range tree is 2^r. Let $t(v)$ be the range tree corresponding to these points. The number of processors required to do a parallel search on i+1-dimensional points stored in $t(v)$ is $2.log(2^r) - 2$. We now present the estimation on the number of processors for $d = 3$. From arguments above we have,

$$P(3, n) = 2.[2.log(2^{h-2}) - 2 + 2.log(2^{h-3}) - 2 + \cdots + 2.log(2^{h-(h-1)}) - 2]$$

$$P(3, n) = 2.\log^2 n - 10.logn + 12$$

We need two more processors to search the two leaves that will be selected during the search of the tree corresponding to dimension 2. Hence the total number of processors needed to search the the 3-dimensional range tree is $2.\log^2 n - 10.logn + 14$. In the above processor estimation we have not included processors needed to search a tree

stored in the node v at height $h-1$. It is not necessary to have additional processors for node v, since if node v is selected during the search none of the nodes in the subtree rooted at v will be selected. Note that with at most $2.logn-6$ processors the node v can be processed. There are $\log^2 n - 5.logn + 7$ processors assigned to the nodes of the subtree rooted at v and they are sufficient to process the tree belonging to node v.

In the case, where there are fixed number of processors, say P, d-dimensional range search can be effected in $O(\log^d n/P)$ time. This bound is obtained as follows. During the search of the i-dimensional range tree the selected nodes are allocated among the P processors. For example, if $O(logn)$ nodes are selected during the processing of dimension 2, each processor is allocated $O(logn)/P$ nodes to be searched. In the range tree a search on d-dimensions will select at most $O(\log^d n)$ nodes (analysis is similar to the one presented for $P(d, n)$), hence, the above bound.

3.2 Distribution of data among processors

In the case of shared memory model data contained in the range-tree need not be distributed among processors and idle processors are allocated dynamically to the selected nodes of the tree. The dynamic assignment of processors to nodes is an overhead to the system as it has to maintain a list of idle processors. Assuming that the selected nodes during the processing of dimension i is I, the time taken to assign idle

processors to the selected nodes is $O(I)$. Other obvious benefits of data distribution which include recovery and data reconstruction motivate the need for static assignment of processors to the nodes of the tree. In the previous subsection we have determined the upper bound on the number of processors required to do a parallel range search in $O(logn)$ time for the case of 2- and 3-dimensional data points. We now show how the processors are actually assigned and give the search strategy for the above cases. The case of d-dimensional is a natural extension of the approach presented here.

We will call an assignment of processors to the nodes of the range-tree *proper* if the number of processors used in the assignment is less than or equal to the number of processors estimated in Section 3.1 to achieve a time-complexity of $O(d.logn)$, for a range search in d-dimensions. We will now present a proper assignment scheme for the case of 2-dimensions first. Let T be a 1-dimensional range tree of height h. Starting with the leaves at height 1 to height $h-3$, we will allocate 2 processors to each of the heights, since at most two nodes are selected from each of those heights by Proposition 7.2.1. If processors p_i and p_j are allocated at height r $(1 \leq r \leq h-3)$, then starting from the left assign nodes at height r p_i and p_j, alternatively. This assignment would guarantee that the two selected nodes would be in different processors. Now, let p_i and p_j be allocated to height $h-2$. The first and the second pairs of nodes at height $h-2$ from the left are assigned p_i and p_j, respectively. The two nodes at

height $h-1$ are assigned the same processor that are assigned to their children. The root of T is assigned any of the processor assigned to its immediate child at height $h-1$. The total number of processors used in the assignment is $(2.logn-2)$. To search the third dimension, the tree corresponding to a node v is assigned new set of processors the same way as described in the case of 1-dimensional range tree. For two nodes v_1 and v_2, their trees are assigned with the same set of processors if the processor assigned to v_1 is the same as the processor assigned to v_2. Thus, the above assignment scheme uses exactly the same number of processors as estimated in Section 3.1. Figure 7.3.a gives the assignment of processors for the tree in Figure 7.2.b.

