Efficient parallel algorithms for chordal graphs*

Philip N. Klein†
Brown University

Abstract

We give the first efficient parallel algorithms for recognizing chordal graphs, finding a maximum clique and a maximum independent set in a chordal graph, finding an optimal coloring of a chordal graph, finding a breadth-first search tree and a depth-first search tree of a chordal graph, recognizing interval graphs, and testing interval graphs for isomorphism. The key to our results is an efficient parallel algorithm for finding a perfect elimination ordering.

Introduction

Chordal graphs are graphs in which every cycle of length > 3 has a *chord*, an edge between non-consecutive nodes of the cycle. Chordal graphs have application in Gaussian elimination [39] and in databases [4], and have been the object of much algorithmic study since the work of Fulkerson and Gross in 1965 [17].

Chordal graphs are an important subclass of the class of *perfect graphs* [5], which are graphs in which the maximum clique size equals the chromatic number for every induced subgraph. No polynomial-time algorithm for recognizing perfect graphs is known. In contrast, chordal graphs can be recognized in linear time.

Our results

In this paper, we give the first efficient parallel algorithms for a host of chordal graph problems. Our deterministic algorithms take $O(\log^2 n)$ time and use only $n + m$ processors of a CRCW P-RAM for n-node, m-edge graphs. Moreover, using randomized techniques [19], [33],[38], we can achieve the same time bound with only $(n + m)/\log n$ processors. Thus our algorithms are nearly optimal in their use of parallelism, in contrast to the previous parallel algorithms that required about n^3 processors to achieve the same time bound. The chordal graph problems we solve are:

---

*A preliminary version of this paper appeared in the Proceedings of the 1988 Symposium on the Foundations of Computer Science, pages 150-161, and also in technical report TR-426 of the Laboratory for Computer Science, MIT, 1988.

†The research described in this paper is part of the author's Ph.D. thesis, done at MIT. Support for the research was provided by a fellowship from the Center for Intelligent Control Systems, with additional support from Air Force Contract AFOSR-86-0078, and from PYI awarded to David Shmoys, with matching support from IBM and Sun Microsystems. Yet more support was provided by ONR Grant N00014-08-K-0243 at Harvard University

1. recognizing chordal graphs,
2. finding all maximal cliques in a chordal graph (and, in particular, finding a maximum-weight clique),
3. finding a maximum independent set (and a minimum clique-cover) in a chordal graph,
4. finding an optimal coloring of a chordal graph,
5. finding a depth-first search tree of a chordal graph,
6. finding a breadth-first search tree of a chordal graph.

Chordal graphs include as a subclass *interval graphs*, the intersection graphs of intervals of the real line. Thus our algorithms for problems 2 through 6 above may be applied to interval graphs. But we can also solve two additional interval graph problems. Namely, in $O(\log^2 n)$ time using $n+m$ processors, we can

- recognize interval graphs and find interval representations, and
- test isomorphism between interval graphs.

The isomorphism algorithm requires the CRCW P-RAM to be of type "priority" (higher-numbered processors win in case of write conflicts). It makes use of an efficient parallel algorithm for tree isomorphism.

Other parallel algorithms

In previous work, Naor, Naor, and Schäffer [34] gave parallel algorithms for chordal graph problems 1 through 4. Their algorithm for problem 1 used $O(n^2 m)$ processors. They also gave an algorithm for problem 2 that required $O(n^{5+\epsilon})$ processors to achieve $O(\log^2 n)$ time, and $O(n^4)$ processors to achieve $O(\log^3 n)$ time. They showed how, subsequent to the solution of problem 2, problems 3 and 4 could be solved in $O(\log^2 n)$ additional time using $O(n^2)$ processors. Thus they identified problem 2 as a bottleneck in analyzing chordal graphs. Subsequent research (independent of and concurrent with our work) by Dahlhous and Karpinski and Ho and Lee [24] reduced the processor bound for problem 2 to $O(n^4)$ and $O(n^3)$, respectively; because of the algorithms of [34], these processor bounds then apply also to problems 3 and 4. The algorithm of Ho and Lee required only $O(\log n)$ time.

A parallel algorithm for finding a depth-first search tree in an arbitrary graph was given by Aggarwal and Anderson [2]. Their algorithm is randomized and uses $O(nM(n))$ processors.[1] A parallel algorithm for breadth-first search in an arbitrary graph was given by Gazit and Miller [20] that uses $M(n)$ processors. Depth-first and breadth-first algorithms specifically for chordal graphs have not previously appeared in the literature.

To our knowledge, no previous NC algorithm was known for interval graph isomorphism. Recognition of interval graphs was previously shown to be in NC by Kozen, Vazirani, and Vazirani [30], but no specific time or processor bound was given. Novick [37] has given an $O(\log n)$ time, n^3 processor CRCW algorithm for recognizing interval graphs. He has also claimed [36] an algorithm with the same bounds for constructing a PQ-tree representing a given interval graph. He observed [35] that this latter task is the first step in Lueker and Booth's interval graph isomorphism algorithm, and suggested that the remaining steps might be parallelizable. Savage and Wloka

[1] Here $M(n)$ denotes the time required to multiply two $n \times n$ matrices. The best bound known, due to Coppersmith and Winograd [11], is $O(n^{2.376})$.

[41] have given an efficient parallel algorithm for optimum coloring of interval graphs. Their algorithm takes $O(\log n)$ time using n processors of an EREW P-RAM, assuming that the interval representation of the graph has been provided.

Background

The key to our algorithmic results is our use of the *perfect elimination ordering* (`peo`) of a graph, a node-ordering that exists if and only if the graph is chordal. Fulkerson and Gross [17] discovered the `peo` and used it to find all the maximal cliques of a chordal graph. Rose ([39]) has related the `peo` to the process of Gaussian elimination in a sparse symmetric positive definite linear system. Rose, Tarjan, and Lueker gave a linear-time algorithm for finding a `peo` in a chordal graph, using the notion of *lexicographic breadth-first search.* This yields a linear-time sequential algorithm for recognizing chordal graphs (problem 1). Once a `peo` for a graph is known, algorithms due to Gavril [18] for problems 2,3, and 4 can be implemented in linear time. Thus the `peo` has emerged as the key technique in sequential algorithms for chordal graphs, and its study has yielded important algorithmic ideas in the sequential realm.

Researchers in parallel algorithms, however, have largely abandoned use of the `peo`—largely because finding a `peo` in parallel seemed so difficult. The existence of an NC algorithm for finding a `peo` algorithm was left open by Edenbrandt [15],[16], and by Chandrasekharan and Iyengar [7], and resolved by Naor, Naor, and Schäffer [34] and independently by Dahlhaus and Karpinski [12]. However, the `peo` algorithms of [34] and [12] required at least n^4 processors (subsequently improved to n^3 by Ho and Lee [24], [25]). In fact, it is suggested in [34] that for parallel algorithms the `peo` may be less useful than the representation of a chordal graph as the intersection graph of subtrees of a tree. The results of this paper suggest otherwise.

We describe a new parallel algorithm for finding a `peo`. Our algorithm takes $O(\log^2 n)$ time and uses only a linear number of processors of a CRCW P-RAM. In fact, we can achieve the same time bound using only $O((n+m)\log n)$ processors of a randomized P-RAM. Thus our `peo` algorithm is nearly optimal. The algorithm relies on a new understanding of the combinatorial nature of `peo`'s. The algorithm in turn forms the basis for our other efficient parallel algorithms for chordal and interval graphs.

Our algorithm for finding a `peo` actually solves the following problem: given a labeling of the nodes of a graph with numbers(a *numbering*), the algorithm finds a perfect elimination ordering consistent with the partial order defined by the numbering, or determines that no such consistent `peo` exists. It accomplishes this by iteratively refining the numbering until each number is assigned to only one node. Our methods ensure that only $O(\log n)$ refinements suffice. Once the numbering is one-to-one, it is easy to check whether it defines a `peo`, as we observe in Subsection 3.2.

Most of our algorithms for solving optimization problems on a chordal graph rely on a tree derived from the `peo`, the *elimination tree.* We show that breaking up the tree by removing a vertex corresponds to breaking up the graph by removing a clique. We use this observation to give a divide-and-conquer algorithm for optimum coloring. In order to find a maximum independent set and a minimum clique cover of the graph, we apply terminal branch elimination, a technique of Naor, Naor, and Schäffer [34], to the elimination tree. We believe the elimination tree may also prove useful in other parallel chordal graph algorithms.

The interval graph algorithms rely on a parallel algorithm, MREDUCE, for manipulating the PQ-data structure of Booth and Lueker [6]. In this paper, we provide only a hint of how that algorithm works. For more details, the reader is referred to the author's Ph.D. thesis [27].

Graph Notation

Let G be an undirected graph. We use $V(G)$ to denote the set of nodes of G. Let H be a subgraph of G or a set of nodes of G. We use $G[H]$ to denote the subgraph of G induced by the nodes of H. We use $G - H$ to denote the subgraph obtained from G by deleting the nodes of H. We use $|H|$ to denote the number of nodes in H. Unless otherwise stated, n and m denote the number of nodes and number of edges, respectively, in the graph G.

1 The peo algorithm

The most algorithmically useful characterization of chordal graphs, the peo, was discovered by Fulkerson and Gross [17] in 1965. Dirac had proved in [14] that every chordal graph has a *simplicial* node, a node whose neighbors form a clique. Fulkerson and Gross observed that since every induced subgraph of a chordal graph is also chordal, deletion of a vertex and its incident edges results in a chordal graph. They proposed an "elimination" process: repeatedly find a simplicial node and delete it until all nodes have been deleted or no remaining node is simplicial. It follows from Dirac's theorem that the process deletes every node of a chordal graph; in fact, Fulkerson and Gross showed conversely that a graph is chordal if the process deletes every node. Thus the elimination process constitutes an algorithm for recognition of chordal graphs. The order in which nodes are deleted is called a *perfect elimination ordering*, which we abbreviate as peo.

1.1 An overview of the peo algorithm

We define a peo of a graph G to be a one-to-one numbering $v_1, \ldots, v_n$ of the nodes of G such that for each i ($i = 1, \ldots, n$), the higher-numbered neighbors of v_i form a clique. We also represent a peo as a sequence of nodes $\sigma = v_1 \ldots v_n$.

Theorem 1.1 [Fulkerson and Gross] A graph G has a perfect elimination ordering if and only if G is chordal.

In order to give a parallel algorithm for finding a peo of a chordal graph G, we generalize the notion to numberings that are not one-to-one. Let \$ be a numbering of the nodes of G (a function mapping nodes to numbers). We use $G_\$$ to denote the graph G with each node v labelled by its number $\$(v)$. We shall use the metaphor of wealth in connection with numberings \$; for example, if $\$(v) > \$(w)$, we shall say v is "richer" than w. The *classes* of $G_\$$ are the subgraphs induced on sets of equal-numbered nodes. The *class-components* of $G_\$$ are the connected components of the classes of $G_\$$.

