

To the Graduate Council:

I am submitting herewith a thesis written by Eric S. Kirsch entitled "Practical Parallel Algorithms for Chordal Graphs". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jean R. S. Blair

Dr. Jean Blair, Major Professor

We have read this thesis
and recommend its acceptance:

DJ Smyk
Don Durigan

Accepted for the Council:

Cowminkel

Vice Provost

and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in her absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Eric S. Kirsch

Date April 24, 1989

**PRACTICAL PARALLEL ALGORITHMS
FOR CHORDAL GRAPHS**

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Eric S. Kirsch

May 1989

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jean Blair, who was patient and understanding but, at the same time, demanding. She has started the long process of teaching me what being a professional means.

I would also like to thank Dr. Dunigan and Dr. Straight for their time and energy while serving on my committee and for everything they taught me while I was taking classes from them.

I would like to thank my parents and family for their continued support and belief in me. I truly couldn't ask for a better family.

My life at the University of Tennessee seemed to be an endless string of emergencies and Ethel Wittenberg, Wallace Mayo and Beth Dosser were always there to help apply bandaids.

I would also like to thank Larry Webb, Clarence Potter and Randy Baker for their friendship and help during my stay at the University of Tennessee. And a special thank you goes to my friend, editor and graphics designer, Michelle Gallahar.

ABSTRACT

Given a chordal graph, parallel solutions are given for finding the set of maximal cliques, a perfect elimination ordering and a minimum coloring. The solutions to these problems have a variety of applications. While other parallel solutions to these problems exist, this paper presents algorithms that use resources more efficiently and involve less total computation and so have a faster running time on a fixed number of processors. Questions of processor grain size, communication costs, algorithm costs and possible pipelining benefits are also discussed.

Contents

1	Introduction	1
1.1	Terminology	2
1.2	Literature Review - Maximal Cliques and Colorings for General Graphs	3
1.3	Classes of Graphs	4
1.3.1	Intersection Graphs	4
1.3.2	Perfect Graphs	6
1.4	Parallel Algorithms	7
2	Chordal Graphs and Algorithms	10
2.1	Properties of Chordal Graphs	10
2.2	Sequential Algorithms on Chordal Graphs	11
2.3	Parallel Algorithms on Chordal Graphs	13
3	Finding Maximal Cliques in Parallel	17
3.1	Discussion of Finding Maximal Cliques	18
3.2	Maximal Clique Algorithm	27
3.3	Example of Maximal Clique Development	28
3.4	Time Complexity Analysis	31
3.5	Summary	34
4	Generating a PEO in Parallel	35

4.1	Discussion of Finding a PEO	35
4.2	PEO Algorithm	39
4.3	Example of PEO Development	41
4.4	Time Complexity Analysis	43
4.5	Summary	46
5	Generating a Minimum Coloring in Parallel	47
5.1	Discussion of Parallel Coloring	47
5.2	Coloring Algorithm	50
5.3	Example of Coloring	50
5.4	Time Complexity	52
5.5	Summary	52
6	Conclusion	55
6.1	Summary	55
6.2	Future Work	58
	BIBLIOGRAPHY	61
	VITA	65

List of Figures

3.1	Chordal Graph	24
3.2	Partial Maximal Clique Generation with Clique Graph Edges . . .	25
3.3	Clique Generation	26
3.4	Algorithm: MaxClique	29
4.1	Removal of A does not remove a maximal clique	38
4.2	Algorithm: Vertex Processor PEO	40
4.3	Algorithm: Host PEO	42
4.4	Clique list and count for each processor	42
4.5	Example of Chordal Graph with little pipelining	45
5.1	Algorithm: Host Coloring	51
5.2	Algorithm: Processor Coloring	51
5.3	Messages and final coloring	53
5.4	Color Sets	53
5.5	Colored graph	54
6.1	Algorithm Statistics	56
6.2	Maximal Clique Algorithm Comparison	57
6.3	PEO Algorithm Comparison	57
6.4	Minimum Coloring Algorithm Comparison	57
6.5	Clique Tree with Simplicial Vertices	60

Chapter 1

Introduction

Many practical problems can be solved using graph algorithms. Unfortunately, most graph problems are NP-Complete when the graph structure is general. This motivates an effort to find sub-classes of graphs which can represent realistic problems and for which there exist efficient solutions to practical problems. Some important graph problems with realistic applications include coloring, shortest path, max-flow, spanning tree and perfect elimination ordering [5][14][21]. Serial graph algorithms have been widely studied [5][11][27]. Recently, research has focused on parallel algorithms [18][20][22]. These parallel algorithms facilitate finding solutions to large problems where solutions to these large problem instances would be prohibitive on a single processor. As the hardware becomes increasingly less expensive, parallel solutions become increasingly accessible. In this thesis, new parallel algorithms for solving three particular graph problems on a class of graphs called chordal graphs are presented. While other parallel algorithms have been developed for these problems, the algorithms presented here are more efficient for parallel machines with a fixed number of processors.

The remainder of this chapter gives the terminology of graphs and a review of the literature concerning graph coloring, classes of graphs, and parallel algorithms. Chapter 2 discusses general properties of chordal graphs. Chapters 3,

4 and 5 present new algorithms for finding maximal cliques, finding a Perfect Elimination Ordering, and for finding a minimum coloring, respectively. All of these algorithms are parallel algorithms and build upon one another. In the last chapter, a summary of the results presented here is given and some open problems are discussed.

1.1 Terminology

In this paper standard graph definitions are used[8] [14]. Briefly, a graph G is an undirected graph which consists of a set of vertices $V = \{x, y, \dots, z\}$ and a set of edges E . The set of edges represents a relation in the vertex set. The set of vertices that are adjacent to a vertex v is denoted by $Adj(v)$ and $N(v) = v + Adj(v)$ is the neighborhood of v . $|V| = n$ is the number of vertices in V , and $|E|$ is the number of edges in E .

A subgraph of G is any graph H such that $H = (V', E')$ where $V' \subseteq V$, $E' \subseteq E$ and the endpoints of each edge in E' are in V' . If $A \subseteq V$ then the subgraph induced by A is $G_A = (A, E_A)$ where $E_A = \{(x, y) \in E | x \in A \text{ and } y \in A\}$. It is apparent that not all subgraphs are induced subgraphs. Another notation that is used here for an induced subgraph is $G - A$, where $A \subseteq V$. This is the subgraph of G induced by the set of vertices $V - A$.

A *clique* is a subset C of vertices of G such that G_C induces a complete subgraph. In other words, a clique is a group of vertices that are pairwise adjacent. A clique C is *maximal* if there is no clique of G that properly contains C . A *clique cover* is a minimum number of cliques such that all vertices, v of G belong to a clique.

The *degree* of a vertex v , denoted by $d(v)$, is equal to $|Adj(v)|$. An *independent set* of G is a subset X of V such that no two vertices in X are adjacent.

A *coloring* of G is a partition of the vertices of G , $X_1, X_2, \dots, X_k$ such that each X_i is an independent set. If $V = X_1 + X_2 + \dots + X_k$ then G is k -colorable. If k is the minimum number of colors necessary to color G then k is the *chromatic number* of G .

As mentioned earlier, the focus of this thesis is three new parallel graph algorithms. These new algorithms determine: (1) the set of maximal cliques, (2) a perfect elimination ordering (to be defined later) and (3) a minimum coloring of a chordal graph. In the next section, a brief overview of the state-of-the-art for finding maximal cliques and for coloring general graphs is given. In section 1.3, various classes of graphs which have previously been considered are presented. Finally, in section 1.4, some background for parallel algorithms is given. We will then be ready to discuss the particular class of graphs that we are interested in and subsequently to present the new parallel graph algorithms.

1.2 Literature Review - Maximal Cliques and Colorings for General Graphs

The problem of finding maximal cliques for a general graph was shown to be NP-Complete in [10]. Consequently, any algorithm to develop maximal cliques will require exponential time on an arbitrary graph. In [27], an exponential time algorithm for finding maximal cliques is given. This algorithm starts by finding the maximal clique for one vertex and then processes the remaining vertices sequentially. As each vertex, v , is processed the set $Adj(v)$ is found, and the maximal cliques found so far are augmented as required.

Two types of algorithms exist for coloring general graphs: exact coloring algorithms and approximate coloring algorithms. Since minimum graph coloring is NP-Complete[10] for general graphs, the exact coloring algorithms also have an

exponential time complexity. For an example of an exact coloring algorithm the reader is referred to [5]. The heuristic used most often for approximate coloring is to arrange the vertices in descending order by degree[5][21]. An approximate coloring can then be obtained by coloring the vertices in order. Each vertex v is colored with the lowest possible number i such that no member of $Adj(v)$ has been colored with i . A variation on this algorithm, as presented in[5], is to re-order the set of uncolored vertices after each color has been assigned. This re-ordering excludes already colored vertices (and adjacent edges) when calculating the degree of each vertex. A significant improvement can be made in the number of colors used if certain interchanges are made to the partial colorings produced after each vertex is colored[21].

1.3 Classes of Graphs

The problem of finding efficient algorithms for many graph problems has given rise to an effort to categorize graphs by different properties. The ability to define a class of graphs has, in many instances, made it possible to find optimal solutions to problems that do not have reasonable solutions on general graphs.

Two commonly used types of graphs will be examined in the next sections: intersection graphs and perfect graphs. The algorithms in this paper are developed for chordal graphs. A chordal graph is a type of intersection graph, as well as a perfect graph.

1.3.1 Intersection Graphs

The class of *intersection graphs* can be characterized as the intersection of some family of sets[14]. By this type of characterization, each vertex represents a

set, and there is an edge between vertices if and only if their corresponding sets intersect. If the family of sets is allowed to be any family of sets, then the class of graphs is the set of all undirected graphs[14]. But it is possible to limit the sets by some characteristic or pattern, motivated by applications found in practice, in such a way that efficient solutions to problems that are not solvable on general graphs can easily be found. Several types of intersection graphs are described below but many other types exist[14].

If the family of sets is limited to intervals on the real line (or some linearly ordered set), then the resulting graph is called an *interval graph*. If the two ends of the “real line” are joined, then the intervals become arcs on a circle and the graph is called a *circle-arc graph*. Interval graphs are a proper subset of circular arc graphs. If none of the arcs properly contain another arc, then the graph is a *proper circle-arc graph*.

Given two parallel rows with p points on each and a set of lines connecting pairs of points on opposite rows, a *permutation graph* has a vertex for each line and an edge between two vertices if and only if the corresponding line segments cross. If the $2p$ points are allowed to be points on a circle, then the set of lines connecting these points are chords of a circle and the graph is referred to as a *circle graph*. Permutation graphs are a proper subset of circle graphs.

Another variation on intersection graphs is one where the graphs represent the intersection of a family of subtrees of a tree. This class of graphs is called *chordal graphs*. Another, more common, definition of chordal graphs defines a graph G as being chordal if every cycle of length 4 or greater has a chord. (A chord is an edge between two non-adjacent vertices in the cycle). Chordal graphs are also known as *triangulated*, *rigid-circuit*, *monotone transitive* and *perfect elimination* graphs. Interval graphs are a proper subset of chordal graphs. There is some overlap between the sets of circular arc graphs, circle graphs and chordal graphs.

However, there is no subset inclusion relationship.

Chordal graphs exhibit a hereditary property, which is to say that every induced subgraph of a graph G is also chordal. Interval graphs also exhibit hereditary properties[14]. This hereditary property is useful for solving various graph problems including the problems solved in this paper.

1.3.2 Perfect Graphs

Another important class of graphs is the class of *perfect graphs*. Perfect graphs have two main properties:

- The chromatic number of any induced subgraph of a perfect graph G is equal to the size of the largest maximal clique for that induced subgraph.
- The size of the clique cover of any induced subgraph of a perfect graph G is equal to the size of the largest independent set for that induced subgraph[14].

Lovasz[14] proved a stronger result, known as the Perfect Graph Theorem which states that the two properties are equivalent. A third property that is also equivalent to the first two is that the size of the largest maximal clique times the size of the largest independent set is greater than or equal to the number of vertices for the perfect graph. These results apply for any induced subgraph of a perfect graph, and will be useful in developing the algorithms presented later.

Chordal graphs were the first class of graphs to be recognized as perfect. Not all perfect graphs are chordal, however. Permutation graphs are perfect, but circle-arc graphs and circle graphs are, in general, not perfect.