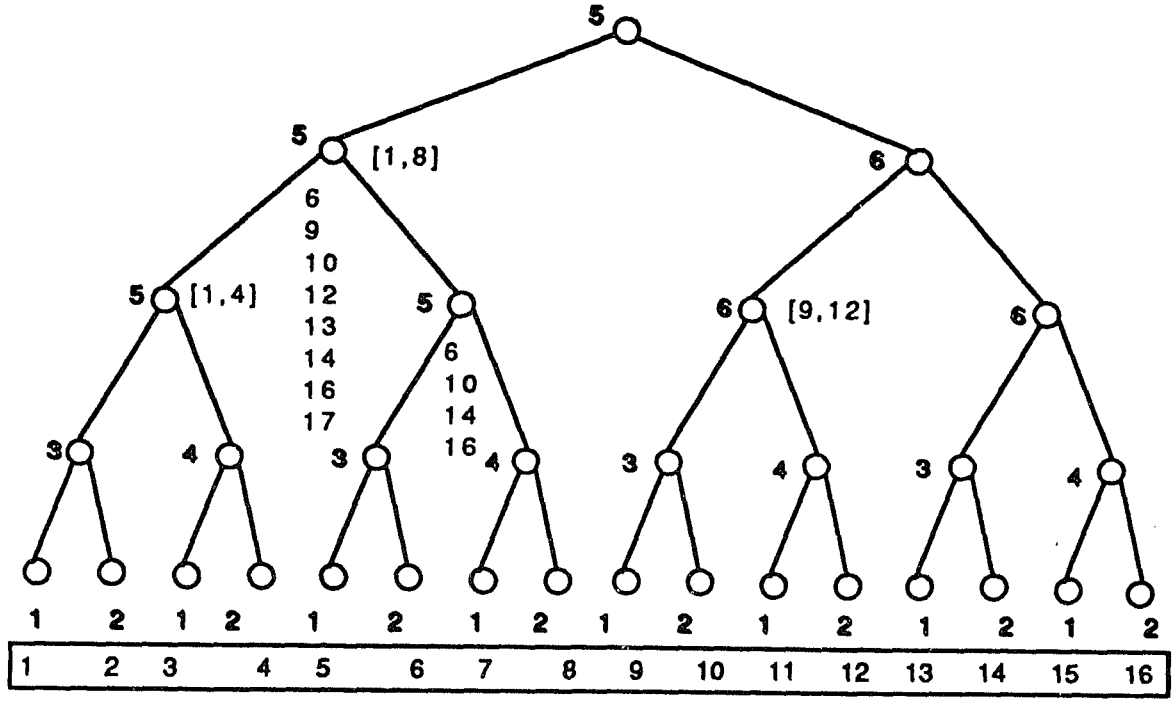

Figure 7.3.a - A proper assignment of processors for the tree in Figure 7.2.b.

The search strategy is very simple. Each processor assigned to a node v at height r is responsible for giving the search message to the appropriate processor at height $r-1$. The search message is sent from a processor v at height r to a processor at height $r-1$ if the query interval does not completely contain the interval $(B[v], E[v])$. If the interval containment is satisfied no more search message is issued from v and the tree at v is searched next. We know that each processor is assigned more than one tree node. The node interval to be chosen for comparison with the query interval and the processor to which the search message has to be sent are all done by the processor with the help of simple array indexes. We skip the details here.

Theorem 7.1: The range search on a 2-dimensional and 3-dimensional sets of n points can be done in $O(logn)$ time with $(2.logn - 2)$ and $(2.\log^2 n - 10.logn + 14)$ processors, respectively. The sum of values in the range can also be done for the case of 2-dimension and 3-dimension in $O(logn)$ time with the above processor bounds.

Proof: The sum of values in a range can be retrieved using the values S_v stored at each node of the range tree (see Section 2.). The rest of the result follows from the discussion above.■

5. RANGE SEARCHING ON THE HYPERCUBE MACHINE.

We will now proceed to give details on how the nodes of the tree can be mapped on to the hypercube for efficient searching. First we will present the 2-dimensional case. A good mapping is one which minimizes the communication time in the hypercube. Consider the processor assignment discussed in the previous section. A mapping which takes the ith processor and maps it to the ith hypercube node would require a total communication time of $O(logn.loglogn)$, since we require $O(logn)$ processors for range searching in 2-dimensions and $O(logn)$ is the height of the range-tree corresponding to the first dimension. Hence the total search time for range-searching in two dimensions using a range-tree on a hypercube would be $O(logn.loglogn)$. We now present a mapping which would reduce the total search time to $O(logn)$ on the hypercube.

Consider the assignment of processors as discussed in the previous section to the nodes of the range tree corresponding to dimension 1. A processor at height r (p_i) after checking its range will send the search message to another processor at height $r-1$ (p_j). If p_i and p_j are adjacent to each other in the hypercube the communication time is a constant, otherwise, it can be as high as $O(loglogn)$ the diameter of the hypercube. We now present a mapping (embedding) technique which gives constant-time communication time between processors in adjacent levels of the range tree.

It can be seen that a processor p_i at height r is adjacent to two processors at heights $r-1$ and $r+1$. Based on the processor assignment discussed earlier and the adjacency relationship between processors we form a graph G called the *processor assignment graph.*

A processor assignment graph G consists of $(2.logn - 2)$ nodes and is connected as follows. The graph G consists of 4 chains c_1, c_2, c_3, and c_4. The chains c_1 and c_2 contains odd and even numbered processors respectively (will be referred as odd and even numbered nodes). Two odd or even numbered nodes are adjacent in their respective chains iff they belong to adjacent levels of the range tree. The chain c_3 (c_4) formed when an edge is drawn from every node a in c_1 (c_2) to every node b in c_2 (c_1), whenever a and b are in adjacent tree levels (see Figure 7.4.a).