Let \$ and £ be two numberings of the nodes of G. We say \$ is *consistent with £* (and £ is a *refinement of \$*) if $\$(v) < \$(w)$ implies $£(v) < £(w)$ for all nodes v and w. We say £ is a refinement of $G_\$$ if we wish to emphasize the graph for which £ is a numbering. Note that each class-component of $G_£$ is a subgraph of some class-component of $G_\$$.

We call ¢ a *partial numbering* if ¢ assigns numbers to *some* of the nodes of G, and is undefined for others; a numbering is trivially a partial numbering. For the numbering \$ and the partial

ITERATED REFINEMENT

R1 To initialize, let \$ be the trivial numbering assigning 0 to every node of G.

R2 While \$ is not one-to-one,

R3 For each nonsingleton class-component C of $G_\$$ in parallel:

R4 call STRATIFY($G_\$, C$).

Figure 1: The ITERATED REFINEMENT algorithm for finding a peo.

numbering ¢, the *refinement of \$ by ¢* is defined to be the numbering \$¢ in which ¢ is used to break ties in \$. That is, $\$¢(v) < \$¢(w)$ if either

- $\$(v) < \$(w)$, or
- $\$(v) = \$(w)$, $¢(v)$ and $¢(w)$ are defined, and $¢(v) < ¢(w)$.

Typically, ¢ will be a numbering of some class-component C of $G_\$$, and hence only a partial numbering of $G_\$$. In this case, we speak of obtaining \$¢ from \$ as *stratifying* the class-component C of $G_\$$, or as *well-stratifying* C if in addition each class-component of $G_{\$¢}$ contains at most $\frac{4}{5}|C|$ nodes of C.

We want to know when a numbering \$ is consistent with some peo. To this end, we introduce the notion of a *backward path* in $G_\$$: namely, a simple path whose endpoints are strictly richer than all its internal nodes.[2] We say a numbering \$ of G is *valid* if every backward path in $G_\$$ has adjacent endpoints. The following lemmas are immediate from the definitions.

Lemma 1.2 For a graph G, if a valid numbering \$ of G is also one-to-one, then \$ is a peo of G.

Lemma 1.3 Let \$ be a valid numbering of a graph G. For any class-component C of $G_\$$, the richer neighbors of C form a clique.

We assume for the remainder of this section that G is a connected chordal graph. Our algorithm for finding a peo in G, which appears in Figure 1, consists of a sequence of $O(\log n)$ stages. In each stage, the algorithm modifies the numbering \$ by well-stratifying every nonsingleton class-component C while preserving the validity of \$, using a procedure STRATIFY($G\$, C$). In each stage, the size of the largest class-component goes down by a factor of four-fifths. Hence after at most $\log_{5/4} n$ stages, the current numbering is one-to-one, and the algorithm terminates, outputting the current numbering, which is a peo by Lemma 1.2.

We shall show in Subsection 1.2 that the procedure STRATIFY($G_\$, C$) can be implemented to run in $O(\log k)$ time using k processors, where k is the number of edges that have at least one endpoint in C. Consequently, executing step R4 of ITERATED REFINEMENT for all class-components in parallel requires $O(\log m)$ time using $O(m)$ processors. Since the number of stages is $O(\log n)$, the total time required by ITERATED REFINEMENT is $O(\log^2 n)$. Using the randomized connectivity

[2] The notion is a generalization of one appearing in Lemma 4 of [40].

Figure 2: To obtain a uniform path, delete the nodes of backward subpaths.

algorithm of Gazit [19], we can reduce the processor bound by a factor of $\log n$ without increasing the time bound.

As an aside, we note that the initial numbering can be any valid numbering \$ of G, e.g. the trivial numbering assigning the same number to all nodes; the algorithm's output will then be a peo consistent with \$. This observation leads to the following theorem, which is not needed for our algorithm, but which justifies our initial definition of validity.

Backward Path Theorem: For a chordal graph G, a numbering \$ of G is valid if and only if it is consistent with some peo of G.

Proof: The "only-if" direction will follow from the correctness of the algorithm. To prove the other direction, suppose σ is a peo of G consistent with \$. We need to show that every backward path in $G_\$$ has adjacent endpoints. For the two endpoints x and y of any backward path, let P be the shortest backward path with these two endpoints. If P consists of the single edge $\{x, y\}$, we are done, so assume that P has internal nodes. Let u be the internal node with the minimum σ-number. Then the neighbors of u in P have higher σ-number, so they are adjacent by definition of a perfect ordering. Thus there is a shorter backward path connecting x and y, a contradiction. $\square$

We return to the algorithm. The key to the efficiency of the procedure Stratify$(G_\$, C)$ is that it need only consider the graph induced by C and its richer neighbors in $G_\$$, as we shall show presently.

We say that a path in $G_\$$ is *weakly backward* if its endpoints are at least as rich as its internal nodes. When we wish to emphasize that a path P is backward, as opposed to only weakly backward, we shall say P is *strictly* backward. If £ is a refinement of \$, a path P that is weakly backward in $G_£$ is also weakly backward in $G_\$$. If P is strictly backward in $G_£$ but not strictly backward in $G_\$$, then at least one of the endpoints of P has the same \$-number as one of the internal nodes of P. We say a path in G is *uniform* (with respect to a numbering \$) if every internal node has the same \$-number as the poorer of the two endpoints.

Lemma 1.4 Suppose \$ is a valid numbering of the graph G. For each weakly backward path P in $G_\$$, there is a uniform weakly backward path P' in $G_\$$ with the same endpoints such that $V(P') \subseteq V(P)$.

Proof: The proof is illustrated in Figure 2. Let $v_1 \ldots v_k$ be the nodes of a weakly backward path P, and let t be the \$-number of its poorer endpoint. Suppose $v_i \ldots v_j$ is a maximal subpath consisting of nodes poorer than t. Then $v_{i-1} \ldots v_{j+1}$ is a backward path in $G_\$$, so its endpoints

v_{i-1} and v_{j+1} are adjacent, by the validity of \$. We can therefore replace the subpath $v_{i-1} \ldots v_{j+1}$ with the edge $\{v_{i-1}, v_{j+1}\}$, obtaining a backward path P_1 with the same endpoints as P but with fewer nodes poorer than t. Note that every node in P_1 is a node of P. Continuing this process yields a path with the same endpoints and with no nodes poorer than t and consisting of a subset of the nodes of P. $\square$

The following lemma shows that STRATIFY$(G_\$, C)$ need only consider C and the higher-numbered neighbors of C. Fix the graph $G_\$$. For a class-component C, let $\widehat{C}$ denote the subgraph of $G_\$$ induced by C and its higher-numbered neighbors in $G_\$$.

Refinement Lemma: Suppose \$ is a valid numbering of a graph G. Let $\not{c}$ be a numbering of a class-component C of $G_\$$. Then the refinement of $G_\$$ by $\not{c}$ is valid if and only if the refinement of $\widehat{C}_\$$ by $\not{c}$ is valid.

Proof: Let $\$\not{c}$ be the refinement of $G_\$$ by $\not{c}$, and let λ be the refinement of $\widehat{C}_\$$ by $\not{c}$. Assume that \$ is a valid numbering of G. If $\$\not{c}$ is also a valid numbering of G, then λ is a valid numbering of $\widehat{C}$, because $\widehat{C}_\lambda$ is a node-induced subgraph of $G_{\$\not{c}}$. Conversely, suppose that λ is a valid numbering of $\widehat{C}$. For each backward path P in $G_{\$\not{c}}$, we must show that P's endpoints x and y are adjacent. Since P is weakly backward in $G_\$$, there exists a uniform path P' in $G_\$$ with endpoints x and y, where $V(P') \subseteq V(P)$. Since $V(P') \subseteq V(P)$, the path P' is still backward with respect to $\$\not{c}$. If P' has no internal nodes, x and y are adjacent. Suppose P' has internal nodes; they all have the same \$-number as the lower-numbered endpoint x, by definition of uniformity. Since P' is strictly backward with respect to $\$\not{c}$, it follows that the internal nodes have a lower $\not{c}$-number than x. Hence the internal nodes and x must all lie in C, because $\not{c}$ is defined only on C. Then the other endpoint y is a neighbor of a node of C, and hence is either in C itself, or is a higher-numbered neighbor of C. In the first case, y has a higher $\not{c}$-number than the internal nodes because P' is strictly backward. In the second case, y has a higher λ-number than the internal nodes. In either case, we conclude that P' is a backward path in $\widehat{C}_\lambda$. By the validity of λ, x and y are adjacent. $\square$

The Refinement Lemma implies that to validly stratify a class-component C in $G_\$$, we need only validly stratify it in $\widehat{C}_\$$. In fact, we observe next that all the class-components of $G_\$$ may be thus stratified simultaneously and independently. To see this, let $C^1, \ldots, C^k$ be the class-components of $G_\$$, ordered in some arbitrary way consistent with \$. We show that stratifying the class-components in order is equivalent to stratifying them all at once, as far as validity is concerned.

For $i = 1, \ldots, k$, let $\not{c}_i$ be a numbering of C^i such that the refinement of $\widehat{C^i}_\$$ by $\not{c}_i$ yields a valid numbering of $\widehat{C^i}$. Let $\$_0 = \$$, and, for $i = 1, \ldots, k$, let $\$_i$ be the refinement of $\$_{i-1}$ by $\not{c}_i$. The numbering that $\$_{i-1}$ induces on $\widehat{C^i}$ is isomorphic to that induced by \$ (the same order relations hold), so refining $\widehat{C^i}_{\$_{i-1}}$ by $\not{c}_i$ is valid. It then follows via the Refinement Lemma that $\$_i$ is valid. We conclude that $\$_k$ is a valid refinement of \$. We can obtain $\$_k$ directly by using $\not{c}_i$ to stratify C^i, for all class-components C^i in parallel. This is how steps R3 and R4 of ITERATED REFINEMENT are carried out. The algorithm STRATIFY$(G_\$, C^i)$ for stratifying C^i is given in Subsection 1.2.

1.2 Valid well-stratification

Before we give the algorithm for valid well-stratification, we give some results used in proving the correctness of the algorithm. We start with a lemma of Dirac [14].

Lemma 1.5 [Dirac] If S is a minimal set of nodes whose removal separates a connected chordal graph into exactly two connected components, then S is a clique.

Corollary 1.6 In a chordal graph, the common neighbors of two nonadjacent nodes form a (possibly empty) clique.

Proof: In the induced subgraph consisting of the two nonadjacent nodes x and y, and their common neighbors, the common neighbors form a minimal separator between x and y. □

Corollary 1.7 Let H be a connected subgraph of a chordal graph G. Then the numbering δ is valid, where δ assigns 1 to H and neighbors of H, and 0 to all other nodes.