Two types of graphs have been discussed in the preceding sections: intersection graphs and perfect graphs. Their areas of overlap have also been examined. The subject of the remainder of this paper is chordal graphs.

1.4 Parallel Algorithms

Most of the parallel algorithms for finding solutions to chordal graph problems have been an attempt to prove that a certain problem was in problem class NC. (A representative set of these parallel algorithms is summarized in the next chapter). NC is a class of problems which have a running time bounded by the logarithm of the size of the input and require a number of processors equal to a polynomial on the size of the input. Unfortunately, solutions under these restrictions are not very practical. Even if the number of processors required is only the square of the input size, a problem with an input size of 1000, not an exceptionally large input size, will require 1,000,000 processors. These types of algorithms also completely ignore the fact that existing parallel processors tend to have a fixed number of processors, and so cannot provide the necessary processors for large graph problems. This paper, while not strictly addressing itself to the specifics of a certain machine or the details of an implementation, attempts to produce algorithms which are more realistic for some existing parallel processors such as a hypercube. That is, algorithms which are not geared to producing only the maximum speedup but which also consider efficient use of resources are presented here.

The algorithms discussed in this paper are designed for what has been called a "multicomputer"[26]. In this model of a parallel processor, each processor has its own memory and there is no shared memory. The multicomputer is also an MIMD computer (Multiple-Instruction stream, Multiple-Data stream). While machines consistent with the multicomputer model are used in practice as often as the shared memory parallel processor, most theoretical research has not explored the problems and potential of this model.

Each of our algorithms will be evaluated by considering its relative[26]:

- **Cost:** the complexity of algorithm times the number of processors used(p),
- **Speedup:** ratio of the time required by the parallel computer to execute the fastest serial algorithm to the time required by the same computer to execute the parallel algorithm,
- **Efficiency:** the speedup divided by p , and
- **Grain size:** the amount of work to be performed by each processor between synchronizations.

The efficiency of an algorithm can be measured either from a theoretical or a practical point of view. The theoretical measurements give insight into the general complexity of the different algorithms. The practical measurements attempt to seriously compare the performance of various methods of solving a problem and to take into account more of the difficulties of implementation. It is not always the case that a theoretically efficient algorithm will be efficient in practice, or that a good practical algorithm will have good theoretical bounds on its performance. For example, even ignoring processor startup times and bus contention problems, the PEO algorithm of [20] (presumably the best of the parallel algorithms on chordal graphs) uses only $n + m$ processors and has a time complexity of $O(\log^2 n)$ which gives it an efficiency of $O(\frac{n+m}{\log^2 n})$ or equivalently $O(1/\log^2 n)$. The efficiency of an algorithm is considered good when it is close to one and for a large graph the efficiency of this PEO algorithm is not close to one. *Processor startup time* is the computation cost of starting up a processor. *Bus contention* occurs when two or more processors need to send messages along the same bus.

The above discussion is not meant to minimize the value of theoretical research into finding parallel solutions to problems. The theoretical research provides valuable insight into methods by which problems may be parallelized. But

there is concern about the unbounded space and processor requirements of such algorithms[16]. It would be difficult to implement an algorithm requiring a number of processors equal to the size to the fourth power of the input size on, say, a hypercube with 16 processors. Furthermore, the argument that it is possible for a processor to perform the work of multiple processors does not answer the question of whether this is the most efficient solution for this number of processors. It is possible to look at the cost of an algorithm as defined above and then to examine the time complexity when this cost is distributed over a fixed number of processors. The goal of parallel algorithms needs to be more than just achieving maximum speedup no matter the cost in processors. Each processor needs to be used efficiently. It is also necessary to consider the grain size of work for each processor. The processor must perform enough work to offset the start up time. The algorithms in this paper have been developed with these goals in mind.

Chapter 2

Chordal Graphs and Algorithms

The algorithms developed in this paper are for chordal graphs. As mentioned previously, chordal graphs have many practical applications. This section will explore some of the properties of chordal graphs and then review some of the existing sequential and parallel algorithms for finding maximal cliques, a PEO, and a coloring for chordal graphs.

2.1 Properties of Chordal Graphs

The first aspect of chordal graphs which distinguishes them from other graphs is that every chordal graph has at least two simplicial vertices[14]. A vertex is simplicial if its adjacency set $Adj(v)$ is a clique. This clique need not be maximal and, in fact, it is not. Since every induced subgraph of a chordal graph is also chordal, all induced subgraphs of a chordal graph G also have a simplicial vertex. Thus it is possible to develop an ordering of vertices of a chordal graph G such that each $v_i \in V$ is simplicial on the induced subgraph $G - \{v_j | j < i\}$. An ordering that meets these requirements is called a *perfect elimination ordering* (PEO). It should be noted that graphs that are not chordal do not have PEOs.

A vertex's position in a PEO ordering will be denoted by $\sigma(v)$. So if a vertex v was the i th vertex in the PEO, then $\sigma(i) = v$ and $\sigma^{-1}(v) = i$. PEOs are useful for a variety of applications including Gaussian elimination [30].

Another aspect of chordal graphs that makes them worth studying pertains to vertex separators. A vertex separator for vertices x and y , where x and y are not adjacent, is a set of vertices $S \subset V$ such that the removal of S from the chordal graph G results in two distinct connected components, one containing x and the other containing y . If there is not a proper subset of S that is also a vertex separator for x and y then S is a *minimal vertex separator* for x and y . For a chordal graph G , each minimal vertex separator is a clique.

The above mentioned properties give rise to the following equivalent[14] statements:

- G is chordal,
- G has a PEO and any simplicial vertex can start a PEO,
- every minimal vertex separator of G is a clique in G .

These definitions and characteristics of chordal graphs will aid in the development of the algorithms presented later.

2.2 Sequential Algorithms on Chordal Graphs

Three basic parallel algorithms for chordal graphs are presented in this paper. They accomplish: (1) finding all maximal cliques, (2) producing a PEO and (3) finding a minimum coloring. In order to judge how well these algorithms perform, it is necessary to compare them to the best existing sequential algorithms. Most sequential approaches for chordal graphs start by generating a PEO. The PEO is then used to generate maximal cliques and a minimum coloring.

There are several sequential algorithms for finding a PEO. Most of these algorithms (all of the algorithms discussed below) generate the PEO in reverse order. This means that the first vertex numbered by these algorithms receives the number n the next $n - 1$ etc.

First, *Lexicographic Breadth-First Search (LBFS)* [30] directs the search for the next vertex of the PEO by maintaining a list of labels for each vertex. Initially, every vertex is assigned a list that contains the number 0. Lists are generated and vertices are chosen as follows: a vertex is selected at random and assigned the number n , and the label n is added to the vertex list for each adjacent vertex. The next vertex to be selected is a vertex, v , with the highest lexicographic label. The vertex v is assigned the next lower PEO number, and every vertex adjacent to v has this number appended to its label. This process continues until all vertices have been numbered. Through some fairly complex list processing, the process of finding the next vertex to number takes constant time and the process of updating the labels of vertices takes a total of $O(m)$ time. Consequently, the running time of the algorithm is $O(n + m)$.

A second sequential algorithm for finding a PEO, known as *Maximum Cardinality Search (MCS)* [33], chooses the first vertex to be numbered at random, and each successive vertex to be chosen is the vertex which is adjacent to the most vertices that have already been numbered. MCS, while conceptually simpler than LBFS, has the same $O(n + m)$ running time.

There are a multitude of PEOs for any chordal graph. For some uses, one PEO may be significantly better than another. Unfortunately, neither LBFS nor MCS as originally presented can produce every PEO of a given chordal graph. However, in [32], a modified MCS called *Maximum Cardinality Component* is presented that can produce every PEO for a given chordal graph. In MCC instead of selecting the vertex adjacent to the most numbered vertices, as was done in

MCS, the vertices eligible for selection include any vertex that is adjacent to the most numbered vertices in any connected component on the subgraph $G - \{v_i | v_i \text{ already numbered by MCC}\}$.

Given a PEO, it can be used to develop maximal cliques since for every maximal clique C_v , the maximal clique containing v must be of the form $C_v = \{v\} \cup X_v$ where $X_v = \{x \in Adj(v) | \sigma^{-1}(v) < \sigma^{-1}(x)\}$ [14]. Obviously, not all cliques of this form are necessarily maximal cliques. But this formulation indicates that there are at most n maximal cliques for a chordal graph[14]. A straight forward algorithm for finding maximal cliques is presented in [14]. For each vertex, v , the set of vertices is found which are adjacent to v and which are higher in the PEO ordering. This set unioned with $\{v\}$ is either a maximal clique or a subset of a maximal clique. The algorithm determines non-maximal cliques in constant time. Safeguards are included in the algorithm to insure that the generated clique is actually maximal and not a subset of a previously produced maximal clique[14]. The total running time of the algorithm is $O(m)$.

The PEO is also used to develop a minimum coloring[11]. For each vertex, v taken in PEO order, the coloring algorithm examines the existing sets of colors. If a set is found in which there is not a member of the set $Adj(v)$, then v is added to this set. Otherwise a new set, $\{v\}$, is added to the list of color sets. Because the vertices are processed in PEO order, it is not possible that more than the minimum number of colors is required[11]. The running time of the algorithm is $O(n^2)$.

2.3 Parallel Algorithms on Chordal Graphs

In this section various parallel algorithms on chordal graphs will be examined. It is more difficult to summarize this research because different researchers

start by solving different problems and use these results to aid in solving other problems.

Consequently, we will examine each research effort separately. The first two papers to be discussed find the set of maximal cliques and use them to generate a clique tree from which they find a minimum coloring and a PEO. The last paper we will examine is somewhat different in that it finds a PEO and then finds a minimum coloring.

Bi-cliques are used to develop maximal cliques in [22]. A *bi-clique* is a graph whose vertex set can be divided into two sets such that the induced subgraphs produced by these sets are cliques. The algorithm computes all the maximal cliques in a chordal bi-clique by divide and conquer and then merges these to find the maximal cliques for the chordal graph. The time complexity of this algorithm is $O(\log^3 n)$ using $O(n^4)$ processors.

Continuing with the work reported in [22], the maximal cliques are used to produce a minimum coloring by first finding a clique tree for the graph. A *clique tree* T is a tree where each vertex in the tree represents a maximal clique of G and the cliques containing any vertex v form a connected sub-tree of T . A divide-and-conquer algorithm is used to construct the clique tree with a centroid clique being used at each division. A *centroid* is a vertex in the tree which, when removed, results in a forest where the largest tree is minimized. A coloring is obtained using this clique tree. Again, centroid cliques are found to divide the problem and recursion is used to color each connected component. These colorings are merged by renaming the color classes relative to each other. Each of these algorithms has an $O(\log^2 n)$ time complexity using $O(n^3)$ processors. A PEO is also found using the clique tree. To find a PEO, the branches of the clique tree are numbered, a PEO is found for each branch and then the PEOs are appended together. The algorithm runs in $O(\log^2 n)$ and uses $O(n^2)$ processors.

Another suite of parallel algorithms is given in [18]. Again, the set of maximal cliques is found before producing a coloring or a PEO. The algorithms in this paper utilize the concept of close-separators to find maximal cliques. A *close separator* is a minimal vertex separator where all the vertices in the separator are adjacent to one of the vertices being separated. Given a vertex w it is possible to partition each other vertex, v , of G by comparing the close separator between v and w . Each connected component within a partition is called a *w-close component*. In [18], the authors prove that for a vertex w of chordal graph G , if there exists a w -close component that is a clique then this component and its corresponding close separator form a maximal clique of G . The algorithm to find maximal cliques using this method has a time complexity of $O(\log^2 n)$ and requires $O(n^3)$ processors.

A clique tree is also found in [18] to construct a minimum coloring. Instead of using centroid cliques, though, minimal vertex separators are used. Once the clique tree has been constructed, a coloring is obtained from the clique tree by adding dummy members to each clique so that each clique has as many elements as the largest maximal clique. Then a link is made from one vertex in one clique to a vertex in a child clique if the vertices are not members of both cliques. These links mean that the vertices will receive the same color. Constructing the clique tree requires $O(\log n)$ time and $O(n^3)$ processors. The coloring requires $O(\log n)$ time and $O(n^2)$ processors. No PEO is found in [18].