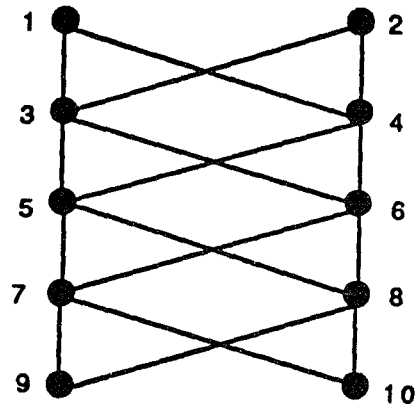

Figure 7.4.a - A processor assignment graph G.

We will show in Proposition 7.4.1 that the graph G cannot be embedded in the hypercube with dilation 1 (i.e., all adjacent nodes in G will not be adjacent when embedded in the hypercube). For dilation two embedding we require that the dimension of the cube be $O(logn)$, i.e., with a cube containing $2^{O(logn)}$ nodes. In this case the expansion, (i.e., the ratio of the number of hypercube nodes to the number of graph nodes) is exponential.

Proposition 7.4.1: The processor assignment graph G cannot be embe
hypercube with dilation one.

Proof: In a hypercube 2 nodes a and b can be adjacent at most to the
two nodes c and d. In G two nodes a and b can be adjacent to t
nodes. This implies either a should be adjacent to b or vice versa.
is 2.■

Lemma 7.4.2: The processor assignment graph G can be embedded on to an hypercube with dilation 3 and expansion 1.

Proof: Let $k = 2.logn - 2$ be the number of nodes of the graph G. We form a grid R containing 2 rows and $logn$ columns as follows. Let $R(i,j)$ represent the ith row and the jth column of the grid R. The embedding of the graph G on the grid R is done as follows:

```
1. R(1,1) := 2; (* node numbered 2 in G)
2. R(1,2) := 4;
3. R(2,2) := 1;
4. R(1,3) := 3;
5. j := 5;
6. For i := 4 To logn Do
       Begin
7.        R(1,i) := j;
8.        R(2,i) := j+1;
9.        j := j + 2;
       End;
```

Figure 7.4.b is the grid obtained for the processor assignment graph shown in Figure 7.4.a. The mapping from graph G to grid R is done with dilation 3 which can be easily verified. Now, the $2 \times logn$ grid can be embedded on to the hypercube with dilation 1 and expansion 1 [22], hence the Lemma. ■

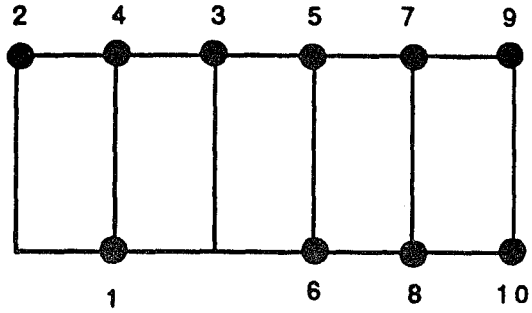

Figure 7.4.b - The embedding of the graph G in Figure 7.4.a onto a grid.

Theorem 7.2: Theorem 7.1. holds in the case when the processors are arranged as an hypercube architecture.

Proof: From Lemma 7.4.2 it is clear that the communication time for the search message to travel from one level to the adjacent one in the range tree is a constant. In the case of 2-dimensions after embedding G in a hypercube with $(2.logn - 2)$ nodes, we can easily see that the range search can be done in $O(logn)$ time. In the case of 3-dimensional range search the processor bound can be achieved with several hypercube machines of different sizes as follows. With the availability of two cubes of size $(2.logn - 6)$, two cubes of size $(2.logn - 8)$ and so on, the 3-dimensional range search can be done using a total of $(2.\log^2 n - 10.logn + 14)$ processors. This is done by embedding each of the subtrees optimally onto their respective hypercubes.■

It can also be easily shown that using a single hypercube with $O(\log^2 n)$ processors a 3-dimensional range search can be completed in $O(logn)$ time.

6. PROCESSOR REDUCTION

In this section we will show that $(3/2.logn - 1)$ processors are sufficient to effect a range search in $O(logn)$ time for a set of points in the plane and thus saving $((logn)/2 - 1)$ processors. The approach can be generalized to d-dimensions easily. The processor reduction is illustrated in the following example. Let T_{256} be a range tree with 256 leaves. Time taken by a single processor to process the tree T_{256} is in the worst case 8 units of time (i.e., time taken to perform the range search on the range tree with 256 leaves). Two T_{16} trees can be processed sequentially by a single processor in 8 units of time. This means that with two processors, the tree T_{256}, and two T_{16} trees can be processed in 8 units of time instead of using three processors and still requiring 8 units of time. We will generalize the above idea and estimate for a two dimensional range tree of height h. The range tree $T_{2^{h-2}}$ requires $h-2$ units of time. Since there are two such trees at height $h-2$, we will allocate two processors. For similar reasons we have to allocate two processors for each of the heights from $h-2$ to $(h-2)/2 - 1$. For heights from $(h-2)/2$ to 1 we allocate a single processor. The total number of processors allocated to the entire tree is now $(3/2.logn - 1)$. Finding a proper processor assignment scheme with reduced number of processors is easy.