Proof: Let P be a backward path in G_δ, and let C be the component of the 0-numbered nodes that contains the internal nodes of P. Let D be the set of 1-numbered neighbors of nodes in C. In the subgraph induced on $C \cup D \cup H$, the nodes of D form a minimal separator between C and H, so D is a clique. The endpoints of P are in D, so they are adjacent. □

Lemma 1.8 Let K be a clique in a chordal graph G. Then the numbering δ is valid, where δ assigns 1 to K and those nodes adjacent to all of K, and 0 to all other nodes.

Proof: by induction on $|K|$. The base case, in which $|K| = 1$, follows from Corollary 1.7. Suppose $|K| > 1$, and let v be a node of K. Let A consist of the nodes of $K - \{v\}$ and the nodes adjacent to all of $K - \{v\}$. Let α be the numbering assigning 1 to A, and 0 to other nodes. By the inductive hypothesis, α is valid. Because K is a clique, v is in A. Let β be the numbering of A that assigns 2 to v and its neighbors, and 1 to other nodes of A. By Corollary 1.7, β is a valid numbering of A. Let γ be the refinement of α by β. The nodes of A have no richer neighbors in G_α, so by the Refinement Lemma, γ is a valid numbering of G. But γ is a refinement of the numbering δ defined in the statement of the lemma, so δ is also valid. This completes the inductive step. □

Lemma 1.9 Let α be any valid numbering of a graph G, and let K be a clique contained in the highest-numbered class of G_α. Suppose the numbering γ is obtained from α by increasing the numbers of nodes of K. Then γ is valid.

Proof: The only backward paths introduced have endpoints in the clique K. □

Lemma 1.10 Let \$ be a valid numbering of a graph G. Suppose C is a class-component of $G_\$$, and all nodes in C have the same richer neighbors in $G_\$$. Let ¢ be a valid numbering of C. Then the refinement of $G_\$$ by ¢ is valid.

Procedure $\textsc{Stratify}(G_\$, C)$:

S1 For each node v in C, identify those edges connecting v to another node in C, and those edges connecting v to a richer node.

S2 Let B be the set of richer neighbors of C.

S3 If B is empty, call $\textsc{None}(G_\$, C, 1)$, and end.

S4 Let ϵ be $1/2|C|$ times the difference between $\$(C)$ and the number assigned to the next higher class.

S5 If every node in B has at least $\frac{2}{5}|C|$ neighbors in C, call $\textsc{HighDegree}(G_\$, C, B, \epsilon)$, and end.

S6 If there are nodes in B with fewer than $\frac{2}{5}|C|$ neighbors in C, call $\textsc{LowDegree}(G_\$, C, B, \epsilon)$.

Figure 3: The main procedure for finding a valid well-stratification.

Proof: By the Refinement Lemma, we need only show that the refinement λ of $\widehat{C}_\$$ by $\not{c}$ preserves validity. Assume \$ is valid, so the nodes of $\widehat{C}$ not in C form a clique by Lemma 1.3. Let P be a backward path in $\widehat{C}_\lambda$ with endpoints x and y. We must show that x and y are adjacent. If both endpoints are in C, they are already adjacent by the validity of $\not{c}$. If neither is in C, they belong to a clique, and hence are adjacent. Suppose therefore that x is in C and y is not. Since P is a backward path and an endpoint lies in C, all the internal nodes of P must also lie in C. Hence y is a richer neighbor of C in $\widehat{C}_\$$. Since all nodes in C have the same richer neighbors, it follows that y is a neighbor of x. $\square$

The procedure $\textsc{Stratify}(G_\$, C)$ appears in Figure 3. If C is a nonsingleton class-component of $G_\$$, the procedure increases the numbers of some of the nodes of C, resulting in a valid refinement of $G_\$$ in which C has been well-stratified. The procedure takes $O(\log k)$ time using $O(k)$ processors, where k is the number of edges of G with at least one endpoint in C. To achieve this processor bound, the procedure first identifies these edges by inspecting the adjacency lists of nodes in C, and subsequently never examines any other edges of G. While inspecting the adjacency lists, the procedure also identifies the set B of richer neighbors of C in $G_\$$. The procedure uses the fact that if \$ is a valid numbering of G, then the set of nodes B form a clique, by Lemma 1.3. The procedure assumes the existence of edges between nodes in B without ever checking for their presence. Specifically, in computing connected components of a graph involving nodes of B, the procedure uses the algorithm of Shiloach and Vishkin [43], suitably modified to take into account the fact that any two nodes of B are adjacent.

Depending on the nodes in B, the procedure $\textsc{Stratify}(G_\$, C)$ calls one of three subprocedures, in which most of the work is done. In each procedure, we make use of *parallel prefix computation*, due to Ladner and Fischer [31]. In particular, the subprocedures increase the numbers assigned to some of the nodes of C. We must make certain that these numbers are not increased too much. The new numbers to be assigned to nodes of C must all be less than numbers currently assigned to nodes richer than C; otherwise, the resulting numbering would not be a refinement of the old numbering. Therefore in each procedure we let ϵ denote a number small enough that the numbers of nodes of G richer than C exceed the numbers of nodes of C by at least $|C|\epsilon$. This choice of ϵ allows us to increase the numbers of C by up to $|C|\epsilon$ and still end up with a refinement of the old numbering. We find this convention useful in presenting the algorithms; however, see Implementation Note 1.

We now show that $\textsc{Stratify}(G_\$, C)$ succeeds in finding a valid refinement that well-stratifies C. We shall refer to the nodes of C as *crimson* nodes, and the nodes of B as *blue* nodes. We consider three cases, corresponding to the three procedures.

Case I: There are no blue nodes. In this case, procedure $\textsc{None}(G_\$, C, \epsilon)$ shown in Figure 4 is called. The procedure first identifies the set D of high-degree nodes: $D = \{v \in C : v \text{ has} > \frac{3}{5}|C| \text{ neighbors in } C\}$. The procedure then branches according to the following cases:

Subcase (1): Some component H of $C - D$ has size $> \frac{4}{5}|C|$. In this case, the procedure uses the spanning tree T of H to choose a connected subgraph H' of H such that the set A consisting of H' and its neighbors includes between $n/5$ and $4n/5$ nodes. Then the numbers assigned to nodes of A are increased, resulting in a well-stratification of C. The validity of the numbering follows from Lemma 1.7.

Subcase (2): Each component of $C - D$ has size at most $\frac{4}{5}|C|$, and D is a clique. In this case, we increase the numbers assigned to nodes of D, placing each node of D in its own class. The components of the remaining nodes of C are small, so we have well-stratified C. The validity of the numbering follows from Lemma 1.9.

Subcase (3): D is not a clique. In this case, we choose two nonadjacent nodes x and y in D. At most $\frac{2}{5}|C|$ nodes of C are not neighbors of x, and so at least $\frac{1}{5}|C|$ neighbors of y are also neighbors of x. We increase the numbers of these common neighbors of x and y, putting each in its own class. The numbering is valid by Lemma 1.6 and Lemma 1.9.

Case II: Each blue node is adjacent to at least $\frac{2}{5}|C|$ crimson nodes. In this case, the procedure $\textsc{HighDegree}(G_\$, C, B, \epsilon)$ of Figure 5 is called. The procedure arbitrarily orders the blue nodes: $B = \{v_1, \ldots, v_k\}$. For $1 \le j \le k$, let F_j be the set of nodes v such that v is adjacent to all the nodes $v_1, \ldots, v_j$. Then $\hat{\jmath}$ is chosen to be the maximum j such that F_j contains at least $|C|/5$ crimson nodes. The numbers of nodes in $F_{\hat{\jmath}}$ are then increased by ϵ. The validity of the resulting numbering follows from Lemmas 1.8 and 1.9. At most $\frac{4}{5}|C|$ nodes of C do not have their numbers increased. However, the set $F_{\hat{\jmath}}$ may be quite large. We consider two subcases.

Subcase (1): $\hat{\jmath} < k$. In this case, by choice of $\hat{\jmath}$, the set of crimson nodes adjacent to all the nodes $v_1, \ldots, v_{\hat{\jmath}+1}$ is less than $|C|/5$. Since there are at most $\frac{3}{5}|C|$ crimson nodes not adjacent to $v_{\hat{\jmath}+1}$, it follows that the number of nodes adjacent to all the nodes $v_1, \ldots, v_{\hat{\jmath}}$ is less than $\frac{1}{5}|C| + \frac{3}{5}|C| = \frac{4}{5}|C|$. Thus C has been well-stratified in this case.

Subcase (2): $\hat{\jmath} = k$. In this case, every node in $F_{\hat{\jmath}}$ is adjacent to every blue node. The procedure finds the largest component C' of $G[F_{\hat{\jmath}}]$, and calls $\textsc{None}(G_\$, C', \epsilon)$, which stratifies C' as if C' had *no* richer neighbors. The validity of the resulting numbering of $\widehat{C}$ follows from Lemma 1.10 and the Refinement Lemma. Since C' is well-stratified and $|C - C'| \le \frac{4}{5}|C|$, C is well-stratified.

Case III: Neither Case I nor Case II holds. In this case, the procedure $\textsc{LowDegree}(G_\$, C, B, \epsilon)$ in Figure 6 is called. The procedure defines D to be the set of nodes in $C \cup B$ having more than $\frac{3}{5}|C|$ crimson neighbors. The procedure then finds a spanning tree T of the (unique) component H of $G[(C \cup B) - D]$ containing blue nodes, and roots T at a blue node. The nodes of T are arranged in some order consistent with their distance from the root: $v_1, \ldots, v_k$. For $1 \le j \le k$, let A_j be the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in $\widehat{C}$. Then $\hat{\jmath}$ is chosen to be the maximum j such that A_j includes at most $\frac{4}{5}|C|$ crimson nodes.

Procedure NONE($G_\$, C, \epsilon$):

N1 Let $D = \{v \in C : v \text{ has } > \frac{3}{5}|C| \text{ neighbors in } C\}$.

N2 Find a spanning forest of $C - D$.

N3 If some component H of $C - D$ has at least $\frac{4}{5}|C|$ nodes,

N4   Let T be the spanning tree of H.

N5   Arrange the nodes of H in some order consistent with their distance in T from the root: $v_1, \ldots, v_k$.

N6   For $1 \leq j \leq k$, let A_j denote the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in C.

N7   Using parallel prefix computation, choose $\hat{\jmath} = \max\{j : |A_j| \leq \frac{4}{5}|C|\}$.

N8   Increase the numbers of nodes in $A_{\hat{\jmath}}$ by ϵ.

N9 Otherwise,

N10   If D is a clique,

N11   Let $v_1, \ldots, v_k$ be the nodes of D.

N12   For $1 \leq i \leq k$, add ϵi to v_i's number.

N13   Otherwise,

N14   Let x and y be two nonadjacent nodes of D.

N15   Let $v_1, \ldots, v_k$ be their common neighbors.

N16   For $1 \leq i \leq k$, add ϵi to v_i's number.

Figure 4: The subprocedure used to well-stratify C if C has no richer neighbors.