Most parallel algorithms, until recently, have started by finding maximal cliques. In fact, it was argued in [22] that this was necessary because “algorithms to build a perfect elimination scheme and use it appear hard to parallelize because they choose one vertex to process at a time based on previous choices.”. In [20], Klein refuted this assertion. There, the notion of *backward paths* is used to iteratively refine an initial numbering until a PEO is produced. Given a numbering

of vertices, a backward path is defined as a path in G where the internal nodes along the path all have higher numbers than the endpoints of the path. For the numbering to be a PEO, all vertices must have a unique number, and all backward paths in the graph must have adjacent endpoints. The algorithm starts with a numbering that doesn't contradict a PEO (all vertices receive the number 0). The algorithm then repeatedly refines this numbering in parallel by modifying the numbering to create backward paths. The process continues until the numbering represents a PEO. The algorithm takes $O(\log^2 n)$ time and $O(n + m)$ processors.

A minimum coloring is found in [20] using cliques, but the cliques are found at the same time as the coloring is made. An initial clique is found and the remainder of the graph is broken into components. Cliques are found in each of these components and a recursive call is made to color these induced sub-graphs. The colorings are then merged. The time complexity is $O(\log^2 n)$ and the number of processors required is $O(m)$.

These algorithms provide many insights into the ways in which chordal graph problems may be parallelized. However, they are not very practical. They require far more processors than are currently available on most existing parallel processors. Furthermore, in many instances the grain size of the tasks that each processor performs are not sufficient to offset startup costs, and the efficiency of the algorithm is not very high. The rest of this paper will be devoted to developing algorithms that find maximal cliques, a PEO and a minimum coloring for a chordal graph. The goal is to develop parallel algorithms that are more practical than the ones discussed above.

Chapter 3

Finding Maximal Cliques in Parallel

As seen earlier, various starting points have been used to solve the three basic problems addressed here. In [22] and [18], maximal cliques were found first. In [20] a PEO was found first. In this paper, maximal cliques are found first but the approach is different from that used in [22] and [18]. In this approach a processor develops all the cliques to which a particular vertex belongs. While this means that cliques are developed more than once, knowing every clique to which a vertex belongs facilitates constructing the PEO and the clique tree. Maximal cliques are found first because maximal cliques are a key defining characteristic of chordal graphs. They can be used to develop a PEO, a clique tree and a coloring.

The first section in this chapter gives a general discussion of the method used to find maximal cliques. The second section gives the maximal clique algorithm. The third section gives an example of finding maximal cliques and the last section discusses the time complexity of the algorithm.

3.1 Discussion of Finding Maximal Cliques

When using the multicomputer model, the data for a problem must be partitioned so that it can be stored at various processors. To this end, for each vertex v_i of chordal graph G , an adjacency vector will be made. This vector consists of an n bit binary number with each bit representing a vertex in the graph. If the vertex v_i is adjacent to v_j then the j th position in the vector v_i is 1; otherwise it is 0. The i th spot in v_i 's vector is also 1. This will facilitate the refinement of candidate cliques. There are two possible ways to partition the data: each processor p gets an adjacency vector for each vertex, or each processor p gets an adjacency vector for itself and every vertex in $Adj(p)$. The first method is useful in the minimum coloring algorithm discussed later but requires $O(n^2)$ space on each processor. The second method, and the method used in this paper, requires $O(d * n)$ space, where d is the degree of the vertex. It should be noted, however, that this second method requires longer messages during the coloring algorithm.

In the explanation of the algorithm and the examples, n processors are going to be used, one for each vertex. Note, however, that it would be possible to allocate more than one vertex to a processor. For convenience, we will refer to a processor by the same name as the vertex that it is in charge of. Thus, processor, p , will be responsible for developing all the maximal cliques to which p belongs. At the same time that all maximal cliques are developed, a clique graph is going to be developed. (Clique graphs have applications in areas of computing probabilities in knowledge based systems[3][17], databases[2][9][15][34], numerical analysis[12][25][29] and other applications[19][6][24][31]). A clique graph, G_C , of a graph G is one where each vertex in G_C represents a maximal clique in G ,

and there is an edge between two vertices in G_C if and only if the intersection of the cliques that they represent is not empty. Each processor, while finding the maximal cliques to which p belongs, also develops edges between maximal cliques for the clique graph.

The idea directing the search for maximal cliques is that if a vertex p is adjacent to a vertex v , then p and v are in at least one clique together. Thus, the set of cliques to which p belongs is developed by using each of its adjacent vertices to refine the current set of candidate cliques. A *candidate clique* is a possible clique to which p belongs based on the information gathered so far. The original candidate clique is the set $Adj(p)$. Each vertex $v \in Adj(p)$ is used in turn to refine the list of candidate cliques. A refinement of one candidate clique by vertex $v \in Adj(p)$ involves possibly generating two candidate cliques. The first will contain p , v and all vertices from the original clique that are also in the adjacency vector of v . The second clique will contain all members of the original candidate clique including p , except for v .

A brute force method for generating the two candidate cliques is to check all edges between v and the vertices in $Adj(p)$. However, it is possible to determine the refinement that a vertex v will make on the candidate cliques in a quicker manner. If a logical *and* is performed between a candidate clique and the adjacency vector of v then the result will be a candidate clique with all vertices which can be in a clique with v . If the result is not equal to the original candidate clique, then there needs to be another candidate clique generated which keeps the vertices that were removed by the *and* operation but deletes the vertex v . Thus, with $d(p)$ refinements of the candidate clique list, every maximal clique in which p is a member can be generated.

It will be assumed throughout the remainder of this paper that logical *and* operations are available. If it is not possible to use *and* operations, perhaps due

to the size of the graph, then the refinement by a vertex v can be obtained by checking for an edge between v and every vertex with a lesser lexicographic¹ value than v . It is important to note that when referring to lexicographic order in this paper, the ordering will be determined by the adjacency vector representation and order. Thus, A is lexicographically greater than B since (1000...) is greater than (0100...). Only edges to vertices of lesser lexicographic value need be examined because each refinement vertex v is chosen in reverse lexicographic order (greatest first). Thus, any refinement that was required among lexicographically greater vertices would have been performed when the lexicographically greater vertices were refinement vertices. For each candidate clique to be refined by v , it is still possible to generate only two new candidate cliques: the first will contain all members of the original candidate clique except those vertices which were not in $Adj(v)$, the second will contain all members of the original candidate clique except v .

To find the maximal cliques then, we begin with a candidate clique that contains all the vertices to which p is adjacent. This clique list is refined by the above process for each $v \in Adj(p)$, and new cliques are created as necessary. The vertex whose adjacency vector is currently being used to refine the candidate cliques will be referred to as the *refinement vertex*. The refinement vertices are processed in reverse lexicographic order (greatest first). After the initial refinement, a clique will be refined only if the refinement vertex is in the candidate clique. If, when the clique and the adjacency vector are *anded*, the result is equal to the original clique, then no action is taken. If the result is not equal to the original clique then this result takes the place of the original candidate clique and a new candidate clique is generated. This new candidate clique is equal to the original candidate clique minus the refinement vertex, and is placed immediately

¹The lexicographic value used here and in the remainder of the paper is an arbitrary ordering, say, the order in which the vertices appear in the input.

following the result clique in the candidate clique list.

Theorem 1 *The method of generating cliques described above maintains a list of cliques that are in reverse lexicographic order.*

Proof: To prove this it is necessary to show:

- that the result of an *and* will always be lexicographically greater than the clique that is generated by removing the refinement vertex.
- that any clique that is the result of a refinement can never be greater than the clique that is the result of a refinement of a greater clique or less than the result of a refinement of a lesser clique.

To prove the first item, note that when a candidate clique is refined by v , it has already been verified that each vertex x , lexicographically greater than v , is in $Adj(v)$. Consequently, each vertex x lexicographically greater than v is in the result candidate clique. The refinement vertex must also be in the result since it is in the original candidate clique (or the *and* wouldn't be performed), and it is in the adjacency vector by definition. The new clique that is generated will retain all the vertices from the original candidate clique except the refinement vertex. Consequently, the refinement vertex **must** be the first difference in the vectors. Thus, the result is greater than the generated clique, and since this new candidate clique is placed following the result, order is maintained.

To prove the second item, note that if a refinement produces two new cliques, C_i and C_{i+1} , the clique that keeps the refinement vertex, C_i , is placed before the clique without the refinement vertex, C_{i+1} . This refinement vertex must be in each clique that can be generated from C_i since it has already been verified that the refinement vertex is adjacent to every vertex in C_i . While any clique that can be generated from C_{i+1} will never have the refinement vertex since

the candidate clique does not have the refinement vertex as a member. Thus, any cliques generated from cliques C_i and C_{i+1} will remain in order relative to the vertices greater than or equal to the refinement vertex. In addition, an induction argument can be used to show that any cliques generated before C_i or after C_{i+1} will remain in order. $\square$

It is possible, while developing these candidate cliques, to generate cliques that are non-maximal. These non-maximal cliques can be identified and removed as soon they are created. To identify non-maximal candidate cliques as they occur, "candidate" edges (as in the clique graph) will be maintained for each pair of candidate cliques. It is also apparent that it is only possible for cliques to become subsets of cliques which appear before them in the candidate clique list since each earlier candidate clique contains a vertex that will never appear later in the candidate clique list. Each candidate clique will keep a list of candidate edges to every candidate clique that appears before it in the candidate clique list. These candidate edges will be kept as vectors. As long as a candidate clique does not equal one of its edges, it contains a set of vertices that are different than any other candidate clique. When a clique is refined, it is possible that one or both of the resulting candidate cliques has had the vertex(vertices) removed that differentiated it from another candidate clique. As soon as this happens, the candidate clique will equal one of its edges and the clique can be removed. After all refinements have been made, the edges will be edges between the cliques for the clique graph.

When a clique that is not the last clique in the list is split, it will be necessary for this clique to update each of the edge lists of candidate cliques that appear later in the list. To facilitate this, a candidate clique will maintain a list of pointers to their edges in edge lists of candidate cliques which appear later in the candidate clique list.

As the candidate cliques are refined and new candidate cliques generated, it is necessary to update the edges between cliques. When a clique is refined and the result is not equal to the original clique, then the result gets the original clique's edges, updated to reflect the refinement. The clique which gets created from the original clique minus the refinement vertex gets a copy of these edges, also updated to reflect the refinement, and a new edge between the result and the new clique is added to this list. As the edges are updated the candidate clique is checked against the edges. If the clique equals any of its edges, it is a subset of another candidate clique and can never be maximal. Therefore, it and its edges are removed from the candidate clique list. Once a candidate clique has been removed, it will also be necessary to remove an edge from each of the later cliques.

The graph G in Figure 3.1 is a chordal graph and Figure 3.2 shows a partial list of the candidate clique development with edges for vertex J for graph G . Note on the fifth level (the last level shown) several candidate cliques have edges that are equal to the candidate clique. Consequently, these cliques will be removed from the candidate clique list. After all the refinements up to and including H , the candidate clique list (after removal of the non-maximal cliques) is the set:

$$\begin{aligned} &(- - - - EFG - - J - L -) \\ &(- - - - E - GH - J - L -) \\ &(- - - - - FG - IJ - L -) \\ &(- - - - - - - - IJKLM) \end{aligned}$$

A summary of the candidate clique list after each of the refinements for vertex J is given in Figure 3.3.