7. CONCLUSION

The problem of range search was solved in parallel using the range tree data structure. The nodes of the range tree were distributed among the processors in such a way that the search can be carried efficiently in parallel. It can be easily shown that our our algorithm is optimal for the chosen data structure in the case of arbitrary dimension $d = O(1)$, from Proposition 7.1.1. Based on the assignment of processors to the nodes of the range tree a processor assignment graph was created. The processor assignment graph was embedded onto an optimal hypercube for the execution of the range search without any communication overhead. Finally, a processor reduction argument was presented.

Chapter Eight

CONCLUSION

In the foregoing chapters we have developed fast parallel algorithms on strongly chordal graphs, directed hypergraphs, and parallel algorithms for performing multi-dimensional range search. The graph structures examined in this thesis have wide range of applications which includes databases and computer networks.

In chapter 2, new characterizations of strongly chordal, ptolemaic, and block graphs in terms of the intersection graph of the maximal cliques of the graphs was developed. These characterizations paved way for the development of efficient parallel recognition algorithms. Note that the above graphs have been shown to have close relationships with the theory of relational databases. A parallel algorithm for determining the strongly perfect vertex elimination scheme was also developed in chapter 2. Methods to construct strongly chordal, ptolemaic, and block graphs from arbitrary graphs by adding a minimal set of edges is presented in chapter 3. This result is significant, since it means that acyclic relational database schemes can now constructed very fast in parallel by the addition of a minimal number of attributes to each scheme of the cyclic schemes. Parallel algorithms for various domination problems on strongly chordal graphs was developed in chapter 4. Note that domination problems are NP-complete even on chordal graphs. The algorithms require linear

sequential time on strongly chordal graphs. The algorithm for dominating set produces a dominating set which is connected and hence a total dominating set.

It is known that if a problem X is NP-complete, then there exists an NC algorithm for X if and only if $P = NP$. In chapter 5, we prove several impossibility results with respect to designing NC algorithms for query implication testing which has several applications in relational databases and distributed processing. We show that for a set of queries which satisfies "chordality" conditions, polynomial time sequential algorithms for implication testing exists.

In chapter 6, we presented efficient parallel algorithms for manipulating directed hypergraphs. An important application of directed hypergraph is in the area of relational databases where it is used for representing functional dependencies. A simple problem such as finding a directed path in the directed hypergraph is shown to be P-complete. This means NC algorithms for manipulating functional dependencies exists if and only if $P = NP$. But, by careful analysis of the structure of the directed hypergraph we presented NC algorithms for the manipulation of the directed hypergraph if the degree and diameter of the graph is fixed.

Fast parallel algorithms for performing multi-dimensional range search was presented in chapter 7. A non-trivial implementation technique on the hypercube

parallel architecture was also presented in chapter 7. This result is significant, since methods described in this chapter can be extended for performing parallel manipulations on many other multi-dimensional structures which have applications in robotics, CAD and other areas.

We now examine some issues and problems which have arisen out of our study. A lot of these problems are probably new.

1. Using the characterization provided for strongly chordal, ptolemaic, and block graphs in Chapter 2, efficient (linear?) sequential recognition algorithms can be developed. Anstee and Farber [4] provide an $O(n^8)$ time sequential algorithm for the recognition of strongly chordal graphs. This time bound can be easily reduced.

2. Efficient sequential algorithm for determining the strongly perfect vertex elimination ordering can be developed and this would be quite useful, since no algorithm to obtain SPEO of a strongly chordal graph in less than $O(n^3)$ is known in the literature. Note that any time complexity which is less than $O(n^3)$ would imply that the domination problems on strongly chordal graphs can be solved efficiently.

3. It was shown by Yannakakis [98] that minimum fill-in (i.e., minimum number of edges needed to make an arbitrary graph chordal) is NP-complete. We have presented algorithms for minimal fill-in of a strongly chordal graph, ptolemaic graph, and block

graph. It would be interesting to find out whether minimum fill-in of these graphs are NP-complete.

4. Several domination problems on strongly chordal graphs were solved in Chapter 4. There are several other domination problems which are left open. No linear-time weighted 1-domination problems are known for strongly chordal graphs. Parallel algorithms for them are also not known. There is a relationship between steiner tree problem and connected domination in a strongly chordal graph [97]. Algorithms to construct a steiner tree for a strongly chordal graph is also not known.

5. There are well-known relationship between chordal graph and its subclasses and relational databases. It would be interesting to examine the parallel algorithms in terms of acyclic relational databases, for example construction of conflict-free multi-valued dependencies given a set of acyclic schemas in parallel [36].

6. On-line algorithms for the manipulation of directed hypergraphs would be interesting, where arcs and nodes are added dynamically.

7. Parallel algorithms for manipulation of data structures is an important area of research. It would be interesting to develop fast parallel algorithms to manipulate several multi-dimensional data structures which have applications in CAD and Robotics. The parallel algorithms developed in Chapter 7 can be extended to handle more

relational database operations thereby developing a parallel database environment.