Next, the numbers of nodes in $A_{\hat{\jmath}}$ are increased, in step L7. To see that the resulting numbering is valid, first consider the intermediate numbering in which all nodes in $A_{\hat{\jmath}}$ have the same number, a number higher than that assigned to the nodes in $\widehat{C} - A_{\hat{\jmath}}$. Since $H[\{v_1, \ldots, v_{\hat{\jmath}}\}]$ is connected, the intermediate numbering is valid by Lemma 1.7. To obtain the numbering produced in step L7 from the intermediate numbering, we need only increase the numbers of some blue nodes. The blue nodes form a clique lying in $A_{\hat{\jmath}}$, so the validity of the final numbering follows from Lemma 1.9.

The set $A_{\hat{\jmath}} \cap C$ of nodes whose numbers have increased has size at most $\frac{4}{5}|C|$. However, the set $C - A_{\hat{\jmath}}$ of nodes whose numbers have not increased may be quite large. The procedure finds the largest component C' of $C - A_{\hat{\jmath}}$, and proceeds according to the following two cases.

Subcase (1): $|C'| \leq \frac{4}{5}|C|$. In this case, we are done; C has been well-stratified.

Subcase (2): $|C'| > \frac{4}{5}|C|$. First, we observe that in this case, $\hat{\jmath} = k$. To see this, suppose $\hat{\jmath} < k$. Then the number of crimson nodes among $\{v_1, \ldots, v_{\hat{\jmath}+1}\}$ and neighbors is more than $\frac{4}{5}|C|$. Since $v_{\hat{\jmath}+1}$ is adjacent to at most $\frac{3}{5}|C|$ crimson nodes, it follows that the number of nodes whose numbers have increased is more than $\frac{4}{5}|C| - \frac{3}{5}|C| = \frac{1}{5}|C|$, which contradicts the fact that $|C'| > \frac{4}{5}|C|$.

Since $\hat{\jmath} = k$, every crimson node in T and every crimson neighbor of T has had its number increased. Therefore, C' contains no nodes of T and no neighbors of T. Every node in D has

Procedure HighDegree($G_\$, C, B, \epsilon$)
(Each node of B has at least $\frac{2}{5}|C|$ neighbors in C.)
H1 Arbitrarily order the nodes of B: $v_1, \ldots, v_k$.
H2 for $1 \leq j \leq k$, let F_j denote the set of nodes of C adjacent to *all* of the nodes $v_1, \ldots, v_j$.
H3 Using parallel prefix computation, choose $\hat{\jmath} = \max\{j : |F_j| \geq \frac{1}{5}|C|\}$.
H4 Increase the numbers of nodes in $F_{\hat{\jmath}}$ by ϵ.
H5 Let C' be the largest component of $G[F_{\hat{\jmath}}]$.
H6 If $\hat{\jmath} = k$, then call None($G_\$, C', \epsilon$).

Figure 5: The procedure HighDegree used to stratify C in case every richer neighbor of C has high degree in C.

more than $\frac{3}{5}|C|$ crimson neighbors, and all but at most $\frac{1}{5}|C|$ of the crimson nodes are in C', so every node in D has more than $\frac{2}{5}|C|$ crimson neighbors in C'. Thus every richer neighbor of C' is adjacent to at least $\frac{2}{5}|C|$ nodes of C'. This shows that the recursive call to Stratify in step L9 results in a call to HighDegree, and not in a call to LowDegree. Thus no further recursive calls occur. The recursive call well-stratifies C', and hence C as well, since $|C - C'| \leq \frac{1}{5}|C|$. The validity of the resulting numbering follows from the Refinement Lemma.

Implementation Note 1 We have described the procedures Iterated Refinement and Stratify as if real numbers were used to number the nodes. In implementing the algorithm, however, it is desirable to use integers¡ to number the nodes. The simple approach is to multiply all numbers by n at the beginning of each stage, and then renumber by sorting at the end of each stage. A more efficient approach is to intially allocate $4\log_{5/4} n$ bits for each node label, four bits per stage of Iterated Refinement. For the ith stage, we use the $4(i-1)+1^{\text{st}}$ through the $4i^{\text{th}}$ most significant bits. In Case I, Subcases (1) and (2), the procedure needs more bits; the procedure must assign a different number to each node of a clique. For each of these nodes, however, we can afford to use all the remaining bits because each such node ends up in its own class, and hence needs no further labeling in subsequent stages.

Implementation Note 2 In step N7 of procedure None, we ordered the nodes $v_1, \ldots, v_k$ of a spanning tree T, and chose $\hat{\jmath}$ maximum such that $\{v_1, \ldots, v_{\hat{\jmath}}\}$ and neighbors comprise at most $\frac{4}{5}|C|$ nodes. Here we provide more details for implementing that step; the techniques are also applicable to step H3 of procedure HighDegree and to step L6 of procedure LowDegree. For each node $v \neq v_1$ in $\widehat{C}$, let *earliest-nbr*(v) be the minimum i such that v is adjacent to v_i, or *undefined* if v has no neighbors among $v_1, \ldots, v_k$. Let *earliest-nbr*$(v_1) = 1$. For each node v_i in T, *earliest-nbr*$^{-1}(v_i)$ is the set of neighbors of v_i in $\widehat{C}$ that are not neighbors of any lower-numbered node. Let $f(v_i) = |\textit{earliest-nbr}^{-1}(v_i)|$, and let $g(\ell) = \sum_{i=1}^{\ell} f(v_i)$ for $\ell = 1, \ldots, k$. Then $g(\ell)$ is the number of nodes comprised by $v_1, \ldots, v_\ell$ and their neighbors in $\widehat{C}$. The function $g(\cdot)$ can be computed from $f(\cdot)$ by a parallel prefix computation, after which $\hat{\jmath}$ is chosen as large as possible such that $g(\hat{\jmath}) \leq \frac{4}{5}|C|$.

Procedure LowDegree($G_\$, C, B, \epsilon$)
(There exists a node in B having fewer than $\frac{2}{5}|C|$ neighbors in C.)

L1 Let $D = \{v \in C \cup B : v$ has $> \frac{3}{5}|C|$ neighbors in $C\}$.

L2 Let H be the connected component of $G[C \cup B - D]$ containing nodes of B.

L3 Find a spanning tree T of H, rooted at a node of B.

L4 Arrange the nodes of T in some order consistent with their distance in T from the root: $v_1, \ldots, v_k$.

L5 For $1 \leq j \leq k$, let A_j denote the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in C.

L6 Using parallel prefix computation, choose $\hat{\jmath} = \max\{j : |A_j \cap C| \leq \frac{4}{5}|C|\}$.

L7 Increase the numbers of nodes in $A_{\hat{\jmath}} \cap C$ by ϵ.

L8 Let C' be the largest component of $C - A_{\hat{\jmath}}$.

L9 If $|C'| > \frac{4}{5}|C|$, then call Stratify($G_\$, C'$).

Figure 6: The procedure LowDegree used to stratify C in case some richer neighbor of C has low degree in C.

Implementation Note 3 In step L2 of procedure LowDegree, we computed the connected components of an induced subgraph of G containing nodes of C and nodes of B. We want to implement this step in a way that the actual edges between nodes of B are not involved. To carry out the connected components computation (and to find spanning trees of the components), we use the connectivity algorithm of [43], suitably modified to take into account our assumption that every two blue nodes are adjacent: we start by constructing a tree containing all the blue nodes in $H - D$ (and another tree containing all the blue nodes in D. We then execute the algorithm of [43], using these artificial edges of these trees as surrogates for the set of edges between nodes in B. Thus the number of processors required is no more than the sum of $|C \cup B|$, the number of edges in C, and an additional term of $|B| - 1$ for the artificial edges.

This completes the description of the algorithm Stratify for valid well-stratification of a class-component. At most one recursive call is made, as we have shown. The time for the algorithm is dominated by the time to compute connected components and find spanning trees, which is $O(\log |C \cup B|)$ using the algorithm of [43]; as shown in Implementation Note 3, we need only $|C \cup B| + |E(C)| + |B| - 1$ processors, which is bounded by at most twice the number of edges with at least one endpoint in C. We thus obtain the following theorem.

Theorem 1.11 Suppose \$ is a valid numbering of a chordal graph G, and C is a class-component of $G_\$$. Valid well-stratification of C can be done in $O(\log k)$ time using $O(k)$ processors of a CRCW P-RAM, where k is the number of edges with at least one endpoint in C. Hence, a peo of the chordal graph G can be found in $O(\log^2 n)$ time using $O(m)$ processors.

Figure 7: A PQ-tree is depicted. The P-nodes are indicated by circles, and the Q-nodes by rectangles. The ground set is $\{a, b, c, d, e, f\}$, and the frontier is $b\ a\ f\ d\ c\ e$.

Using our algorithm for valid well-stratification in the procedure ITERATED REFINEMENT, we can therefore find a `peo` of a graph G in $O(\log^2 n)$ time using $O(n+m)$ processors.

2 PQ-trees

In this section, we review the PQ-tree data structure developed by Booth and Lueker [6]. This data structure is useful in recognition and isomorphism-testing of interval graphs, problems we address in Section 3. In Subsection 2.1, we introduce a parallel PQ-tree-processing algorithm that arises in parallel algorithms for interval graph recognition and isomorphism-testing (Section 3).

A PQ-tree is a data structure developed by Booth and Lueker [6] for representing large sets of orderings of a ground set S. A PQ-tree T over the ground set S is a rooted tree whose leaves are the elements of S; every internal node is designated either a P-node or a Q-node, and has at least two children. Hence T has at most $2n-1$ nodes. The children of each internal node are ordered from left to right. These orderings induce a left-to-right ordering on the leaves of the tree; the sequence of leaves is called the *frontier* of the tree T, and is denoted $\mathrm{fr}(T)$.

Let us say an automorphism of a PQ-tree T is *legal* if for every internal node v,

- if v is a Q-node then the automorphism either reverses the order of v's children or leaves the order unchanged, and
- if v is a P-node then the automorphism arbitrarily permutes the order of v's children.

The set of orderings of the ground set represented by a PQ-tree T is defined as

$$L(T) = \{\mathrm{fr}(T') \ : \ T' \text{ is obtained from } T \text{ by a legal automorphism}\}$$

Consider, for example, a PQ-tree tree consisting of a P-node root whose children are all the leaves. This PQ-tree represents the set of all orderings of the ground set S, and is therefore called the *universal* PQ-tree for S. Note that for any PQ-tree T, the automorphism that reverses the order of children of every node is legal, so if λ is in $L(T)$ then the reverse of λ is also in $L(T)$.

A special *null* PQ-tree is defined to represent the empty set of orderings.

Let A be a subset of S. An ordering λ of the elements of S is said to *satisfy* A if the elements of A form a consecutive subsequence of λ. For the PQ-tree T, let $\Psi(T, A) = \{\lambda \in L(T) : \lambda \text{ satisfies } A\}$.