When all adjacency vectors of $v \in Adj(p)$ have been used to refine each of the cliques in the candidate clique list, the candidate clique list is a list of all

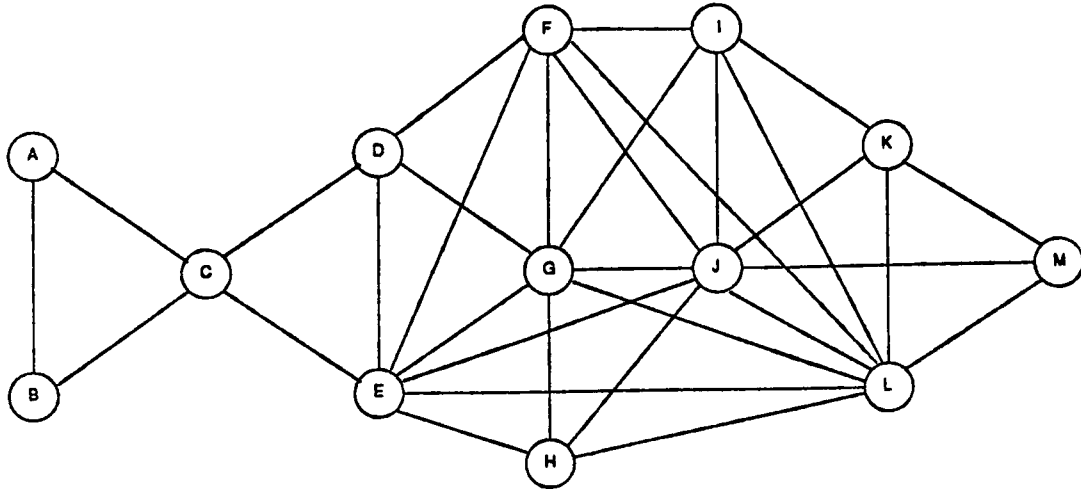


Figure 3.1: Chordal Graph

the maximal cliques in which p is a member, and the edge list gives a complete list of edges between each of the cliques containing p . Since every processor, p , develops all maximal cliques in which p appears, a copy of a maximal clique appears on each processor, v , where v is a member of the maximal clique. Thus, to eliminate the possibility of duplication when assembling the maximal cliques and/or clique graph the *owner* of a clique, defined to be the processor p such that p is the lexicographically greatest vertex in the clique, is responsible for reporting the existence of the clique. (Recall that lexicographically greatest is determined by vector representation). For example, A is the owner of the clique ABC and processors B and C will not report the clique ABC . In addition, only edges which are in the edge list of owned cliques are contributed.

For later use, a count of the number of maximal cliques involved with each particular vertex is also maintained. As mentioned earlier, a processor p develops all cliques to which p belongs.

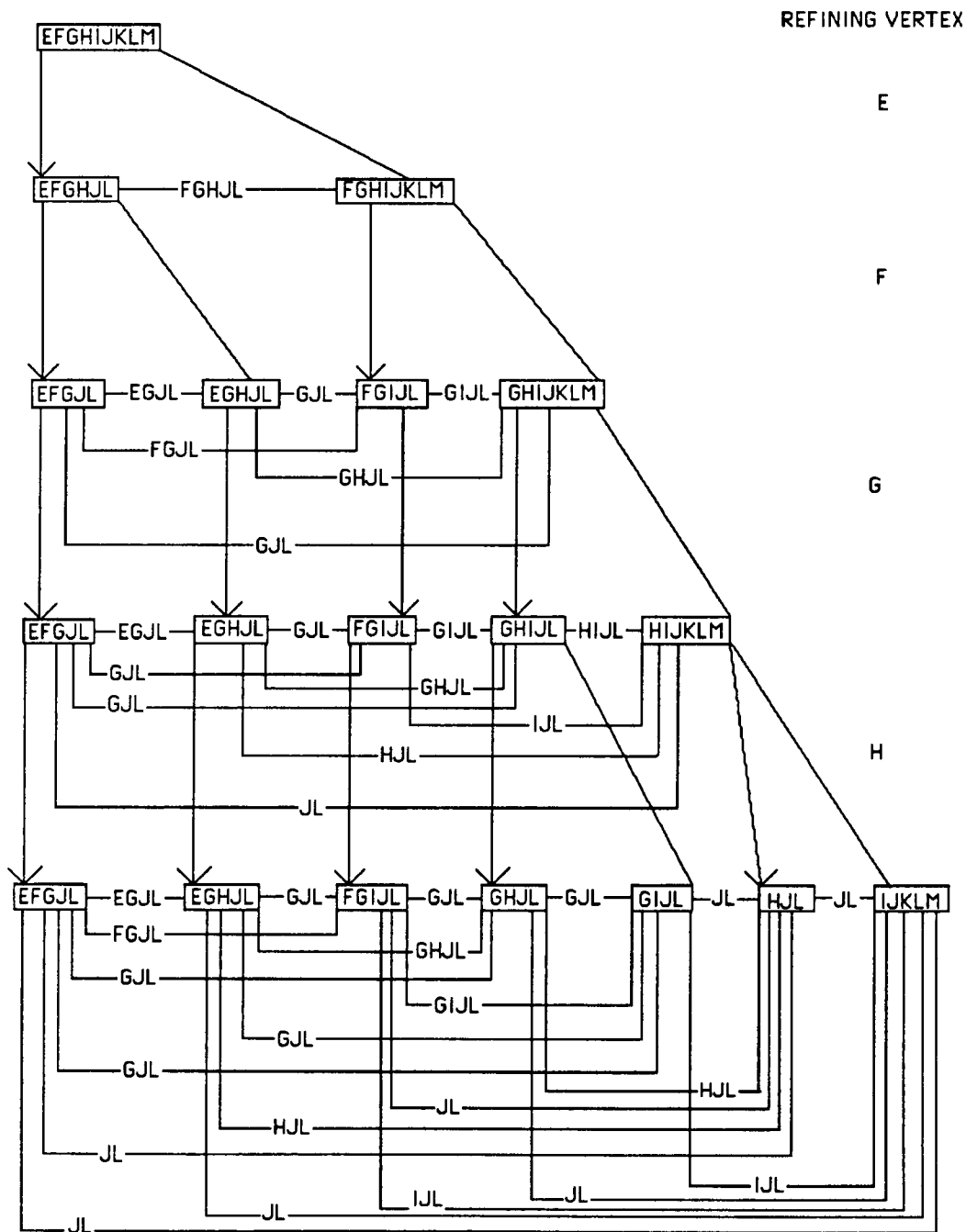


Figure 3.2: Partial Maximal Clique Generation with Clique Graph Edges

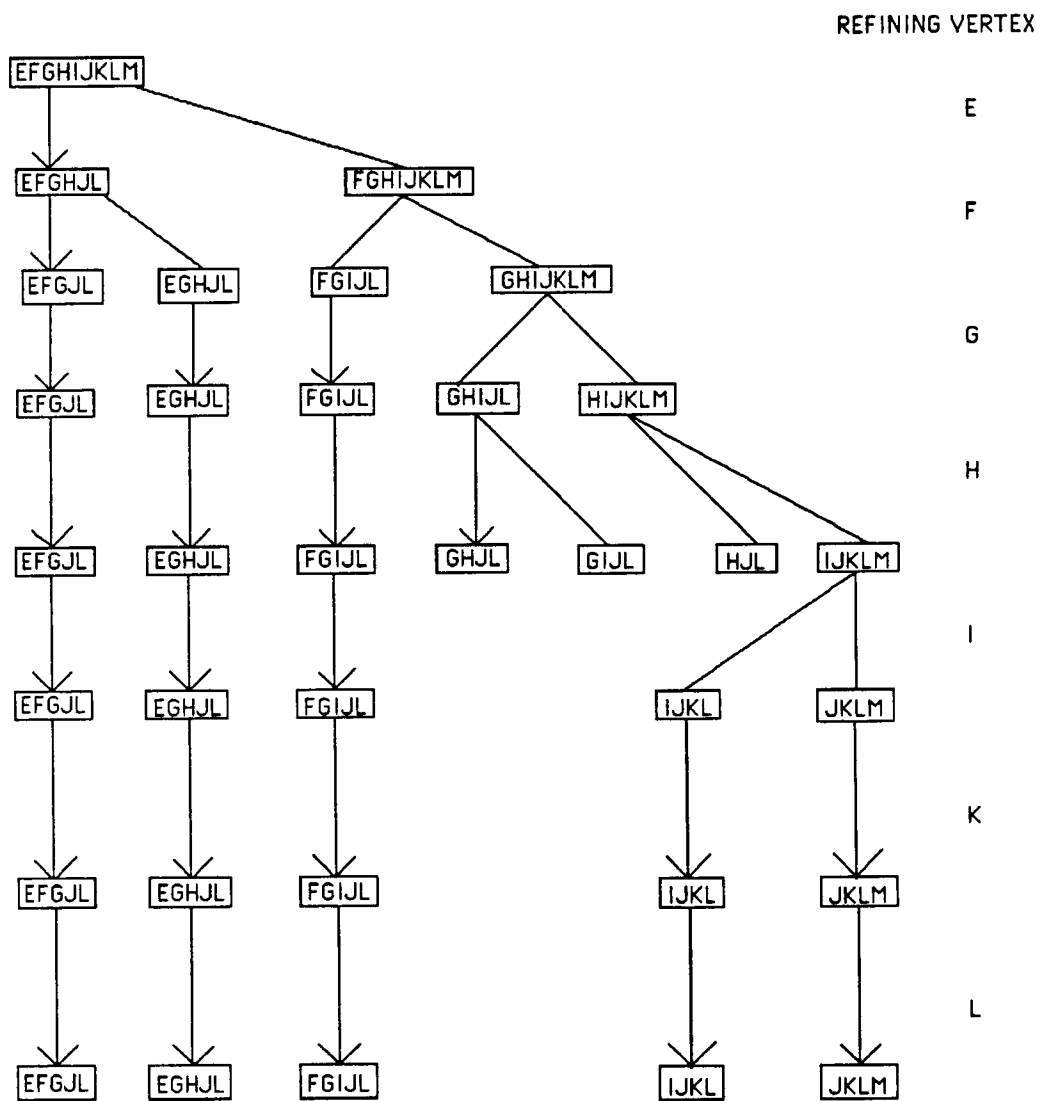


Figure 3.3: Clique Generation

It should be noted that it is possible to find maximal cliques in parallel without generating the duplicates inherent in the process used above. To do so we need only restrict processor p to finding cliques that it owns. This can be accomplished by using only the portion of the adjacency vectors that is less than or equal to p . However, by not developing all cliques containing a vertex, we lose the ability to build a clique graph. In addition, determining the number of cliques that a particular vertex belongs to would be significantly harder.

3.2 Maximal Clique Algorithm

The algorithm for developing the maximal cliques is given below.

Algorithm: MaxClique.

Input: Chordal Graph G .

Output: The set of maximal cliques for G and the corresponding edges in G_C .

Method: Initially, each processor p is given the adjacency vector for p and the adjacency vectors for each $v \in Adj(p)$. The processor for vertex p will maintain a candidate clique list (CCL) and a list of candidate edges for each candidate clique. The candidate clique list is refined by each vertex $v \in Adj(p)$. The list of candidate cliques remaining after all refinements is the list of maximal cliques to which p belongs. Details are given in figure 3.4.

The majority of the work in the algorithm is performed when a refinement occurs (steps M7 - M34). Briefly, if the result is not equal to the original candidate clique then the following occurs:

- place the result clique in the candidate clique list before the current clique,
- give this result clique a copy of the original cliques edges, update the edges and if the clique is equal to any of the edges, remove the clique, the edges, and an edge in each of the later cliques,

- if the clique was not removed then update the edges in each of the later cliques in the clique list and update the edgepointers in each of the earlier cliques in the clique list,
- remove the refinement vertex from the current clique,
- update each of the edges and if the clique is equal to any of the edges, remove the clique, the edges and an edge in each of the later cliques,
- if the clique was not removed then update the edges in each of the later cliques in the clique list and update the edgepointers in each of the earlier cliques in the clique list.

3.3 Example of Maximal Clique Development

In this section, the algorithm presented in the preceding section will be used to develop maximal cliques for the chordal graph shown in Figure 3.1. The graph G has 13 vertices and thus 13 processors are used to find the maximal cliques for this graph. Each processor p is given a copy of adjacency vector p and an adjacency vector for each $v \in Adj(p)$. Let p in this example be vertex J . Vertex J has as an adjacency set $\{E, F, G, H, I, J, K, L, M\}$. Each of these vertices is in at least one maximal clique with J . The set $\{J\} + Adj(J)$ is the first candidate clique vector. The processing begins by *anding* this vector with the adjacency vector for E , since E is the lexicographically greatest vertex in the adjacency vector of J .