References

References

[1] ABITEBOUL, S., KANELLAKIS, P., AND GRAHNE, G., "On the Representation and Quering of Sets of Possible Worlds," *Proc. ACM SIGMOD Internat. Conf. on Management of Data, San Fransisco*, pp. 34-48, May 27-29. 1987.

[2] ABITEBOUL S. AND VIANU V., "Equivalence and Optimization of Relational Transactions," *JACM*, vol. 35, no. 1, pp. 70-120, January 1988.

[3] ALESSANDRO, D'ATRI AND MARINA, MOSCARINI, "On Hypergraph Acyclicity and Graph Chordality," *IPL*, vol. 29, no. 1, November 1988.

[4] ANSTEE, R.P. AND FARBER, M., "Characterizations of Totally Balanced Matrices," *Journal of Algorithms*, vol. 5, no. 1, pp. 215-230, 1984.

[5] ARNBORG, S., "On the Complexity of Multivariable Query Evaluation," *FOA Rapport C20292-D8, National Defence Research Institute, Stockholm, Sweden*, 1979.

[6] ARNBORG, S., "Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposibility," *BIT*, vol. 25, pp. 2-33, 1985.

[7] ATALLAH, M.J., COLE, R., AND GOODRICH, M.T., "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *Siam J. Comput.*, May 1989.

[8] AUSIELLO, GIORGIO, ATRI, ALESSANDRO D', AND SACCA, DOMENICO, "Graph Algorithms for Functional Dependency Manipulation," *JACM*,

vol. 30, no. 4, pp. 752-766, October 1983.

[9] AUSIELLO, GIORGIO, ATRI, ALESSANDRO D', AND SACCA, DOMENICO, "Minimal Representation of Directed Hypergraphs," *SIAM J. COMPUT.*, vol. 15, no. 2, pp. 418-431, May 1986.

[10] BARU, C. K. AND FRIEDER, O., "Database operations on Cube-Connected Multicomputer System," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 920-927, June 1989.

[11] BEERI., CARTIEL, FAGIN., RONALD, MAIER., DAVID, AND YANNAKAKIS., MIHALIS, "On the Desirability of Acyclic Database Schemes," *JACM.*, vol. 30, no. 3, pp. 479-513, July 1983.

[12] BENTLEY, J.L., "Multidimensional Divide-and-Conquer," *CACM*, vol. 23, no. 4, pp. 214-229, April 1980.

[13] BENTLEY, J.L. AND FRIEDMAN, J.H., "Data Structures for Range Searching," *ACM Surveys*, vol. 11, no. 4, December 1979.

[14] BENTLEY, J.L. AND MAURER, H.A., "Efficient Worst-Case Data Structures for Range Searching," *Acta Informatica*, vol. 13, pp. 155-168, 1980.

[15] BERGE C., "Graphs and Hypergraphs," *North-Holland, Amsterdam*, 1976.

[16] BERTOSSI, A.A. AND MORETTI, S., "Parallel Algorithms On Circular-Arc Graphs," *IPL*, vol. 33, pp. 275-281, 1989-90.

[17] BODLAENDER H.L., "Dynamic Programming Algorithms on Graphs with Bounded Treewidth," *MIT-LCS-TR-394, MIT Laboratory for Computer*

Science, Cambridge MASS., 1987.

[18] CHANDRASEKHARAN, N. AND HEDETNIEMI, S.T., "Fast Parallel Algrorithms for Tree Decomposing and Parsing Partial-K-Trees," *In Proc. 26th Annual Allerton Conference on Communication, Control, and Computing, Urbana-Champaign, Illinois*, October 1988 .

[19] CHANDRASEKHARAN, N. AND IYENGAR, S.S., "NC Algorithms for Recognizing Chordal Graphs and k-trees," *Tech. Rept. #86-020, Dept. of Comp. Sci., Louisiana State University. Also in IEEE Transactions on Computers, June 1988.*

[20] CHANG, G.J., "Labeling Algorithms For Dominating Problems In Sun-Free Chordal Graphs," *Disc. Appl. Math.*, vol. 22, pp. 21-34, 1988/89.

[21] CHANG, G.J. AND NEMHAUSER, G.L., "The k-Domination And k-Stability Problems On Sun-Free Chordal Graphs," *SIAM J. Alg. Disc. Meth.*, vol. 5, pp. 332-345, 1984.

[22] CHAN, M.Y., "Dilation-2 Embeddings of Grids Into Hypercubes," *Int'l Conf. on Parallel Processing*, 1988.

[23] CHAZELLE, B., "Filtering Search: A New Approach To Query-Answering," *Siam J. Comput.*, vol. 15, no. 3, pp. 703-724, August 1986.

[24] COLE, R., "Parallel Merge Sort," *Siam J. Comput.*, vol. 17, no. 4, August 1988.

[25] CORNEIL, D.G. AND STEWART, L.K., "Dominating Sets In Perfect Graphs," *Submitted for Publication, University of Toronto*, 1986.