Booth and Lueker give an algorithm $\mathrm{REDUCE}(T, A)$ that transforms T into a PQ-tree T' such that $L(T') = \Psi(T, A)$. We call this *reducing* T with respect to the set A. In this context, we call A a *reduction set*. Note that reduction can yield the null tree. The algorithm of Booth and Lueker takes $O(|A|)$ time. Each PQ-tree for a ground set S can be obtained from the universl

PQ-tree by a series of reductions. Moreover, given any non-null PQ-tree, it is easy to read off one of the orderings represented, namely the frontier of the tree. Klein and Reif [29] gave an algorithm MDREDUCE($T, \{A_1, \ldots, A_k\}$) that reduces T with respect to all the nonempty sets A_i simultaneously in the case where the sets A_i are all pairwise disjoint.The algorithm MDREDUCE runs in $O(\log n)$ time using n processors, where n is the size of the ground set of T. The case where the reduction sets are not disjoint was handled in [29], but the algorithm required $O(kn)$ processors and $O(\log k \log^2 n)$ time

In Subsection 2.1 we give a parallel reduction algorithm that handles nondisjoint reduction sets and requires only a linear number of processors. More specifically, the algorithm MREDUCE reduces a PQ-tree T with respect to nonempty subsets $A_1, \ldots, A_k$ in time $O(\log n \cdot (\log n + \log m))$ time using $n+m$ processors, where $m = \sum_i A_i$. A preliminary version of this algorithm was given in [27].

2.1 PQ-tree nondisjoint reduction

The reduction algorithm uses a divide-and-conquer strategy in which recursive calls are made to different parts of the tree in parallel. At each level of recursion a constant number of calls are made to a subroutine for reduction with respect to disjoint sets. These calls serve two purposes. The purpose of one call is to separate out parts of the tree from each other so that the algorithm can recur on them in parallel. Some of the the reduction sets A_i are relevant to two parts of the tree and thus to two recursive calls; such a set gives rise to two subsets, one for each call. Dividing A_i into two subsets and reducing the parts of the tree with respect to these subsets, however, does not completely solve the problem. One must introduce some additional constraints, effectively "gluing" the subsets together insofar as they constrain the PQ-tree. Thus the purpose of two other calls to the subroutine for disjoint reduction is to reduce the tree with respect to two special "gluing" sets that are derived from the original reduction sets A_i.

We first give Lemmas 2.1 and 2.2, which describe some simple properties of orderings. We next give a procedure GLUE which derives two sets from reduction sets A_i. The key property of these gluing sets is described in Lemma 2.3. Next, in Lemma 2.4, we show that reducing with respect to the sets A_i is equivalent to reducing with respect to subsets of these sets that lie in different parts of the tree (and also reducing with respect to the gluing sets). A key subroutine, SUBREDUCE, is then presented that is based on Lemma 2.4. The main algorithm of this section, MREDUCE, is mutually recursive with SUBREDUCE.

Lemma 2.1 Suppose λ satisfies A and B. Then

intersection property: λ satisfies $A \cap B$,

union property: if $A \cap B \neq \emptyset$ then λ satisfies $A \cup B$,

difference-property: if $A \not\supseteq B$ then λ satisfies $A - B$.

Lemma 2.2 Suppose λ satisfies sets B and C, and the leftmost symbol of B in λ coincides with the leftmost symbol of C in λ. Then either $B \subseteq C$ or $C \subseteq B$.

Proof: Write $\lambda = \ldots \epsilon_1\epsilon_2 \ldots$ where ϵ_1 is the leftmost symbol of B in λ and the leftmost symbol of C in λ. Then the subsequence of elements of B in λ is $\epsilon_1 \ldots \epsilon_{|B|}$ and the subsequence of elements of C is $\epsilon_1 \ldots \epsilon_{|C|}$. Thus if $|B| \leq |C|$ then $B \subseteq C$ and if $|C| \leq |B|$ then $C \subseteq B$. $\square$

We say two sets A and B have a *nontrivial intersection* if the sets $A \cap B$, $A - B$, and $B - A$ are all nonempty.

The procedure to construct the gluing sets is as follows.

$\textsc{Glue}(E, \{A_1, \ldots, A_k\})$

- Let $\mathcal{A}$ be the collection of sets in $\{A_1, \ldots, A_k\}$ that have a nontrivial intersection with E.
- If $\mathcal{A}$ is empty, return the pair $(\emptyset, \emptyset)$
- Let A_p be a set in $\mathcal{A}$ that minimizes $|A_p \cap E|$.
- Let A_q be a set in $\mathcal{A}$ that minimizes $|A_q - E|$ subject to the constraint $A_q \cap E \supseteq A_p \cap E$;
- Let $D = A_p \cap A_q$.
- Let $\mathcal{A}'$ be the collection of sets A_i in $\mathcal{A}$ such that $A_i \cap E \not\supseteq A_p \cap E$.
- If $\mathcal{A}'$ is empty then return $(D, \emptyset)$.
- Let A_r be a set in $\mathcal{A}'$ that minimizes $|A_r \cap E|$.
- Let A_s be a set in $\mathcal{A}'$ that minimizes $|A_s - E|$ subject to the constraint $A_s \cap E \supseteq A_r \cap E$.
- Let $F = A_r \cap A_s$.
- Return (D, F).

The sets are represented by a bipartite graph. Each ground set element and each set A_i is a vertex. There is an edge between the ground set element x and the set A_i if $x \in A_i$. To determine which sets A_i have a nontrivial intersection with E, we first mark the ground set elements that belong to E. Then we determine, for each set A_i, how many marked elements A_i is adjacent to in the bipartite graph. If the number is bigger than zero but smaller than $|A_i|$ and smaller than $|E|$, then A_i has a nontrivial intersection with E. The other steps of the algorithm $\textsc{Glue}$ can be implemented in a similar way, using a constant number of marking and counting operations and finding the minimum among k numbers.

Let n denote the size of the ground set and let $m = \sum_{i=1}^{k} |A_i|$. Then the size of the bipartite graph is $O(n+m)$, and each such operation can be done in $O(\log(n+m))$ time using $(n+m)/\log(n+m)$ processors. Thus $\textsc{Glue}(E, \{A_1, \ldots, A_k\})$ takes $O(\log(n+m))$ time using $(n+m)/\log(n+m)$ processors.

The proof of the following lemma is somewhat technical and can be skipped on a first reading.

Lemma 2.3 Suppose there exists some ordering satisfying $E, A_1, \ldots, A_k$. Let $(D, F) = \textsc{Glue}(E, \{A_1, \ldots, A_k\})$. For $i = 1, \ldots, k$, if A_i has a nontrivial intersection with E, then either $D \subseteq A_i$ or $F \subseteq A_i$.

Proof: Let σ be the ordering satisfying $E, A_1, \ldots, A_k$. Let us write

$$\sigma = \ldots \alpha_1 \epsilon_1 \ldots \epsilon_2 \alpha_2 \ldots$$

where α_1 is the last symbol before the elements of E, α_2 is the first symbol after the elements of E, and ϵ_1 and ϵ_2 are, respectively, the first and last symbols of E in σ.

Suppose $\mathcal{A}$ is nonempty. In this case, $D = A_p \cap A_q$. Since A_p contains at least one element of E and at least one symbol not in E, it contains two adjacent symbols, one in E and one not in E. Thus either $\alpha_1, \epsilon_1 \in A_p$ or $\alpha_2, \epsilon_2 \in A_p$. Assume without loss of generality that $\alpha_1, \epsilon_1 \in A_p$ (else replace σ with the reverse of σ). Since $A_q \cap E \supseteq A_p \cap E$, we have $\epsilon_1 \in A_q$. Since A_q also contains at least one symbol not in E, either $\alpha_1 \in A_q$ or $\alpha_2 \in A_q$. Since A_q is satisfied by σ, if α_2 were in A_q then all of E would also be in A_q, contradicting the fact that A_q has a nontrivial intersection with E. Thus $\alpha_1 \in A_q$. We conclude that if D is defined then $\alpha_1, \epsilon_1 \in D$.

Furthermore, since α_1 is the rightmost symbol in σ of both $A_p - E$ and $A_q - E$, by applying Lemma 2.2 to the reverse of σ, we infer that either $A_q - E \subseteq A_p - E$ or $A_p - E \subset A_q - E$. The second inclusion would imply $|A_p - E| < |A_q - E|$, which would contradict the choice of A_q. Thus we have $A_q - E \subseteq A_p - E$. By choice of A_q we have $A_p \cap E \subseteq A_q \cap E$. We conclude that $(A_q - E) \cup (A_p \cap E) = A_q \cap A_p$, which in turn is D.

Suppose $\mathcal{A}'$ is also nonempty, so $F = A_r \cap A_s$. An analogous argument shows that either $\alpha_1, \epsilon_1 \in F$ or $\alpha_2, \epsilon_2 \in F$. Assume for a contradiction that the former holds. Then ϵ_1 is the leftmost symbol in σ of $A_p \cap E$ and of $A_r \cap E$. Hence by Lemma 2.2, either $A_p \cap E \subseteq A_r \cap E$ or $A_r \cap E \subset A_p \cap E$. The first inclusion would violate the choice of A_r. The second inclusion would imply $|A_r \cap E| < |A_p \cap E|$, which would violate the choice of A_p. This proves that if F is defined then $\alpha_2, \epsilon_2 \in F$.

To complete the proof, suppose A_i has a nontrivial intersection with E. We must show that either $D \subseteq A_i$ or $F \subseteq A_i$.

Since $\mathcal{A}$ is nonempty (it certainly contains A_i), $D = A_p \cap A_q$. Since σ satisfies A_i, which contains some symbols in E and some not in E, either $\alpha_1, \epsilon_1 \in A_i$ or $\alpha_2, \epsilon_2 \in A_i$.

Case I: $\alpha_1, \epsilon_1 \in A_i$. Then ϵ_1 is the first symbol in σ of both $A_i \cap E$ and $A_p \cap E$. By Lemma 2.2, therefore, either $A_p \cap E \subseteq A_i \cap E$ or $A_i \subset A_p \cap E$. The second inclusion would imply $|A_i \cap E| < |A_p \cap E|$, contradicting the choice of A_p. Hence we have $A_p \cap E \subseteq A_i \cap E$. Also, α_1 is the last symbol in σ of both $A_i - E$ and $A_q - E$. By applying Lemma 2.2 to the reverse of σ, we infer that either $A_q - E \subseteq A_i - E$ or $A_i - E \subset A_q - E$. The second inclusion would imply $|A_i - E| < |A_q - E|$, contradicting the choice of A_q. Hence we have $A_q - E \subseteq A_i - E$. We infer $(A_q - E) \cup (A_p \cap E) \subseteq (A_i - E) \cup (A_i \cap E)$. Thus $D \subseteq A_i$.