(- - - - EFGHIJKLM)	CC (current clique)
(- - CDEFGH - J - L -)	AV (adjacency vector of E)
(- - - - EFGH - J - L -)	result

```

M1:  receive adjacency vectors in lexicographical order and store owner of each
      vector (refinement vertex)
M2:  cliquecount  $\leftarrow$  1
M3:  AV is first element in adjacency vector list
M4:  while adjacency vector list not empty do
M5:      CC is first element in candidate clique list
M6:      while not the end of candidate clique list do
M7:          if refinement vertex is in CC then do
M8:              result  $\leftarrow$  and of CC and AV
M9:              if result  $\neq$  CC then do
M10:                  cliquecount  $\leftarrow$  cliquecount + 1
M11:                  add result to candidate clique list before CC
M12:                  CC = CC - refinement_vertex
M13:                  result gets copy of edgelist and edgepointers of CC
M14:                  for result and CC do
M15:                      for each edge  $\in$  edgelist do
M16:                          update edge
M17:                          if edge = result/CC then do
M18:                              remove result/CC and edgelist
M19:                              cliquecount  $\leftarrow$  cliquecount - 1
M20:                          end if do
M21:                      end for do
M22:                  end for do
M23:                  if both result and CC were removed then do
M24:                      remove edge from each clique that appears later in CCL
M25:                  else do
M26:                      update edgelist of each clique that appears later in CCL
M27:                      update edgepointer list of every vertex that appears in CCL
M28:                  end else do
M29:                  if neither result nor CC removed then do
M30:                      add an edge to edgelist of CC
M31:                      add an edge pointer to edgepointer of result
M32:                  end if do
M33:              end if do
M34:          end if do
M35:          CC is next element of candidate clique list
M36:      end while do
M37:      AV is next element of adjacency vector list
M38:  end while do

```

Figure 3.4: Algorithm: MaxClique

There are two simple ways to check for equality of result and the original candidate clique. First, the two could be *exclusive-ored*. If the result is 0, then the two vectors are equal. Second, the result could be subtracted from the original candidate clique. If the answer is 0, then the vectors are equal.

If the vectors are equal then the candidate clique list needs no alteration. If they are not, then the result is put in the list before the current candidate clique, and the current clique is altered by removing the refinement vertex from the current clique. Thus the original candidate clique list was $(---EFGHIJKLM)$, and after refinement by the adjacency vector of E there are two candidate cliques in the list: $(---EFGH-J-L-)$, $(---FGHIJKLM)$. The first clique in this list is the result of the *and*, and the second is the original candidate clique minus the refinement vertex E . The second clique also retains the original edge list which is empty. In addition, in step M13 a copy of this edge list is given to the first clique. Then in steps M30 and M31 an edge is added to *edgelist* of the second clique, and a pointer to this edge is added to the first clique's *edgepointer* list. The edge is simply the result of the *and* minus the refinement vertex E . So the edge added to the second candidate clique's *edgelist* is $(---FGH-J-L-)$.

In Figure 3.2, the first level represents the original candidate clique. The second level shows the candidate clique list after refinement by the adjacency vector of E . There are no more elements in the candidate clique list, at this point, so the loop of M4 is complete. AV becomes the next adjacency vector in the adjacency vector list, which is the adjacency vector for vertex F . This adjacency vector is used to refine both of the existing candidate cliques. The adjacency vector is first *anded* with the first clique in the candidate clique list.

$(---EFGH-J-L-)$	CC (current clique)
$(---DEFG-IJ-L-)$	AV (adjacency vector of F)
$(---EFG--J-L-)$	<i>result</i>

Since the result is not equal to the original candidate clique, this clique

is placed prior to the original candidate clique and given a copy of its edges and edgepointers. The copied *edgelist* is empty while the *edgepointer* list contains one element. The refinement vertex, F , is removed from the original candidate clique, and an edge is added to its *edgelist* which is *result* minus the refinement vertex. An edge pointer is also added to *result*'s *edgepointer* list to point to this edge, and for each *edgepointer* in the edgepointer list, the edges in edgelists are updated.

CC then becomes the second clique in the candidate clique list, and it is *anded* against AV which is still the adjacency vector for F .

$$\begin{array}{ll} (- - - - - FGHIJKLM) & CC \text{ (current clique)} \\ (- - - DEFG - IJ - L-) & AV \text{ (adjacency vector of } F) \\ (- - - - - FG - IJ - L-) & result \end{array}$$

Again *result* is not equal to the original candidate clique, CC . Consequently, *result* is placed in the candidate clique list prior to CC and the refinement vertex, F is removed from CC . The result gets a copy of the *edgelist* and *edgepointer* list of CC , and these lists are updated. A new edge is added to *edgelist* for CC , and a pointer pointing to this edge is added to the *edgepointer* list of *result*. The third level of Figure 3.2 shows the candidate clique list after refinement by the adjacency vector of F .

Refinement continues until all adjacency vectors have been used to make a refinement. At this point the candidate clique list contains all maximal cliques in which J is a member. Each clique will also have a list of edges to each clique which appears earlier in the list.

3.4 Time Complexity Analysis

The time complexity is evaluated for a single vertex/processor. The speedup, efficiency and cost are examined later in the paper.

The algorithm consists of 6 loops inside the while loop at M6 which is inside the while loop (M4). M1 is also a loop and will take d steps, where d is defined to be $d(p)$. All other steps take constant time. The inside loops are at M13, M14, M15, M24, M26, M27.

The loop at M14 is executed twice, once for *result* and once for *CC*. All other loops that appear inside the while loops are executed at most once for each clique in the candidate clique list for that iteration through the while loops. This follows from the fact that each candidate clique can have no more than one edge to each other candidate clique.

We claim that at the end of each iteration through the while loop at M6, the number of elements in the clique list is no more than the number of maximal cliques in G that contain p . To see this, note that any candidate cliques that might be subsets of other candidate cliques or non-maximal relative to the current candidate clique list are discarded as they are generated. Thus, at the end of any iteration through the while loop at M6, the candidate cliques are unique. Furthermore, using an argument similar to that used when proving that the maximal cliques were kept in order while being generated, it can be shown that each clique is differentiated by some unique initial segment of vertices in $Adj(p)$. This unique initial segment consist of all vertices in each candidate clique up to and including the current refinement vertex. Recall also that each candidate clique has a refinement vertex that no other candidate clique has, so for each candidate clique the set of vertices up to and including the refinement vertex is unique. Since each such initial segment corresponds to a set of completely connected vertices, each distinct candidate clique must have at least one distinct maximal clique containing this initial segment. Thus, the number of candidate cliques at the end of an iteration through the while loop at M6 must be no more than the number of maximal cliques containing p .

Moreover, the same formulation of maximal cliques that proved that there are at most n maximal cliques in G also proves that there are at most d maximal cliques containing p in G . This formulation was $C_v = \{v\} \cup X_v$ where $X_v = \{x \in Adj(v) | \sigma^{-1}(v) < \sigma^{-1}(x)\}$ [14], for some PEO σ of the vertices. A vertex w can only appear d different ways in this formula because w can only have one possible set of adjacent vertices with higher PEO numbers, and w can only appear in d minus the number of vertices that appears in the set of adjacent vertices with higher PEO numbers different X_v sets. Thus, all inside loops, except the trivial loop at M14, can be executed a maximum of d times.

To determine the total complexity, observe that the loop at M4 must also be executed d times because p has d adjacency vectors, and once M6 has been entered, it executes a maximum of d times because of the argument above. Thus, for each iteration of the loop at M4, the loop at M6 is executed a maximum of d times. This means that the loop at M6 is executed a maximum of d^2 times. The inner five loops are executed a maximum of d times, for each iteration of the loop at M6. Since the M6 loop is executed a maximum of d^2 times, the inner five loops are executed a maximum of d^3 times.

This gives a total running time of the algorithm equal to $O(D^3)$ where D is the maximum degree of all the vertices. This time complexity is based on the bit vector model of computation[1]. In this model, logical operations on vectors are considered to be constant time operations. Thus, the $O(D^3)$ complexity here is $O_{BV}(D^3)$. However, as noted in [1], "most computers do logical operations on full-word bit vectors in one instruction cycle. Thus bit vectors of length 100 could be manipulated in three or four steps." If this maximal clique algorithm were implemented on a machine with a b bit word length, then a logical *and* could be performed in n/b steps. For a general machine, the actual running time of the parallel maximal clique algorithm is $O(D^3 n/b)$.

3.5 Summary

In this chapter, we have given a parallel algorithm for developing the set of maximal cliques of a chordal graph. The algorithm also produces the clique graph of the original graph. The algorithm runs in $O(D^3n/b)$ time on a distributed-memory machine using $O(n)$ processors. Furthermore, with no added implementation difficulties, the algorithm can be implemented on a machine with x processors. This would result in a time complexity of $O(D^3(n/b)(n/x))$.

In the next two chapters, the maximal cliques developed here will be used to develop a PEO and a minimum coloring respectively.

Chapter 4

Generating a PEO in Parallel

Another important aspect of chordal graphs is a PEO or Perfect Elimination Ordering. A PEO, as mentioned previously, is an ordering of the vertices of chordal graph G such that each vertex v_i is simplicial on the induced subgraph $G - \{v_j | j < i\}$. A simplicial vertex v is a vertex whose adjacency set is a clique.

The commonly used serial algorithms to find a PEO are LBFS, MCS and MCC. In [22], maximal cliques and a clique tree are used to find a PEO; [20] employs backward paths and an algorithm to refine these backward paths to find a PEO. In this paper, a PEO is found directly from the maximal cliques.

The first section of this chapter discusses the PEO algorithm. The second section presents algorithms for both the host and vertex processors. The third section goes through an example of how the PEO algorithm works, and the last section discusses the time complexity of the algorithm.

4.1 Discussion of Finding a PEO

The PEO found by this algorithm will use the information available at the end of the maximal clique algorithm. In fact, a certain amount of pipelining¹

¹By pipelining we mean that there is a certain amount of overlap between the execution of

is possible since a processor p can start its work of finding a PEO as soon as it has completed finding all maximal cliques for p . Recall that at the end of the maximal clique algorithm each processor p has a list of all the cliques to which p belongs and a count of those cliques. A Lemma will aid in the understanding of the PEO algorithm. Although this Lemma may have been proven elsewhere, for completeness, we include our version here.

Lemma 1 *Any simplicial vertex is a member of exactly one maximal clique.*

Proof (by contradiction): Assume simplicial vertex v belongs to two distinct maximal cliques. Then there must be vertices x and y such that $(x, v) \in E$ and $(y, v) \in E$ but $(x, y) \notin E$. But this contradicts the definition of a simplicial vertex. $\square$

A consequence of Lemma 1 is that we can find a PEO from the maximal cliques that have already been found. The PEO generation involves the use of a host processor to collect the ordering, and the n processors for the vertices of G . As vertices are discovered to be simplicial, they are sent to the host processor to be included in the PEO.

It is possible to determine if a vertex is simplicial by determining when it is a member of exactly one clique. Thus, a processor p whose clique count is 1, knows that p is a simplicial vertex and will send a message to the host processor indicating that p is ready to be included in the PEO. The host processor will receive the message and include p in the PEO. The processor p will also send a message to each $v \in C$ where C is the one clique in which p is a member. This message will inform each of these processors that p has been removed from the graph. As simplicial vertices are eliminated from the graph, maximal cliques are also going to be removed. A maximal clique will be removed from the graph when

the maximal clique algorithm and the PEO algorithm.

all its vertices have become simplicial (and thus are removed from the graph) or when the clique is no longer maximal. (In other words, when the clique becomes a subset of another maximal clique). Eventually all maximal cliques will be removed. This follows directly from the facts: (1) a simplicial vertex is a member of one maximal clique, and (2) a chordal graph must have a PEO.

It is not necessarily true that when all simplicial vertices have been removed from a maximal clique, that it is no longer a maximal clique. In Figure 4.1, ABC is a maximal clique and A is simplicial. When A is removed, however, it is not true that vertices B and C are in one less clique because BC is still a maximal clique on the induced subgraph $G - \{A\}$. So processors B and C cannot decrement their *cliquecount* when they receive a message from A indicating that A has been removed. They must first check to see if removing A results in one less clique in the induced subgraph.

When processor p receives a message indicating that a vertex v has been removed from the graph, it will perform a binary search on the maximal clique list to find the clique containing v . It should be possible to perform a binary search by determining whether a clique is lexicographically greater or less than the clique containing vertex v . When the clique containing v is found, v is removed from the clique. Recall that v can only be a member of one clique since it was simplicial. It now needs to be determined whether this new clique C is still maximal, and if the clique is still maximal, then its new place in the maximal clique array needs to be determined. It should be noted here that removal of any vertex from a clique can only make the clique lexicographically lower. Recalling that the maximal cliques are kept in reverse lexicographic order (greatest first), this implies that it is not possible for C to move towards the start of the list and thus, it is not necessary to search back in the list to find C 's new location.