[26] DAHLHAUS, ELIAS AND KARPINSKI, MAREK, "On the Parallel Complexity of Matching for Chordal and Path Graphs," *Institut Fur Informatik Der Universitat Bonn, Report No: 8511-CS*, pp. 1-13, March 1987.

[27] DAHLHAUS, E. AND KARPINSKI, M., "An Efficient Parallel Algorithm for the Minimal Elimiantion Ordering (MEO) of an Arbitrary Graph," *FOCS*, 1989.

[28] DAHLHAUS, E. AND KARPINSKI, M., "Fast Parallel Computation of Perfect and Strongly Perfect Elimination Schemes," *Tech. Rept. 8519-CS, Institut fur Informatik, Universitat Bonn, 1987, and IBM Research RJ5901, October 1987.*

[29] DEHNE, F. AND STOJMENOVIC, I., "An $O(\sqrt{n})$ Time Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh-Of-Processors," *IPL*, vol. 28, 1988.

[30] DIEDERICH JIM. AND MILTON JACK., "New Methods and Fast Algorithms for Database Normalization," *ACM TODS*, vol. 13, no. 3, pp. 339-365, September 1988.

[31] DOWLING, W.F. AND GALLIER, J.H., "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae," *J. Logic Programming*, vol. 3, 1984.

[32] DUKE, RICHARD, "Types of Cycles in Hypergraphs," *Annals of Discrete Mathematics*, vol. 27, no. 1, pp. 399-418, 1985.

[33] ECKSTEIN, D.M., "Simultaneous Memory Access," *Dept. Comput. Sci.,*

Iowa State Univ., Iowa City, Tech. Rep. TR-79-6, 1979.

[34] EDENBRANDT, A., "Chordal Graph Recognition is in NC," *IPL*, vol. 24, pp. 239-241, 1987.

[35] ELLIS, C.S., "Distributed Data Strucutures: A Case Study," *Int'l Conf. on Parallel and Distributed Computing*, 1985.

[36] FAGIN., RONALD, "Degrees of Acyclicity of Hypergraphs and Relational Database Schemes," *JACM.*, vol. 30, no. 3, pp. 514-550, July 1983.

[37] FARBER, M., "Characterizations of Strongly Chordal Graphs," *Discrete Mathematics*, vol. 43, no. 1, pp. 173-189, 1983.

[38] FARBER, M., "Domination, Independent Domination, and Duality In Strongly Chordal Graphs," *Discrete Appl. Math.*, vol. 7, no. 2, pp. 115-130, 1984.

[39] FARLEY, A.M., "Networks Immune to Isolated Failures," *Networks*, vol. 11, pp. 255-268, 1981.

[40] FARLEY, A.M., "Networks Immune to Line Failures," *Networks*, vol. 12, pp. 393-403, 1982.

[41] FISHER, D.C., "Lower Bounds On the Number of Triangles in the Graph," *Jou. of Graph Theory*, vol. 13, no. 4, pp. 505-512, Sept. 1989.

[42] FULKERSON, D. AND GROSS, O., "Incidence Matrices and Interval Graphs," *Pac. J. Math*, vol. 15, pp. 835-855, 1965.

[43] GALIL Z., "An almost linear time algorithm for computing dependency basis in a relational database," *JACM*, vol. 29, no. 1, pp. 96-102, January

1982.

[44] GAREY, M.R. AND JOHNSON, D.S., "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W.H. Freeman and Company, San Fransisco*, 1979.

[45] GAVRIL, F., "Algorithms For Minimum Coloring, Maximum Clique, Minimum Covering By Cliques, and Maximum Independent Set Of A Chordal Graph," *SIAM J. Comput.*, vol. 1, pp. 180-187, 1972.

[46] GNESI, S., MONTANARI, U., AND MARTELLI, A., "Dynamic Programming as Graph Searching," *JACM*, vol. 28, 1981.

[47] GOLUMBIC, M.C., "Algorithmic Graph Theory and Perfect Graphs," *Academic Press, New York*, 1980.

[48] GOODMAN, NATHAN AND SHMUELI, ODED, "NP-Complete Problems Simplied on Tree Schemas," *Acta Informatica*, vol. 20, pp. 171-178, 1983.

[49] GOULD, RONALD, "Graph Theory," *The Benjamin/Cummings Publishing Company, Inc.*, 1988.

[50] HARARY F., "Graph Theory," *Addison-Wesley, Reading, MA*, 1969.

[51] HE, XIN AND YESHA, YAACOV, "Efficient Parallel Algorithms for r-Dominating Set and p-Center Problems on Trees," *Algorithmica*, vol. 5, pp. 129-145, 1990.

[52] HO CHIN-WEN. AND LEE R.C.T., "Efficient Parallel Algorithms for Finding Maximal Cliques, Clique Trees, and Minimum Coloring on

Chordal Graphs," *IPL*, vol. 28, pp. 301-309, August 1988.

[53] HOWORKA E., "A Characterization of Ptolemaic Graphs," *J. of Graph Theory*, vol. 5, pp. 323-331, 1981.