Case II: $\alpha_2, \epsilon_2 \in A_i$. Since $\epsilon_1 \notin A_i \cap E$, we have $A_i \cap E \not\supseteq A_p \cap E$. Therefore $\mathcal{A}'$ is nonempty, so $F = A_r \cap A_s$. By essentially the same argument as in case I, $F \subseteq A_i$. Thus the claim is proved.

The following lemma is the basis for our reduction algorithm.

Lemma 2.4 Let $E, A_1, \ldots, A_k$ be subsets of the ground set of T. Suppose there exists some ordering satisfying all these sets. Let $(D, F) = \text{GLUE}(E, \{A_1, \ldots, A_k\})$. Then an ordering λ satisfies these sets if and only if the following conditions hold.

1. λ satisfies E,

2. λ satisfies $E \cap A_i$ for all i,

3. λ satisfies $\widehat{A}_i$ for all i, where

$$\widehat{A}_i = \begin{cases} A_i - E & \text{if } A_i \text{ and } E \text{ have a nontrivial intersection} \\ A_i & \text{otherwise} \end{cases}$$

4. λ satisfies D and F.

Proof: First we prove the "only-if" direction. Suppose λ satisfies $A_1, \ldots, A_k, E$. Then condition 1 follows trivially. Furthermore, condition 2 follows from the intersection property of Lemma 2.1 and condition 3 follows from the difference property. If D is nonempty, it is the intersection of two sets A_p and A_q that are satisfied by λ; hence by the intersection property D is itself satisfied by λ. Similarly F is satisfied by λ. Thus condition 4 holds.

Now we prove the "if" direction. Suppose conditions 1 through 4 hold of λ. Clearly λ satisfies E. We show that λ satisfies A_i $(i = 1, \ldots, k)$.

By condition 2, λ satisfies $A_i \cap E$. By condition 3 we have that λ satisfies $\widehat{A}_i$. If A_i has a trivial intersection with E then $\widehat{A}_i = A_i$, so we are done. Assume therefore that A_i and E have a nontrivial intersection. In this case, $\widehat{A}_i$ is $A_i - E$. By condition 4, D and F are satisfied by λ. By Lemma 2.3 either $D \subseteq A_i$ or $F \subseteq A_i$. In the first case, since $A_i - E$ and $A_i \cap E$ both intersect D, it follows from the intersection property of Lemma 2.1 that λ satisfies the union $(A_i - E) \cup D \cup (A_i \cap E)$, which is just A_i. In the case where $F \subseteq A_i$, the proof is analogous. $\Box$

Definition 2.5 For a node v of a PQ-tree T, $leaves_T(v)$ denotes the set of pendant leaves of v, i.e. leaves of T having v as ancestor. Let $lca_T(A)$ denote the least common ancestor in T of the leaves belonging to A. Suppose that $v = lca_T(A)$ has children $v_1 \ldots v_s$ in order. We say A is *contiguous* in T if

- v is a Q-node, and for some consecutive subsequence $v_p \ldots v_q$ of the children of v, $A = \bigcup_{p \leq i \leq q} leaves(v_i)$, or
- v is a P-node or a leaf, and $A = leaves(v)$.

For example, in Figure 7, the set $\{a, f, d\}$ is contiguous. Also, the set $\{a, f, d, c\}$ is contiguous, as is the set $\{c, e\}$. The set $\{f, d, c\}$ is not contiguous, nor is $\{b, c\}$.

The significance of contiguity is as follows. Lueker and Booth ([32]; also Lemma 2.1 of [29]) prove that if every ordering in $L(T)$ satisfies some set E then E is contiguous in T.

Suppose that E is indeed contiguous in T. The *E-pertinent subtree of T with respect to E* is the subtree consisting of $lca_T(E)$ and those children of $lca_T(E)$ whose descendents are in E. Note that the E-pertinent subtree is a PQ-tree over the ground set E. We denote this tree by $T|E$.

For a set A, define

$$A_i|E = \begin{cases} A_i \cap E & \text{if } A_i \cap E \neq E \\ \emptyset & \text{if } A_i \cap E = E \end{cases}$$

Remark 2.6 Suppose we modify the tree T by reducing its E-pertinent subtree with respect to a subset of E. It follows directly from the PQ-tree definitions that the result is the same as if we had reduced the whole tree T with respect to this subset. (If the reduction of the E-pertinent subtree yields the null tree, then we replace T with the null tree.)

The above observation suggest that our algorithm might profitably operate in parallel on smaller disjoint subtrees of a PQ-tree T. It is also useful to operate on a tree obtained by deleting a subtree from T.

Let $\star_E$ denote $lca_T(E)$. Let T/E denote the subtree of T obtained by omitting all the proper descendants of v that are ancestors of elements of E. Then T_0 is a PQ-tree whose ground set is $S - E \cup \{\star_E\}$. For a set A, define

$$A_i/E = \begin{cases} A_i - E \cup \{\star_E\} & \text{if } A_i \supseteq E \\ A_i - E & \text{otherwise} \end{cases}$$

Remark 2.7 Suppose that either $A \supseteq E$ or $A \cap E = \emptyset$. It follows from Lemma 2.18 of [29] that if we reduce T/E with respect to A/E, the effect on T is the same as if we had reduced T with respect to A. (Again, if the reduction of T/E yields the null tree, then we replace T with the null tree.)

Based on the above observations, we give a subroutine SUBREDUCE used in our algorithm MREDUCE for nondisjoint reduction. The subroutine SUBREDUCE and the main routine MREDUCE are mutually recursive. SUBREDUCE is designed in accordance with Lemma 2.4.

SUBREDUCE$(T, E, \{A_1, \ldots, A_k\})$:

1. Reduce T with respect to E using MDREDUCE.
2. Let $(D, F) :=$ Glue$(E, \{A_1, \ldots, A_k\})$.
3. If D is well-defined, reduce T with respect to D using MDREDUCE.
4. If F is well-defined, reduce T with respect to F using MDREDUCE.
5. In parallel, make the following recursive calls to Mreduce:
   (a) Modify T by calling MREDUCE$(T|E, \{A_1|E, \ldots, A_k|E\})$
   (b) Modify T by calling MREDUCE$(T/E, \{A_1/E, \ldots, A_k/E\})$.
6. Check that in the frontier of the resulting tree T, each reduction set A_i is consecutive. If so, return T. If not, return the null tree.

Each step of SUBREDUCE except for the recursive calls takes $O(\log(n + m))$ time using $n + m$ processors.

We now prove that the effect of SUBREDUCE$(T, E, \{A_1, \ldots, A_k\})$ is to reduce T with respect to $E, A_1, \ldots, A_k$. Step 1 reduces T with respect to E and A_p. Steps 3 and 4 reduce T with respect to the sets D and F. Let us assume inductively that the calls to MREDUCE correctly reduce the PQ-trees $T|E$ and T/E with respect to the given reduction sets.

In step 5a, $T|E$ is reduced with respect to the sets $A_i|E$. Since these sets are subsets of E, as discussed in Remark 2.6, this has the effect of reducing T with respect to the same sets. If $A_i \cap E \neq E$ then $A_i|E = A_i \cap E$. Thus in this case the effect is to reduce T with respect to $A_i \cap E$. If $A_i \cap E = E$ then $A_i|E = \emptyset$ so the reduction has no effect but reducing T with respect to E has no effect either, since T was already reduced with respect to E in step 1. Thus in either case the effect is that of reducing T with respect to $A_i \cap E$.

In step 5b, T/E is reduced with respect to the sets A_i/E defined immediately before Remark 2.7. If A_i has a nontrivial intersection with E then certainly $A_i \not\supseteq E$, so $A_i/E = A_i - E$. If A_i has a trivial intersection with E then one of the following three cases must hold: $A_i \subseteq E$, $A_i \supseteq E$, and $A_i \cap E = \emptyset$. In the first case, $A_i/E = \emptyset$, so reducing T/E with respect to A_i/E has no effect. In the second and third cases, by Remark 2.6 the effect of reducing T/E by A_i/E is to reduce T with respect to A_i.

Thus in general the effect is to reduce T with respect to the set $\widehat{A}_i$ defined in condition 3 of Lemma 2.4. It follows by that lemma that if there exists an ordering $L(T)$ satisfying $E, A_1, \ldots, A_k$ then the effect of the entire call is to reduce T with respect to these reduction sets. Thus if there exists such an ordering, the resulting PQ-tree represents all such orderings. Moreover, in this case the frontier of that PQ-tree is one such orderings, so the resulting PQ-tree is returned. Conversely, if no such ordering exists, the frontier of the PQ-tree will certainly not be such an ordering. This shows the correctness of the procedure SUBREDUCE.

The algorithm MREDUCE uses the subroutine SUBREDUCE in conjuction with a technique for choosing E, the second argument to SUBREDUCE, so that it consists of roughly half the elements of the ground set of T. This choice ensures that the recursion depth of MREDUCE is logarithmic.

Before giving the algorithm, we discuss the notion of the *intersection graph* of a collection of sets. Let $\mathcal{F}$ be a family of subsets $A_1, \ldots, A_k$ of S. The intersection graph of $\mathcal{F}$ is a graph whose nodes are the sets A_i and where two sets are considered adjacent if they intersect. In the present context the significance of the intersection graph is given by the following easy corollary to the union property of of Lemma 2.1.

Corollary 2.8 Suppose an ordering λ of S satisfies the sets $A_1, \ldots, A_k$, and the intersection graph of these sets is connected. Then λ satisfies their union $\cup_i A_i$.

Implementation Note 4 *Note that the intersection graph of $\mathcal{F}$ may have a number of edges greatly exceeding the sum of the cardinalities of the sets in $\mathcal{F}$. Therefore, to efficiently compute the connected components of the intersection graph, we construct an auxiliary bipartite graph as described in connection with the procedure* GLUE. *The auxiliary graph has node-set $\mathcal{F} \cup S$, and there is an edge between a set in $\mathcal{F}$ and an element of S if the element belongs to the set. Two sets in $\mathcal{F}$ are in the same connected component of the intersection graph if they are in the same component of the auxiliary graph. Moreover, a spanning forest of $\mathcal{F}$ can easily be obtained from a spanning forest of the auxiliary graph. Note that the number of edges in the auxiliary graph is just the sum of the cardinalities of the sets in $\mathcal{F}$. Thus, by using a standard connectivity algorithm [19, 43] on the auxiliary graph, we can obtain the connected components and spanning forest of the intersection graph of $\mathcal{F}$ in time $O(\log |\mathcal{F} \cup S|)$ using $|\mathcal{F} \cup S|$ processors (or $|\mathcal{F} \cup S| / \log |\mathcal{F} \cup S|$ processors using randomization).*

We finally give the algorithm for multiple nondisjoint reduction.