So, when v 's maximal clique C is found, $C - \{v\}$ is compared against each

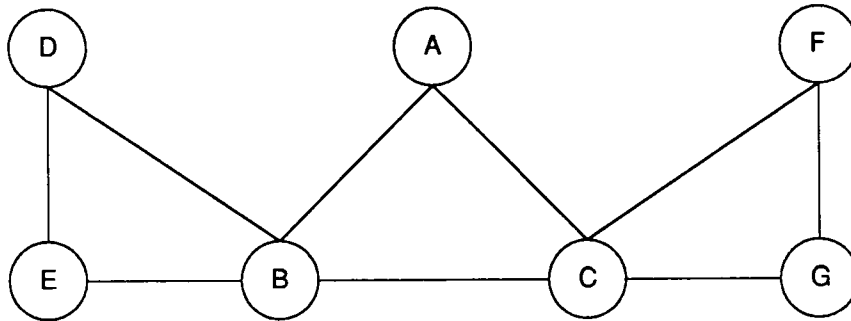


Figure 4.1: Removal of A does not remove a maximal clique

maximal clique which appears earlier in the list to determine if C is a subset of one of these maximal cliques. If it is, then C is discarded and each maximal clique which appeared later in the list is moved up. If C is not a subset of any maximal clique, then C must be compared to each maximal clique which appears after it in the list to determine its new position. There are three possibilities: C is lexicographically less than the maximal clique to which it is being compared, C is a subset of the maximal clique, or C is lexicographically greater than the maximal clique. If the first possibility is true, then C 's place in the maximal clique has been found. If the second possibility is true, then C is removed from the list, and every maximal clique that appears later in the list is moved up. If the last possibility is true then the maximal clique to which C is being compared is moved up in the list and C is compared to the next maximal clique.

If the clique C , from which v was removed, is not inserted back into the maximal clique list it means that the clique no longer exists on the induced sub-

graph. Processor p , therefore, will decrement its *cliquecount*. If the *cliquecount* is now 1 then p is now simplicial and can be sent to the host processor. A message will also be sent to each vertex in the remaining clique. This process continues for each processor until *cliquecount* = 1 and p sends a message to the host, at which time that processor has completed its work in PEO generation². The process will continue for the host processor until it has n messages, one from each processor. You should recall here that induced subgraphs are chordal, and chordal graphs have simplicial vertices. Therefore, this process will continue to completion.

4.2 PEO Algorithm

The following algorithm is performed by each processor p immediately following the maximal clique algorithm.

Algorithm: PEO

Input: For each processor p a list of all maximal cliques containing p and a *cliquecount*.

Output: A PEO on the host processor.

Method: Each processor will maintain an array of maximal cliques and a count of maximal cliques for the induced subgraph as vertices are removed. To begin, each processor p will check its *cliquecount*, and if it is 1 then p will send a message to the host and to every vertex remaining in the clique. Otherwise, p will wait to receive messages indicating that other vertices have been removed from the graph. When p receives a message indicating x 's removal, p deletes x from its appropriate clique and adjusts C 's maximal clique representation accordingly. This process continues until the *cliquecount* = 1, at which time p will send messages to the host and each member of the remaining clique. The details of the vertex processor algorithm are given in Figure 4.2.

²Simplicial vertices send a message to the host first and then to each vertex processor x in the last clique. The vertex processor x must then do some computation before it can send a message. It is therefore assumed that the messages will not get out of order. If this assumption were not reasonable for a given machine, relative time stamps can easily be added to the messages to ensure synchronization.

```

P1:  while cliquecount > 1 do
P2:      receive(wait) vertex removal message (v)
P3:      use binary search to find v in maximal clique C
P4:       $C \leftarrow C - v$ 
P5:       $MC \leftarrow$  the maximal clique before C
P6:      complete  $\leftarrow$  off
P7:      while not beginning of maximal clique array and not complete do
P8:          if  $C \subset MC$  then do
P9:              cliquecount  $\leftarrow$  cliquecount - 1
P10:             move up all maximal cliques that appeared after C
P11:             complete  $\leftarrow$  on
P12:         end if do
P13:         if not complete then  $MC \leftarrow$  clique before MC
P14:     end while do
P15:      $MC \leftarrow$  the maximal clique after C
P16:     while not end of maximal clique array and not complete do
P17:         if  $C \subset MC$  then do
P18:             cliquecount  $\leftarrow$  cliquecount - 1
P19:             move up MC
P20:             move up all maximal cliques that appeared after MC
P21:             complete  $\leftarrow$  on
P22:         end if do
P23:         if  $C < MC$  then complete  $\leftarrow$  on
P24:          $MC \leftarrow$  the maximal clique after MC
P25:     end while do
P26: end while do
P27: send(wait) message to host (p)
P28: send message to each  $v \in E(p)$ 

```

Figure 4.2: Algorithm: Vertex Processor PEO

Since, each vertex is a member of exactly one maximal clique when it is simplicial, and every chordal graph has a simplicial vertex, the process of sending messages will begin as soon as the maximal clique algorithm has finished for some vertex. Eventually all vertices will have become simplicial, all maximal cliques will be removed and all processors will have dropped out of the while loop at P1.

The host processor begins work when it receives its first message. It will receive a message from each processor in turn indicating that the vertex is ready to be included in the PEO. The host will keep a list of vertices in PEO order. The details of the host processor algorithm are given in Figure 4.3.

4.3 Example of PEO Development

Figure 4.4 shows the list of cliques developed by each processor during the maximal clique algorithm on the chordal graph of Figure 3.1. Obviously, vertices A, B, H and M are simplicial. Thus, these vertices will send messages to the host and to each vertex in the maximal clique. A sends messages to the host, B and C . B sends messages to the host, A and C . H sends messages to the host E, G, J and L . M sends messages to the host and J, K and L . All these messages are sent concurrently.

Each processor that receives a message will use a binary search to find the vertex in the maximal clique array. For example, J will first receive a message from either H or M . If H is received first, a binary search will be used to find $EGHJL$ in the maximal clique array. H will be removed from the clique C , and then C will be compared to the first clique before it in the array. C is a subset of the clique, so C will be removed from the array, and everything after C will be moved up in the array. The *cliquecount* will also be decremented and since it is not yet 1, J will continue to wait for vertex removal messages. J may receive M

```

H1:  start up  $n$  processors and distribute adjacency vectors { work for maximal cliques}
H2:   $msgcount \leftarrow 0$ 
H3:  while  $msgcount < n$  do
H4:    receive(wait) a vertex message ( $v$ )
H5:    add  $v$  to PEO list
H6:     $msgcount \leftarrow msgcount + 1$ 
H7:  end do while

```

Figure 4.3: Algorithm: Host PEO

Vertex/Proc	A	B	C	D	E	F	G
Count	1	1	2	2	4	3	3
Clique	ABC	ABC	ABC	CDE	CDE	DEFG	DEFG
			CDE	DEFG	DEFG	EFGJL	EFGJL
					EFGJL	FGIJL	EGHJL
					EGHJL		FGIJL

Vertex/Proc	H	I	J	K	L	M
Count	1	2	5	2	5	1
Clique	EGHJL	FGIJL	EFGJL	IJKL	EFGJL	JKLM
		IJKL	EGHJL	JKLM	EGHJL	
			FGIJL		FGIJL	
			IJKL		IJKL	
			JKLM		JKLM	

Figure 4.4: Clique list and count for each processor

next. Again, the clique C , after the removal of M , will be a subset of a maximal clique. So C is removed and the *cliquecount* decremented. Messages will also be received from K and L , and in each case C is a subset and will be removed. At this point the *cliquecount* = 1, and J will send a message to the host to indicate that p is ready to be included in a PEO.

The exact PEO will depend on the synchronization of the processors. One reasonable sequence of events would be for all vertices in exactly one clique to be first in the PEO. The next group might be any vertex which was in exactly one clique after removal of all cliques containing vertices in the first group and so on. A possible PEO generated this way is:

ABHMCKDIEFGJL

4.4 Time Complexity Analysis

In this section, the time complexity of the algorithm for each vertex processor is examined followed by an analysis of the complexity of the host processor. Questions of speedup and efficiency are, again, postponed until later.

The binary search for the PEO algorithm (step P3) requires $\log c$ time where, c is the number of cliques of which p is a member. Clearly, $c \leq d$. The loops at P7 and P16 take, in the worst case, c total time for one iteration through P2, since every maximal clique may need to be examined. The outside loop at P2 must be executed at least $c - 1$ times. However, each message received will not necessarily remove a maximal clique so it is possible that d messages will be required to remove the $c - 1$ cliques. Steps P8 and P17 take constant time since it is possible to check if $(C - MC) = 0$. Thus the time complexity for the PEO algorithm on each processor is $O(d * c)$ plus some possible wait time. *Wait time* is the time an algorithm spends waiting for messages. Because of the accumulated

effect of vertices becoming simplicial, the possible total wait time for any vertex is $O(D * C)$, where D and C are the degree and clique count of the “worst” vertex, respectively. So, the time complexity is still $O(d * c)$.

Note, however, that the processor that develops maximal cliques for a simplicial vertex will, in all likelihood, complete this process sooner than processors working on vertices with multiple cliques. It can also be stated that for the “average” chordal graph there will be a tendency to pipeline the messages for the PEO algorithm. By pipelining we mean that there will be a tendency for messages from simplicial vertices to go to vertices with a few cliques and then these vertices become simplicial and send messages to vertices with a few more cliques etc. When this pipelining effect is present, the idle wait time for most processors will be negligible because processors with few maximal cliques will determine that they are simplicial and send messages to processors that are continuing to find cliques. When these processors have finished finding maximal cliques, simplicial vertex messages will be available for them to process. However, this is not always the case. See Figure 4.5 for an example of a graph where the pipelining is limited. Vertices B and C would need to wait until F became simplicial; D and E would need to wait until G became simplicial; and A would need to wait until either B and C became simplicial or D and E became simplicial. A, B, C, D and E are all in 2 cliques but, to make the example worse, F could be made a member of any number of cliques.

Because of the pipelining, the host processor will not have to wait until all maximal cliques have been developed to start receiving messages from the vertex processors about the PEO. The host will receive n messages, hence its time complexity is $O(n)$. However, the host processor should finish receiving all messages soon after all vertex processors are complete.

On the graph in Figure 3.1 every simplicial vertex, except M , sends a

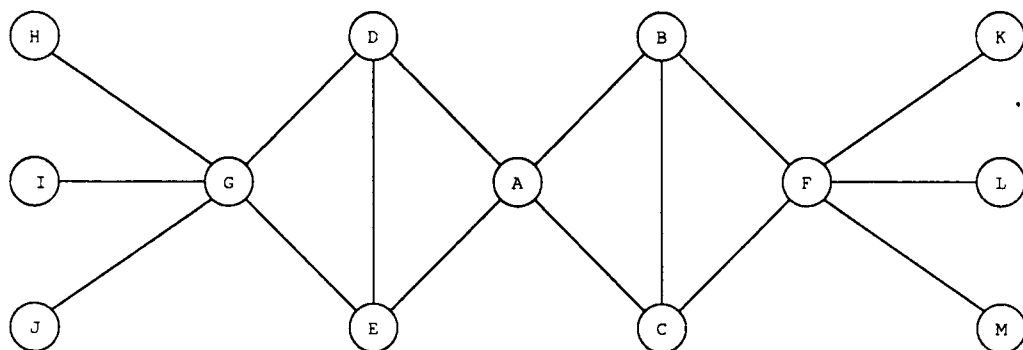


Figure 4.5: Example of Chordal Graph with little pipelining

message to a processor with just two cliques. *A* and *B* both send a message to *C*, and *J* sends a message to *I*. *C* sends a message to *E*, and *I* sends a message to *L*, both of which have just two cliques. The pipelining tendency is apparent.

Another important aspect of the complexity of this algorithm is the communication. The nice part of the maximal clique algorithm is that after receiving the adjacency vectors, each processor can work without interruption to develop the cliques. When work can be divided so that each processor can work with no communication, then the benefits of parallelism are at their peak, as long as the extra work required due to using no communication is not prohibitive. For the PEO algorithm, a message will be sent every time a vertex appears in a maximal clique. So, the total number of messages is equal to $\sum_{i=1}^c |C_i|$, where the set $\{C_1, C_2, \dots, C_c\}$ is the set of all maximal cliques. The pipelining that is possible with the combination of the maximal clique and PEO algorithm may reduce some of the bus contention but it cannot guarantee it. One area for further research is

to consider how to reduce the communication required by the PEO algorithm.