[54] IYENGAR S.S., RAO N.S.V., KASHYAP R.L., AND VAISHVANI V.K., "Multidimensional Data Structures: Review and Outlook," *Advances in Computers*, vol. 27, pp. 70-119, 1988.

[55] KARLSSON, R.G. AND OVERMARS, M.H., "Normalized Divide-and-Conquer: A Scaling Technique for Solving Multi-Dimensional Problems," *IPL*, vol. 26, pp. 307-312, 1987/88.

[56] KARP, R.M. AND RAMACHANDRAN, V., "A Survey of Parallel Algorithms for Shared-Memory Machines," *Rept. No. UCB/CSD 88/408, Comp., Sci., Division, Berkeley, California*, 1988.

[57] KARP, R.M., UPFAL, E., AND WIGDERSON, A., "Finding a Maximum Matching is in Random NC," *17th Annual Symp. on Theory of Computing*, pp. 22-32, 1985.

[58] KATZ, M.D. AND VOLPER, D.J., "Geometric Retrieval in Parallel," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 92-102, 1988.

[59] KEIL, J.M., "Total Domination In Interval Graphs," *Information Processing Letters*, vol. 22, pp. 171-174, 1986.

[60] KLEIN, P.N., "Efficient Parallel Algorithms for Chordal Graphs," *FOCS 1988*.

[61] KLUG, A., "On Conjunctive Queries Containing Inequalities," *JACM*, vol. 35, no. 1, pp. 146-160, Jan. 1988.

[62] LASKAR, R. AND PFAFF, J., "Domination And Irredundance In Split Graphs," *Tech. Rept. 430, Clemson University, Clemson, SC*, 1983.

[63] LASKAR, R., PFAFF, J., HEDETNIEMI, S.M., AND HEDETNIEMI, S.T., "On the Algorithmic Complexity of Total Domination," *SIAM J. Alg. Disc. Meth.*, vol. 5, pp. 420-425, 1984.

[64] LAURITZEN, S.L. AND SPIEGELHALTER, D.J., "Local Computations with Probabilities on Graph Structures and their Applications to Expert Systems," *Journal of Royal Statistical Society*, 1988.

[65] LEHMAN, P. AND YAO, S.B., "Efficient locking for concurrent operations on B-trees," *ACM TODS*, vol. 6, no. 5, pp. 650-670, December 1981.

[66] LIPSKI, W., "On Databases with Incomplete Information," *JACM*, vol. 28, no. 1, pp. 41-70, Jan. 1981.

[67] LUBIW, A., "Γ-free Matrices," *M.S. Thesis, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ont.*, 1982.

[68] MAIER DAVID., "The Theory of Relational Databases," *Computer Science Press, Rockville, Md.*, 1983..

[69] MILLER, R. AND STOUT, Q. F., "Mesh Computer Algorithms for Line Segments and Simple Polygons," *Int'l Conf. on Parallel Processing*, pp. 282-285, 1987.

[70] MOITRA, ABHA AND IYENGAR S.S., "Parallel Algorithms for Some Computational Problems," *Advances in Computers*, vol. 26, pp. 94-153, 1987.

[71] MULMULEY, K., VAZIRANI, U.V., AND VAZIRANI, V.V., "Matching is Easy as Matrix Inversion," *Combinatorica*, vol. 7, no. 1, pp. 105-113, 1987.

[72] MU, Z. AND CHEN, M.C., "Communication-Efficient Distributed Data Structures on Hypercube Machines," *Department of Computer Science, Yale University*, 1986.

[73] NOAR J., NOAR M., AND SCHAFFER A.A., "Fast Parallel Algorithms for Chordal Graphs," *Proc. 19th Ann. ACM Symp. on Theory of Computing*, pp. 355-364, 1987.

[74] OMIECINSKI, E. AND TIEN, E., "A Hash-Based Join Algorithm for a Cube-Connected Parallel Computer," *IPL*, vol. 30, pp. 269-275, March 1989.

[75] PARBERRY, I., "Parallel Complexity Theory," *Monographs*, 1988.

[76] PIPPENGER, N., "On Simultaneous Resource Bounds," *Proc. STOC., New York*, May 1987.

[77] PREPARATA, F.P. AND SHAMOS, M.I., "Computational Geometry : An Introduction," *Springer-Verlag, New York*, 1985.

[78] RABIN, M.O. AND VAZIRANI, V.V., "Maximum Matchings in General Graphs through Randomization," *Journal of Algorithms*, vol. 10, pp.

557-567, 1989.

[79] RADHAKRISHNAN, SRIDHAR AND IYENGAR, S.S., "Fast Parallel Algorithms on Strongly Chordal Graphs," *Submitted to IPL*, 1989.

[80] RADHAKRISHNAN, SRIDHAR AND IYENGAR, S.S., "The Complexity of Processing Implication on Queries and Chordal Graphs," *Submitted to IEEE Trans. on Software Engineering*, 1989.