MREDUCE($T, \{A_1, \ldots, A_k\}$):

1. Purge the collection of input sets A_i of empty sets. If no sets remain, return.
2. Let n be the size of the ground set of T. If $n \leq 4$, carry out the reductions one by one.
3. Otherwise, let $\mathcal{A}$ be the family of (nonempty) sets A_i. Let $\mathcal{S}$ consist of the sets A_i such that $|A_i| \leq n/2$. We call such sets "small.". Let $\mathcal{L}$ be the remaining, "large," sets in $\mathcal{A}$. Find the connected components of the intersection graph of $\mathcal{A}$, find a spanning forest of the intersection graph of $\mathcal{S}$, and find the intersection $\bigcap \mathcal{L}$ of the large sets. Let E_1
4. Proceed according to the following four cases:

Case I: The intersection graph of $\mathcal{A}$ is disconnected. In this case, let $\mathcal{C}_1, \ldots, \mathcal{C}_r$ be the connected components of $\mathcal{A}$. For $i = 1, \ldots, r$, let E_i be the union of sets in the connected component $\mathcal{C}_i$. Call MDREDUCE to reduce T with respect to the disjoint sets $E_1, \ldots, E_r$. Next, for each $i = 1, \ldots, r$ in parallel, recursively call MREDUCE($T|E_i, \mathcal{C}_i$).

Case II: The union of sets in some connected component of $\mathcal{S}$ has cardinality at least $n/4$. In this case, from the small sets making up this large connected component, select a subset whose union has cardinality between $n/4$ and $3n/4$. (See Implementation Note 5). Let E be this union, and call SUBREDUCE($T, E, \{A_1, \ldots, A_k\}$).

Case III: The cardinality of the intersection of the large sets is at most $3n/4$. In this case, from the large sets choose a subset of the large sets whose intersection has cardinality between $n/4$ and $3n/4$. (See Implementation Note 6.) Let E be this intersection, and call SUBREDUCE($T, E, \{A_1, \ldots, A_k\}$).

Case IV: The other cases do not hold. In this case, let E be the intersection of the large sets, and call SUBREDUCE($T, E, \{A_1, \ldots, A_k\}$).

Implementation Note 5 *In this note we address the problem arising in case II, selecting some of the sets making up a component of size at least $n/4$. Each of these sets has size at most $n/2$, and our goal is that the union of the sets chosen has cardinality between $n/4$ and $3n/4$. A spanning tree of the component has been computed in step 3. For each of the sets comprising the component, compute the distance in the spanning tree from the root. These distances can be obtained using the Euler-tour technique [44]. Sort the sets according to distance, and let $B_1, \ldots, B_s$ be the sorted sequence. Observe that any initial subsequence $B_1, \ldots, B_i$ of this sequence is connected. Let $\hat{\imath}$ be the minimum i such that the union $\bigcup_{j=1}^{i} B_j$ has cardinality $\geq n/4$. Since each set is small, it follows that the cardinality of the union is no more than $3n/4$.*

Implementation Note 6 *In this note we address the problem arising in case III, selecting a subset of the large sets. Our goal is that the intersection of the selected subset of sets has cardinality between $n/4$ and $3n/4$. Order the large sets arbitrarily, and let $\hat{\imath}$ be the maximum i such that the intersection of the first i sets has cardinality at least $n/4$. Since each set has size at least $n/2$, the intersection of the first $\hat{\imath}$ has at most $n/2$ elements not appearing in the intersection of the first $\hat{\imath} + 1$ sets. The latter intersection has cardinality less than $n/4$, so the intersection of the first $\hat{\imath}$ sets has cardinality less than $3n/4$.*

First we address the correctness of the procedure MREDUCE.

Lemma 2.9 The PQ-tree T' returned by $\text{MREDUCE}(T, \{A_1, \ldots, A_k\})$ satisfies

$$L(T') = \{\lambda \in L(T) \ : \ \lambda \text{ satisfies } A_1, \ldots, A_k\}$$

Proof: Let us assume inductively that the calls to SUBREDUCE in cases II, III, and IV correctly carry out the reductions of T with respect to $E, A_1, \ldots, A_k$. To verify the correctness of the call to MREDUCE in cases II through IV, therefore, we must only check that in fact reducing T with respect to these sets is equivalent to reducing T with respect to the sets $A_1, \ldots, A_k$. That is, we must prove the assertion that any ordering satisfying $A_1, \ldots, A_k$ also satisfies E.

In case II, since E is the union of a connected subcollection of the collection of reduction sets A_i, the truth of the assertion follows from the union property of Lemma 2.1. In cases III and IV, since E is the intersection of some of the A_i, the truth of the assertion follows from the intersection property of Lemma 2.1.

Next we address the correctness in case I. For each component $\mathcal{C}_i$ of the intersection graph of $\mathcal{A}$, we let E_i be the union of sets in $\mathcal{C}_i$. By the union property of Lemma 2.1, any ordering satisfying $A_1, \ldots, A_k$ also satisfies the sets $E_1, \ldots, E_r$. Hence reducing T with respect to $A_1, \ldots, A_k$ is equivalent to first reducing T with respect to $E_1, \ldots, E_r$ and then reducing with respect to $A_1, \ldots, A_k$. Furthermore, by Remark 2.6, reducing with respect to $A_1, \ldots, A_k$ is equivalent to the reductions carried out in case I, namely reducing each subtree $T|E_i$ with respect to the family $\mathcal{C}_i$ of sets whose union is E_i. We assume inductively that the these reductions are correctly carried out by the recursive calls to MREDUCE. This argument proves the correctness of MREDUCE in case I. □

Now we analyze the time and processor requirements of MREDUCE.

Theorem 2.10 Consider an invocation $\text{MREDUCE}(T, \{A_1, \ldots, A_k\})$. Let n be the size of T's ground set, and let $m = \sum_i |A_i|$. The number of levels of recursion is $O(\log n)$. At each level, the sum of sizes of all ground elements in all PQ-trees is $O(n)$ and the sum of sizes of all reduction sets is $O(m)$.

We analyze the algorithm by considering the tree R of recursive calls to MREDUCE. The root of R is the initial invocation of the procedure, and the other vertices of R are all the subsequent recursive invocations. The children of an invocation v are the invocations called by v or by an invocation of SUBREDUCE called by v. Let $T(v)$ denote the PQ-tree to which the invocation v is applied. Let $n(v)$ denote the size of the ground set of $T(v)$. Let $p(v)$ denote the parent of v in R.

The proof of Theorem 2.10 consists of three parts. In Lemma 2.11, we bound the recursion depth. In Lemma 2.12, we show that, at any level of recursion, the sum of sizes of all ground sets is $O(n)$. In Lemma 2.13, we show that at every level of recursion the sum of sizes of all reduction sets is $O(m)$.

Lemma 2.11 [Bounding the recursion depth] The depth of recursion is $O(\log n)$, where n is the size of the ground set of the initial input PQ-tree.

Proof: Say a a vertex v is *smaller than* a vertex w if $n(v) \leq 3n(w)/4 + 1$. Consider a vertex w. If w is a case II or case III invocation, the choice of E in these cases is such that $n(w)/4 \leq |E| \leq$

$3n(w)/4$. The children of w in these cases involve the PQ-trees $T|E$ and T/E. The ground set of $T|E$ is E, so it has size at most $3n(w)/4$. The ground set of T/E is the ground set of T minus the set E, together with the element $\star_E$, so it has size at most $3n(w)/4 + 1$.

Suppose w is a case IV invocation. The children of w involve the PQ-trees $T|E$ and T/E. In this case E is the intersection of the large sets. Since case III does not hold, the cardinality of E is larger than $3n(w)/4$. Hence the ground set of T/E has size at most $n(w)/4$. Thus the corresponding child is smaller than w. Consider the other child u. Its ground set is E, and its reduction sets are the sets $A_i \cap E$ that are strictly contained in E. Since E is the intersection of the large sets, only small sets A_i have the property that $A_i \cap E$ is strictly contained within E. It follows that the connected components of the reduction sets of u are contained within the connected components of the small sets of w. Since case II does not hold, each of these connected components has size at most $n(w)/2$. Thus u is either a case I invocation, in which case all its children are smaller than w, or the reduction sets of u form a single connected component, in which case u is a case II invocation.

Summarizing, if w is case II or III then its children are all smaller than it, and if w is case IV then its grandchildren are all smaller than it. Finally, if w is a case I invocation, none of its children is a case I invocation, so its great-grandchildren are all smaller than it. We infer that the number of levels of recursion is $O(\log n)$. $\square$

Lemma 2.12 [Bounding the sum of sizes of ground sets] For any level of recursion, the sum of sizes of ground sets of all PQ-trees being recursed on is $O(n)$, where n is the size of the ground set of the initial input PQ-tree.

Proof: We show that the ground sets of all PQ-trees at a given level of recursion are disjoint and that all but n of the elements are in the ground set of the initial input PQ-tree.

For any vertex w that is case I, the ground sets of the children of w are disjoint subsets of the ground set of w. Hence no new ground set elements are introduced by a case I vertex. Each vertex w that is case II, III, or IV has two children, one working on $T(w)|E$ and one working on $T(w)/E$. The ground set of the first child is E, a subset of the ground set of $T(w)$, and that of the second is the ground set of T minus the set E, together with a new element $\star_E$.

We see that every vertex's children have disjoint ground sets. It follows by induction on j that the ground sets of all PQ-trees at level j of the recursion tree R are disjoint.

We have also seen that each vertex introduces at most one new ground set element, and that only vertices with two children introduce such new elements. Furthermore, each vertex has a nonempty ground set. Let R' denote the recursion tree R truncated at some level j. Let s be the number of ground set elements at that level that do not belong to the ground set of the initial input PQ-tree. Then the total number of ground set elements is $n + s$. Since each PQ-tree has a nonempty ground set and the ground sets of the leaves of R' are all disjoint, the number of leaves of R' is at most $n + s$. Hence the number of internal vertices having two or more children is at most $(n + s)/2$. The number of new ground elements is at most the number of internal vertices having two or more children, so $s \leq (n + s)/2$. It follows that $s \leq n$. $\square$

Lemma 2.13 [Bounding the sum of sizes of all reduction sets] At any level of recursion, the sum of sizes of all reduction sets is $O(m)$, where m is the sum of sizes of reduction sets in the initial invocation of MREDUCE.

Proof: We analyze the way reduction sets for one level of recursion are transformed by SUBREDUCE into reduction sets at the next level of recursion. There is a forest that represents this process. The vertices of the forest are pairs (invocation, reduction set). Consider one invocation $u = \text{MREDUCE}(T, \{A_1, \ldots, A_k\})$. Depending on which case arises during this invocation, each reduction set A_i gives rise to one or two reduction sets in child invocations. In case I, each reduction set A_i gives rise to one reduction set, namely A_i, in some child invocation v. In this case, the only child of (u, A_i) is (v, A_i). In cases II through IV, each reduction set A_i gives rise to two, $A_i|E$ and A_i/E. In these cases, the children of (u, A_i) are the pairs $(v, A_i|E)$ and $(w, A_i/E)$, where v and w are the appropriate children of the invocation u. Note that the reduction set A_i gives rise to disjoint reduction sets. Moreover, in cases II through IV a new element $(\star_E)$ may be included in A_i/E. Thus each vertex introduces at most one new element, and only vertices with two children introduce such new elements.