4.5 Summary

In this chapter, we presented an algorithm to generate a PEO. It is not necessary that every processor have found all maximal cliques before a processor can start on the PEO algorithm. Processor p can start the PEO algorithm as soon as it has found all the maximal cliques to which p belongs. The PEO algorithm presented here has a worst case time complexity of $O(d * c)$ plus any necessary wait time, and uses $n + 1$ processors.

As in the maximal clique algorithm, the work here can easily be divided among a constant, say x , processors with no sacrifices to the worst case time complexity (i.e., worst case time complexity would be $O(d * c(n/x))$).

In the next chapter, the PEO generated here is used to find a minimum coloring.

Chapter 5

Generating a Minimum Coloring in Parallel

To color a graph G , each vertex, v is assigned a color such that no vertex $u \in Adj(v)$ has the same color as v . A minimum coloring is a coloring that uses the minimum number of colors possible. Coloring of graphs has many applications including VLSI design and resource allocation.

In this chapter, a parallel algorithm that begins with a PEO and produces a minimum coloring is developed. The first section discusses coloring in parallel. The second section presents the coloring algorithm. The third section provides an example of a coloring, and the last section examines the time complexity of the algorithm.

5.1 Discussion of Parallel Coloring

Unlike the PEO algorithm, the coloring algorithm cannot begin until the PEO generation has been completed. The processors which were used to develop the maximal cliques can now be used for the coloring algorithm. However, here each processor will handle a color instead of a vertex. This will avoid the expense of

starting up new processors. Obviously it is not possible for more than n processors to be required, since at most n colors will be required.

The method used here to color a graph is a parallel version of Gavril's serial coloring algorithm[11]. As with Gavril's algorithm, vertices are colored in reverse PEO order. The host finds vertices that can be colored with the first color and pass vertices that cannot be colored with color 1 to the next processor representing color 2. When processor 2 determines that a vertex cannot be colored with color 2, it must pass the vertex on to processor 3, and so on. Consequently, either the host must inform processor 2 to which processor it should pass vertices or some scheme must be understood. One such scheme is that the processor that handled vertex A represents color 2, the processor that handled B will represent color 3 and so on. We assume this scheme is used.

The host begins by taking the vertices in reverse order. The last vertex in the PEO will be assigned color 1. The next to last vertex, x , is then selected. The host must determine whether x can have the same color as every vertex that already has color 1. To do this, the host keeps a conflict vector that is n bits long like the adjacency vectors, and where a bit is 1 if it has been colored 1 by the host and is 0 otherwise. To determine whether x is adjacent to any vertex, $v \in \{v | v \text{ colored } 1\}$. The host will perform an *and* between the adjacency vector of x and the conflict vector. If the result is not equal to 0, x cannot be colored 1 and must be passed to processor 2. Since processor 2 will have to verify whether x is adjacent to any vertex colored 2, the host will need to send the adjacency vector for x . To ensure synchronization, a timestamp is sent with the message. In other words, the first message sent by the host is stamped 1, the second 2 etc. Thus, processor 2 knows if it has received messages out of order and can wait until it had received the proper message. Continuing with the coloring process, processor 2 determines whether x can be colored 2. Again, this will be done by performing

an *and*. If x cannot be colored 2, then processor 2 sends x to processor 3. So each processor p in turn performs an *and* between the adjacency vector of x and the conflict vector to determine if x can be colored p and sends x and its adjacency vector to the next processor if it cannot. For each processor, if the result of the *and* is 0, x can be colored with this color and the appropriate position in the conflict vector is set to 1.

The number of colors required is equal to the size of the largest maximal clique. But this number was never determined, and it is not a trivial task to find it. To achieve this, each processor would need to determine the largest clique it owned and then these would have to be combined and compared. If the size of the largest maximal clique is known then the number of colors, k , (the chromatic number) is known. Since it is known that the algorithm will produce a minimum coloring, the processor $k - 1$ would not start up another processor. Instead, processor $k - 1$ would simply keep two colors. Any vertex that could not be colored $k - 1$ would receive color k .

It is not necessary to know the chromatic number, however. We will allow each processor p to start another processor if it finds a vertex x which it cannot color with p . Processor $k - 1$ then will start up processor k when it finds a vertex which cannot be colored $k - 1$. Processor k will check the adjacency of each x it receives but will not receive any adjacent vertices, and thus, will not start up another processor.

The host continues until every vertex in the PEO has been examined and either colored 1 or sent to processor 2. When it has completed every vertex, it will send a message to processor 2 that all vertices have been colored, and processor 2 passes the message along. In addition to producing a coloring at this point, we also know the maximum clique size which is k .

5.2 Coloring Algorithm

The following is the algorithm the host will perform for the coloring. This will follow immediately after all n vertices have been received for the PEO.

Algorithm: Host Coloring.

Input: A PEO for G .

Output: A coloring distributed across k processors.

Method: The host maintains a PEO list and a conflict vector CV which contains the vertices which can be colored 1. The host processes vertices in reverse order. The host determines for each vertex processed whether it can be colored 1 and if it can, the vertex is added to the conflict vector. Otherwise, the vertex is passed to processor 2. The details are given in Figure 5.1.

The algorithm performed by each color processor is similar except that instead of using vertices from the PEO, the vertices are sent to the processor from the next lowest color. The algorithm terminates when a termination message is received by processor k . See Figure 5.2 for details.

5.3 Example of Coloring

Figure 5.3 shows how the coloring proceeds and the way vertices are passed from color to color for the graph in Figure 3.1. The host processor places L in its conflict vector. It then checks whether J is in $Adj(L)$. It does this by performing an *and* between the conflict vector and the adjacency vector of J . L is adjacent to J , so J is sent to processor 2, which puts J in its conflict vector. The host, meanwhile, determines whether the next vertex in the PEO, G , is adjacent to anything in its conflict vector. G is adjacent to L so G is sent to processor 2. The host then determines whether F is adjacent to anything in the conflict vector while processor 2 determines whether G is adjacent to anything in its conflict

```

C1:  remove last vertex in PEO and place in  $CV$ 
C2:   $AV \leftarrow$  adjacency vector of last vertex  $v$  in PEO
C3:  while PEO not empty do
C4:     $result \leftarrow CV$  anded with  $AV$ 
C5:    if  $result = 0$  then do
C6:      add  $v$  to  $CV$ 
C7:    end do it
C8:    else do
C9:      send  $v$  and  $AV$  to processor 2
C10:   end else do
C11:    $AV \leftarrow$  adjacency vector of last vertex  $v$  in PEO
C12: end while do
C13: send processor 2 completion message

```

Figure 5.1: Algorithm: Host Coloring

```

D1:  receive message
D2:  add message vertex to  $CV$ 
D3:  receive message
D4:  while message  $\neq$  completion message
D5:     $AV \leftarrow$  received adjacency vector
D6:     $result \leftarrow CV$  anded with  $AV$ 
D7:    if  $result = 0$  then do
D8:      add  $v$  to  $CV$ 
D9:    end do it
D10:   else do
D11:     send  $v$  and  $AV$  to processor  $p + 1$ 
D12:   end else do
D13:   receive message
D14: end while do
D15: send processor  $p + 1$  completion message

```

Figure 5.2: Algorithm: Processor Coloring

vector, and so on. When the host processor has examined all vertices in the PEO it sends a completion message to processor 2, which passes the message to 3 etc. Figure 5.4 shows the final color sets and Figure 5.5 shows the colored graph.

5.4 Time Complexity

The time required to complete the coloring after finding a PEO is dependent on the time required by the host processor plus the extra time needed to finish processing at the other processors after the host has sent vertex n along. The loop at C3 for the host will take n time since there are n elements in the PEO. Thus the running time of the algorithm is $O(n)$ for the host. The other processors will terminate soon after the host has finished. In the worst case, each other processor will also take $O(n)$ time after receiving its messages. Thus, the entire time is $O(n)$. It should also be noted that the message passing can avoid bus contention if the processors are arranged so a processor with color p is adjacent to a processor with color $p + 1$.

5.5 Summary

We have presented a parallel minimum coloring algorithm that requires as input the PEO produced by the algorithm in the previous chapter. After receiving the PEO, our coloring can run in $O(n)$ time on $O(k)$ processors, where k is the chromatic number.

In the next chapter, we summarize the suite of our algorithms presented in the last three chapters and briefly discuss some open problems.

Host		2		3		4		5
L								
~~J~~	→	J						
~~G~~	→	~~G~~	→	G				
~~F~~	→	~~F~~	→	~~F~~	→	F		
~~E~~	→	~~E~~	→	~~E~~	→	~~E~~	→	E
~~I~~	→	~~I~~	→	~~I~~	→	~~I~~	→	I
D								
~~K~~	→	~~K~~	→	K				
~~C~~	→	C						
~~M~~	→	~~M~~	→	~~M~~	→	M		
~~H~~	→	~~H~~	→	~~H~~	→	H		
B								
~~A~~	→	~~A~~	→	A				
complete	→	complete	→	complete	→	complete	→	complete

Figure 5.3: Messages and final coloring

$Color1 = \{L, D, B\}$
 $Color2 = \{J, C\}$
 $Color3 = \{G, K, A\}$
 $Color4 = \{F, M, H\}$
 $Color5 = \{E, I\}$

Figure 5.4: Color Sets

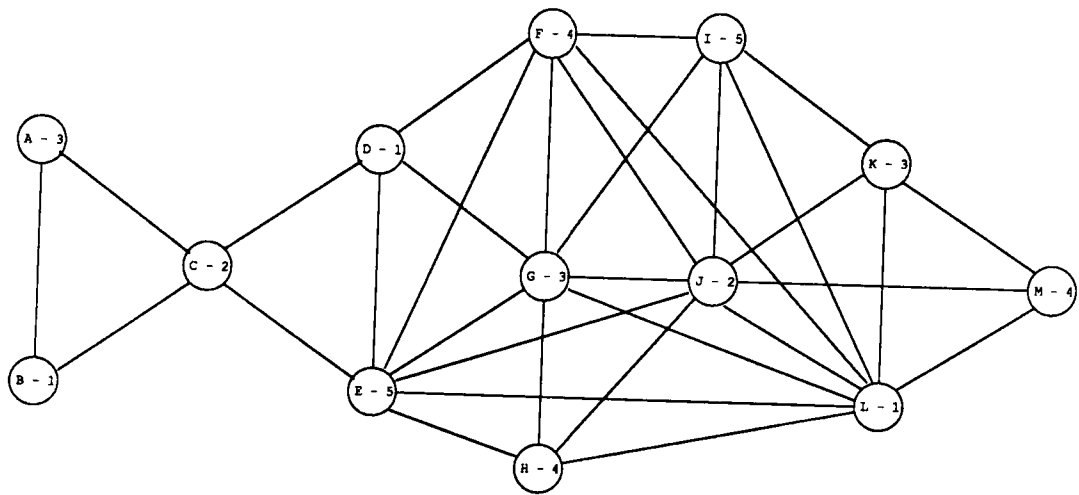


Figure 5.5: Colored graph

Chapter 6

Conclusion

6.1 Summary

In this paper, three algorithms have been developed. Maximal cliques were found by refining candidate cliques using adjacency vectors. Each processor p found every clique in which p was a member. Thus, processor p knew whether vertex p was simplicial. The PEO algorithm used this fact to find a PEO. Any processor that had a simplicial vertex sent it to the host. Other processors then updated their maximal clique lists and determined whether any cliques had been absorbed by other maximal cliques. In this way, processor p determined when vertex p was simplicial and when it was, a message was sent to the host. The coloring algorithm started after a PEO was found. In this algorithm, a vertex v was passed from one processor p to another until there were no vertices in the color list of p that were adjacent to v .

Figure 6.1 provides a summary of the algorithms presented, their running time, number of processors required, sequential running time, speedup, efficiency, and cost. The maximal clique algorithm efficiency depends on the maximum degree of the graph. For sparse graphs, the figure is good. However, as the density of the graph approaches that of a complete graph, the algorithm approaches the

Algorithm	Parallel Running Time	Number of Processors	Serial Running Time
Max Clique	$O(D^3)$	n	$O(n + m)$
PEO	$O(d * c)$	n	$O(n + m)$
Coloring	$O(n)$	k	$O(n^2)$

Algorithm	Speedup	Efficiency	Cost
Max Clique	$(n + m)/D^3$	$(n + m)/D^3n$	nD^3
PEO	$(n + m)/d * c$	$(n + m)/d * cn$	$d * cn$
Coloring	n	n/k	nk

Figure 6.1: Algorithm Statistics

time of a serial algorithm. The efficiency of the PEO algorithm is dependent on the size of c and d which is the maximum number of cliques and degree of a vertex. Depending on the graph, this figure could be very good. The coloring algorithm efficiency is dependent on the number of colors k , but it is clear that this figure is greater than 1 and so the speedup is superlinear.