[81] RADHAKRISHNAN, SRIDHAR AND IYENGAR, S.S., "Efficient Parallel Algorithms For Functional Dependency Manipulations," *2nd Int'l Symposium on Databases in Parallel and Distributed Systems, July 2-4, Dublin, Ireland. Submitted to IEEE Trans. on Comput.*, 1989.

[82] RADHAKRISHNAN, SRIDHAR AND IYENGAR, S.S., "Fast Parallel Algorithms for Recognizing Strongly Chordal, Ptolemaic, and Block graphs," *To be presented a the 19th International Conference on Parallel Processing. Submitted for Publication*, August, 1990.

[83] RADHAKRISHNAN, SRIDHAR AND IYENGAR, S.S., "Linear-Time Sequential and Fast Parallel Algorithms for Domination Problems in Strongly Chordal Graphs.," *Submitted to SIAM J. on Discrete Mathematics*, March, 1990.

[84] RADHAKRISHNAN, SRIDHAR, IYENGAR, S.S., AND RAJANARAYANAN, SUBBIAH, "Range Search In Parallel Using Distributed Data Structures," *Int'l PARBASE-90 Conf., Miami, Florida, To Appear in Journal of Parallel and Distributed Computing*, 1990.

[85] REIF, J.H., "An Optimal Parallel Algorithm for Integer Sorting," *FOCS*, pp. 496-503, 1985.

[86] SCOTT, L.R., BOYLE, J.M., AND BAGHERI, B., "Distributed Data Structures for Scientific Computation," *Departments of Computer Science and Mathematics, Pennsylvania State University*, 1987.

[87] SHILOACH, Y. AND VISHKIN, U., "An $O(logn)$ Parallel Connectivity Algorithm," *Journal of Algorithms*, vol. 3, pp. 57-67, 1982.

[88] SOUMYANATH, K. AND DEOGUN, J.S., "On the Bisection Width of partial-K-trees," *Presented at the 20th Southeastern Conf. on Combinatorics, Graph Theory and Computing, Boca Raton, Florida*, Feb. 1989.

[89] STOJMENOVIC, I., "Computational Geometry on a Hypercube," *Int'l Conf. on Parallel Processing*, pp. 100-103, 1987.

[90] SUN, XIAN-HE, NABIL, K.N., AND LIONEL, N.M., "Processing Implication on Queries," *IEEE Trans. on Software Engineering*, vol. 15, no. 10, pp. 1168-1175, Oct. 1989.

[91] TARJAN, R.E. AND VISHKIN, UZI, "An Efficient Parallel Biconnectivity Algorithm," *SIAM J. Comput.*, vol. 14, no. 4, November 1985.

[92] TARJAN., ROBERT E. AND YANNAKAKIS., MIHALIS, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs," *SIAM J. COMPUT.*, vol. 13, no. 3, pp. 566-579, August 1984.

[93] D., ULLMAN, J., "Principles of Database Systems," *Computer Science*

Press, Rockville, Md., 1983..

[94] UPFAL, E. AND WIGDERSON, A., "How to Share Memory in a Distributed Environment," *Proc. 25th Annu. IEEE Symp. Foundations Comput. Sci., West Palm Beach, FL*, pp. 171-180, October 1984.

[95] VITTER, J.S. AND SIMONS, R.A., "New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems in *P*," *IEEE. Trans. On Comput.*, vol. c-35, no. 5, pp. 403-417, May 1986.

[96] WALD, J.A. AND COLBOURN, C.J., "Steiner Trees, partial-2-trees, and Minimum IFI Networks," *Networks*, vol. 13, pp. 159-167, 1983.

[97] WHITE, K., FARBER, M., AND PULLEYBLANK, W.R., "Steiner Trees, Connected Domination and Strongly Chordal Graphs," *Network*, vol. 15, pp. 109-124, 1985.

[98] YANNAKAKIS, M., "Computing Minimum Fill-In is NP-Complete," *SIAM J. Algebraic Discrete Math.*, vol. 2, pp. 77-79, 1981.

[99] IYENGAR, S.S., "Fast Parallel Algorithms for a Class of Graph Related Structures," *4th Annual Parallel Processing Symposium, Fullerton, California*, vol. 2, pp. 1, April 4-6, 1990.

VITA

Sridhar Radhakrishnan was born in Kanchipuram, Tamil Nadu, India on March 25, 1963. He obtained his undergraduate degrees in physics and computer science from Vivekananda college, University of Madras in 1983 and University of South Alabama, Mobile, Alabama in 1985, respectively. He completed the dual masters's degree program in library and information science and systems science at Louisiana State University, Baton Rouge, Louisiana in 1987.

His research interests include design and analysis of parallel algorithms, databases, graph theory, and computational geometry.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Sridhar Radhakrishnan

Major Field: Computer Science

Title of Dissertation: Fast Parallel Algorithms On A Class Of Graph Structures With Applications In Relational Databases and Computer Networks.

Approved:

S. Vij
Major Professor and Chairman

F. Glen Hamburg
Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination: April 23, 1990