Figures 6.2 through 6.4 compare various parallel algorithms in the literature to those presented here. In almost all cases, the time complexity of the other algorithms is better than the algorithms presented here if D and c are large. For sparse graphs, however, our algorithms perform better than the others. Moreover, time complexity is not the only aspect of a good parallel algorithm. Perhaps the best way to compare these algorithms is by their cost. Again, it is difficult to make any hard and fast statements about these figures, but for all algorithms the cost of the algorithms presented in this paper compares favorably with the other algorithms.

Another important benefit of the algorithms presented here is that the number of processors required is, most often, orders of magnitude less than those required by the other algorithms. Furthermore, the algorithms presented here are

Paper	Running Time	Number of Processors	Cost
This Paper	$O(D^3)$	n	nD^3
[22]	$O(\log^3 n)$	n^4	$n^4 \log^3 n$
[18]	$O(\log^2 n)$	n^3	$n^3 \log^2 n$

Figure 6.2: Maximal Clique Algorithm Comparison

Paper	Running Time	Number of Processors	Cost
This Paper	$O(d * c)$	n	ndc
[20]	$O(\log^2 n)$	$n + m$	$(n + m) \log^2 n$
[22]	$O(\log^2 n)$	n^2	$n^2 \log^2 n$

Figure 6.3: PEO Algorithm Comparison

Paper	Running Time	Number of Processors	Cost
This Paper	$O(n)$	k	nk
[20]	$O(\log^2 n)$	m	$m \log^2 n$
[18]	$O(\log n)$	n^2	$n^2 \log n$
[22]	$O(\log^2 n)$	n^3	$n^3 \log^2 n$

Figure 6.4: Minimum Coloring Algorithm Comparison

more efficient than the other algorithms on machines with a constant number of processors. This can be seen from the fact that, in almost all cases, the costs of the algorithms presented in this paper are definitely lower, and in all other cases the costs will be lower or comparable than the other parallel algorithms on chordal graphs. Since the cost of an algorithm is the total amount of computation time, a lower cost will result in faster computation on a fixed number of processors. The approach, when developing these algorithms, was to develop algorithms that made more efficient use of the resources provided by commercially available parallel processors. Our algorithms do this.

When assigning more than one vertex to a processor, there are several approaches which could be used. One possible approach would be to assign several vertices which are adjacent to a processor at the very start of the algorithms. It might be possible, using this approach, to limit some of the duplication of maximal cliques. Another approach, which might improve the load balance, would be to have the host distribute one vertex per processor. Then, when a processor finishes processing that vertex, it would ask the host for another vertex.

The granularity of the algorithms presented in this paper is also better than the other parallel algorithms for chordal graphs. Each processor performs enough work to offset the cost of starting up the processor. There is also some pipelining of work which helps improve the efficiency of the parallel algorithm.

6.2 Future Work

One of the most promising areas of future work is to consider the construction of a clique tree immediately after finding all maximal cliques. The clique tree could be constructed at the same time that a PEO is found without much added time complexity. In [22], a clique tree was used to generate a PEO, but it should

be possible to generate a PEO and construct a clique tree at the same time. The clique tree can then be used to generate a minimum coloring. Figure 6.5 shows a clique tree for the graph in Figure 3.1. The vertices listed in the squares at each node are simplicial on the graph induced by removal of all simplicial nodes in cliques below. As maximal cliques are removed from the graph through the algorithmic process described in the PEO section of this paper, it should be possible to determine the paths of the clique tree and collect simplicial vertices as the tree is generated bottom-up. From the clique tree it is possible to generate a minimum coloring much faster than was done here, while keeping the number of processors small. The possibilities for additional parallelism in a bottom-up coloring process are also of interest. This would increase the potential for pipelining.

Another area for future work is to explore the parallel generation of a fill-in for a graph which is not chordal. Given an ordering of graph G , the *fill-in* generated by the ordering is defined as $\{(x, y) | (x, y) \notin E, \text{ and there is a path in } G \text{ from } x \text{ to } y \text{ including only } x, y \text{ and vertices which are ordered before both } x \text{ and } y\}$. The relation between PEOs, fill-in and Gaussian elimination is discussed in [28]. A PEO is termed a PEO because of its relation to fill-in. If a graph G is a chordal graph and an ordering of the vertices of G is a PEO, then the vertices of G can be removed in order of the PEO without producing any fill-in. By converting a general graph to a chordal graph using fill-in, heuristic solutions to general graph problems that are NP-Complete are found using the corresponding chordal graph.

It would also be profitable to explore the uses of the clique graph. The clique graph provides the most complete definition of a chordal graph, and it has a wide range of applications. It would also be of interest to explore the possibilities of using a clique graph or clique tree to maintain certain aspects of a dynamic graph. For instance, it would be interesting to consider the use of a clique graph to maintain a coloring of a dynamic graph.

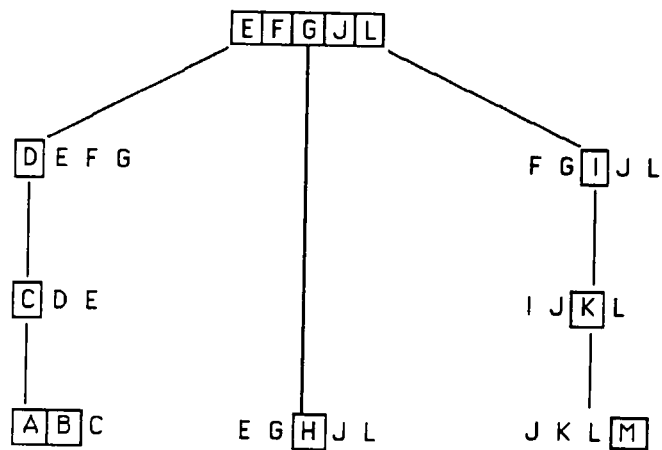


Figure 6.5: Clique Tree with Simplicial Vertices

BIBLIOGRAPHY

Bibliography

- [1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.
- [2] C. Beeri, R. Fagin, D. Maier and M. Yannakakis, "On the desirability of acyclic database schemes," *Journal of the Association for Computing Machinery* vol. 30, no. 3, pp. 205-212, 1974.
- [3] J.R.S. Blair, R. E. England, and M. G. Thomason, "Cliques and their separators in chordal graphs," Private Communication.
- [4] N. Chadrsekharan and S. S. Iyengar, "NC algorithms for recognizing chordal graphs and k trees," *IEEE Transactions on Computers*, Vol. 37, No. 10, pp. 1178-1183, 1988.
- [5] N. Christofides, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975.
- [6] E. G. Coffman Jr and R. L. Graham, "Optimal scheduling for two processor systems," *Acta Infomat.*, vol. 1, pp.200-213, 1972.
- [7] A. Edenbrandt, "Chordal graph recognition is in NC," *Information Processing Letters*, No. 24, pp. 239-241, 1987.
- [8] S. Even, *Graph Theory* Computer Science Press, Rockford, Maryland, 1979.
- [9] R. Fagin, A. O. Mendelzon and J. D. Ullman, "A simplified universal relation assumption and its properties," *ACM Trans. Database Systems*, vol. 7, pp. 343-360, 1982.
- [10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to NP-Completeness*, Freeman, San Francisco, 1978.
- [11] F. Gavril, "Algorithms for minimum coloring, maximum covering by cliques, and maximum independent set of a chordal graph," *SIAM Journal of Computing*, Vol. 1, No. 2, pp. 180-187, 1972.
- [12] J. A. George, "Nested dissection of a regular regular finite element mesh," *SIAM Journal of Numerical Analysis*, vol. 10, pp. 345-363, 1973.
- [13] J. R. Gilbert, D. J. Rose, A. Edenbrandt, "A separator theorem for chordal graphs," *SIAM J. Alg. Disc. Meth.* Vol. 5, No. 3, pp. 306-313, 1984.

- [14] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, London, 1980.
- [15] M. Goodman, O. Shmuli, "Syntactic characterizations of tree database schemes," *J. Assoc. Comput. Mach.*, vol. 30, pp. 767-786, 1983.
- [16] X. Guan and M. A. Langston, Time-Space Optimal Parallel Merging and Sorting, Private Communication.
- [17] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their applications to expert systems," *Journal of the Royal Statistical Society B*, vol. 5, no. 2, pp. 157-224, 1988.
- [18] C. Ho and R. C. T. Lee, "Efficient parallel algorithms for finding maximal cliques, clique trees, and minimum coloring on chordal graphs," *Information Processing Letters* Vol. 28, pp. 301-309, 1988.
- [19] T. C. H, "parallel sequencing and assembly line problems," *Operations Research*, vol. 9, pp. 841-848, 1961.
- [20] P. N. Klein, "Efficient parallel algorithms for chordal graphs," *Symposium on Foundations of Computer Science*, pp. 150-161, October 1988.
- [21] D. W. Matula, G. Marble, J. D. Isaacson, "Graph Coloring Algorithms," *Graph Theory and Computing*, R. C. Read Editor, Academic Press, New York, pp. 109-122, 1972.
- [22] J. Naor, M. Naor, A. A. Schaffer, "Fast parallel algorithms for chordal graphs," *Proceedings 19th Annual ACM Symposium on Theory of Computing*, pp. 355-364, 1987.
- [23] T. Ohtsuki, L. K. Chueng, T. Fujisawa, "Minimal Triangulation of a graph and optimal pivoting order in a sparse matrix," *Journal of Mathematical Analysis and Applications* Vol. 24, pp. 622-633, 1976.
- [24] C. Papadimitriou and M. Yannakakis, "Scheduling interval-ordered tasks," *SIAM J. Comput.*, vol. 8, pp. 405-409, 1979.
- [25] "Some applications of the clique trees to the solutions of the sparse linear systems," PhD Thesis, Clemson University, Clemson SC, 1986.
- [26] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers* McGraw-Hill, New York, 1987.
- [27] R. C. Read, "Algorithms in Graph Theory," in *Applications of Graph Theory*, R. J. Wilson Editor, Academic Press, New York, pp. 381-417, 1979.
- [28] D. J. Rose, "Triangulated Graphs and the Elimination Process," *Journal of Mathematical Analysis and Applications*, Vol. 32, pp. 597-609, 1970.
- [29] D. J. Rose, "A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations," in *Graph Theory and Computing*, ed. R. Read, pp. 183-217, Academic Press, New York, 1973.
- [30] D. J. Rose, R. E. Tarjan, G. S. Lueker, "Algorithmic aspects of vertex elimination on graphs," *SIAM Journal of Computing*, Vol. 5, No. 2, pp. 266-283, 1976.

- [31] R. Sethi, "Scheduling graphs on two processes," *SIAM Journal of Computing*, vol. 13, no. 3, pp. 566-579, August 1984.
- [32] D. R. Shier, "Some aspects of perfect elimination orderings in chordal graphs," *Discrete Applied Mathematics*, Vol. 7, pp. 325-331, 1984.
- [33] R. E. Tarjan and M. Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," *SIAM Journal of Computing*, vol. 13, no. 3, pp. 566-579, August 1984.
- [34] M. Yannakakis, "Algorithms for acyclic database schemes," *Proc. International Conference on Very Large Data Bases*

VITA

Eric Kirsch was born in Pittsburgh, Pennsylvania and still holds a special fondness for that city. He moved to Worthington, Ohio in the fourth grade and graduated from Worthington High School. After a brief period of living in New Jersey, memorable only for its proximity to New York City, he moved with his family to Harrisonburg, Virginia.

In Harrisonburg, he attended James Madison University and received a Bachelor of Arts degree in English. He pursued English because he wanted to be a novelist. He still loves to read, especially science fiction, and still hopes to write some science fiction.

Due to a lack of jobs in his field, he began work for Kawneer Corporation and remained there until he decided to further his education. He decided to study Computer Science at the University at Tennessee and although he really had no idea what he was getting into, he has had nothing but good experiences at UT. The author received a Master of Science degree from the University of Tennessee in May of 1989 and plans to pursue a PhD there.