The remainder of the proof is similar to that of Lemma 2.12. Let Q' be the forest Q truncated at some recursion level j. We focus on the reduction sets in the leaves of Q'. Let s be the number of occurences in the leaf reduction sets of elements not belonging to the reduction sets of the original invocation. Then the sum of sizes of all the leaf reduction sets is $m + s$. Since empty reduction sets are discarded, we may assume that every leaf of Q' has a nonempty reduction set. Thus the number of leaves is at most $m + s$. Hence the number of internal vertices with two children is at most $(m + s)/2$. As in the proof of Lemma 2.12, it follows that $s \leq m$. □

This completes the proof of Theorem 2.10. Since each level of MREDUCE takes $O(\log(n + m))$ time and the recursion depth is $O(\log n)$, the total time required is $O(\log n \log(n + m))$ using $O(n + m)$ processors.

3 Applications

In this section, we show how having a `peo` for a chordal graph enables one to solve many problems efficiently in parallel. The key to our efficient algorithms is our use of the *elimination tree*. The elimination tree is a structure introduced by [42] in the context of sparse Gaussian elimination, but implicit in the work of others, including [40]. In Subsection 3.1, we show that the elimination tree determined by a `peo` of a chordal graph has useful separation properties. Most of the chordal graph algorithms described in this chapter rely on the elimination tree.

3.1 The elimination tree determined by a `peo`

Let \$ be a one-to-one numbering of the nodes of the connected graph G. As in Section 1, we shall say v is *richer* than u, and u is *poorer* than v, if the number assigned to v is higher than that assigned to u. We define the *elimination tree* $T(G_\$)$ of $G_\$$ as follows. For every node v except the highest-numbered, v's parent $p(v)$ is defined to be the poorest neighbor of v that is richer than v. The tree $T(G_\$)$ can easily be constructed from $G_\$$ in $O(\log n)$ time using $(n + m)/\log n$ processors.

Since parents are richer than their children, there are no directed cycles in $T(G_\$)$. Since each vertex (except the richest) has exactly one parent, $T(G_\$)$ is in fact a tree. Recall that a `peo` of a chordal graph is a numbering of the nodes of the graph such that for each node v, the richer

Figure 8: The existence of a cross-edge e connecting u and v implies the existence of a cross-edge e' connecting w and v.

Figure 9: When the node v and its richer neighbors are removed, the subtrees rooted at children of v become separated from each other and from the remainder of the graph.

neighbors of v form a clique. Define a *cross-edge* to be an edge of G such that neither endpoint is an ancestor of the other in $T(G_\$)$. If there are no cross-edges, we call $T(G_\$)$ a *depth-first search tree*. If \$ is a peo, the existence of a cross-edge between the node u and a poorer node v implies the existence of a cross-edge between u and the parent of v; using induction on the distance in the tree between endpoints of an edge, we can prove the following lemma.

Lemma 3.1 Let G be a chordal graph. If \$ is a peo of G, then $T(G_\$)$ has no cross-edges.

Proof: Let u and v be two nodes; we show that there is no cross-edge between u and v by induction on the length of the path in $T(G_\$)$ connecting u and v. If this length is 1, v is the parent of u or vice versa, so an edge between them is not a cross-edge. Therefore, assume the length is greater than 1.

Suppose e is an edge between u and v. Assume without loss of generality that v is richer than u. Let w be the parent of u; by choice of parent, w is richer than u but poorer than v. See Figure 8. Using e, we can form a backward path through u with one endpoint w and the other v, proving via the Backward Path Theorem the existence of an edge e' between w and v. By the inductive hypothesis, e' is not a cross-edge, so v must be an ancestor of w in $T(G_\$)$. This shows that e is not a cross-edge. □

We next show $T(G_\$)$ has desirable separation properties. For a node v, let $T_v(G_\$)$ denote the subtree of $T(G_\$)$ rooted at v. As illustrated in Figure 9, removing v and its richer neighbors separates the subtrees rooted at children of v from the remainder of the graph.

Lemma 3.2 Let \$ be a peo of G. Let v be a node of G, with children $v_1, \ldots, v_k$ in $T(G_\$)$. Let K be the clique of G consisting of v and its richer neighbors. Then $G[T_{v_i}(G_\$)]$ is a connected component of $G - K$, for $i = 1, \ldots, k$.

Proof: To see that $G[T_{v_i}(G_\$)]$ is connected in G, note that edges in $T_{v_i}(G_\$)$ are edges in G, and hence $T_{v_i}(G_\$)$ is a spanning tree of $G[T_{v_i}(G_\$)]$. None of the nodes in $T_{v_i}(G_\$)$ are in K, so $G[T_{v_i}(G_\$)]$ remains connected when K is removed from G.

Suppose there is an edge between a node v' in $T_{v_i}(G_\$)$ and a node w not in $K \cup T_{v_i}(G_\$)$. The edge cannot be a cross-edge in $T(G_\$)$, so w must be an ancestor of v'; since w is not in $T_{v_i}(G_\$)$, it must be an ancestor of v as well. Using the edge, we can construct a backward path from w through v' and up the tree $T(G_\$)$ to v. By the Backward Path Lemma, w must be adjacent to v, so w belongs to K, a contradiction. □

As a corollary to Lemma 3.2, we can show that a chordal graph has a clique whose removal breaks the graph into pieces of at most half the size. (This fact was first shown by [21].) Let the node v of Lemma 3.2 be the lowest node in the elimination tree having more than $n/2$ descendents. Then every component $G[T_{v_i}(G_\$)]$ has at most $n/2$ nodes, but together these components comprise at least $n/2$ nodes. Hence the clique consisting of v together with its richer neighbors forms a separating clique. We use this idea below in our algorithm for finding an optimal coloring.

3.2 Recognition

A recognition algorithm for chordal graphs follows easily from the peo algorithm. When the peo algorithm produces a total ordering \$ of G, the correctness of the algorithm implies that if G is chordal then \$ is a peo; of course, if \$ is a peo, then G is chordal. It therefore suffices to check whether \$ is a peo. We can parallelize a technique used in [40]. Each node v sends to its parent $p(v)$ in $T(G_\$)$ a list of v's richer neighbors (excluding $p(v)$). Then each node w sorts the elements of all the lists it received, together with w's own adjacency list, and verifies that it is a neighbor of every node on every list it received.

Claim: The numbering \$ is a peo if and only if no verification step fails.

Proof: Suppose \$ is a peo. Then for every node v, the richer neighbors of v form a clique. In particular, the parent of v is adjacent to all v's other richer neighbors. Thus every verification step succeeds.

Suppose no verification step fails. We claim that for each node v, the richer neighbors of v form a clique. The proof is reverse induction on the depth d of v in the tree $T(G_\$)$. The claim is trivial for $d = 0$, because the root has no richer neighbors. Suppose the claim holds for d, and let v be a node at depth $d + 1$. By the inductive hypothesis, $p(v)$ and its richer neighbors form a clique K. By the success of $p(v)$'s verification step, every richer neighbor of v is a neighbor of $p(v)$, and hence lies in K. This proves the induction step. □

The claim shows that we can determine whether a given ordering is a peo of G. The time for carrying out verification is $O(\log n)$ using $n + m$ processors. For subsequent applications, assume that the numbering \$ is a peo of the chordal graph G.

3.3 Maximum-weight clique

Fulkerson and Gross observed that every maximal clique S of G is of the form

$$\{v\} \cup \{\text{richer neighbors of } v\}$$

To see this, we need only let v be the poorest node of S. It follows that the maximal cliques of G can be determined from \$. Suppose each node is assigned a non-negative weight. As Gavril observed, any maximum-weight clique is maximal, so a maximum-weight cliques may easily be determined from \$.

3.4 Depth-first and breadth-first search trees

We showed in Subsection 3.1 that the elimination tree determined by a peo \$ is a depth-first search tree. To obtain a breadth-first search tree of G, we construct a tree similar to the elimination tree, by choosing the parent of each node v (except the richest node) to be the richest neighbor of v. Let T be the resulting tree, rooted at the richest node, which we shall denote by r. Our proof that T is a breadth-first search tree relies on two claims.

Claim 3.3 For each node v the shortest path from v to r in G is monotonically increasing in wealth.

Proof: Any subpath whose internal nodes are poorer than its endpoints can be replaced by a direct edge between the endpoints, by the validity of \$. □

For the second claim, let $d(v)$ denote the length of the shortest path in G from v to r.

Claim 3.4 If w is a descendent of v in the elimination tree, then $d(w) \geq d(v)$.

Proof: The proof is by reverse induction on the wealth of w. The basis, in which $w = r$, is trivial. Otherwise, let w' be the second node on a shortest path in G from w to r, so $d(w) = 1 + d(w')$. By Claim 3.3, w' is richer than w, hence an ancestor of w in the elimination tree by Lemma 3.1. If w' is a descendent of v in the elimination tree, then $d(w') \geq d(v)$ by the inductive hypothesis. If w' is an ancestor of v, then there is a backward path from v back along tree edges to w, and then forward to w', proving by validity of \$ that v is adjacent to w' in G, and hence that $d(v) \leq 1 + d(w')$. □

For each node $v \neq r$, let $p(v)$ be the richest neighbor of v in G. Any other neighbor w of v is a descendent of $p(v)$, so $d(w) \geq d(p(v))$ by Claim 3.4. It follows that $p(v)$ is the second node in a shortest path in G from v to the richest node of G. Thus the tree defined by $p(\cdot)$ is a breadth-first search tree.

3.5 Maximum independent set

Gavril showed that a maximum independent set $\mathcal{I}$ of the chordal graph G is obtained by the greedy maximal-independent-set algorithm when applied to nodes in order of the peo \$ $= v_1 \ldots v_n$. His algorithm proceeds as follows. First put v_1 into $\mathcal{I}$, and delete v_1 and its neighbors. Next, put the poorest remaining node in $\mathcal{I}$, and so forth. Once $\mathcal{I}$ has been found, the family of cliques of

PMIS

P1 For each node v, let $b_0[v]$ denote the lowest ancestor of v that is not adjacent to v (or else v).

P2 For stages $k = 0, \ldots, \lceil \log n \rceil - 1$, for each node v, let $b_{k+1}[v] := b_k[b_k[v]]$.

P3 Mark the leaf x as being in the independent set.

P4 For stages $k = \lceil \log n \rceil - 1, \lceil \log n \rceil - 2, \ldots, 0$, for each marked node v, mark $b_k[v]$.

Figure 10: A simple implementation of the algorithm PMIS for finding a maximum independent set when the elimination tree is a path.

the form $\{x\} \cup \{\text{richer neighbors of } x\}$ for $x \in \mathcal{I}$ is a clique cover (a set of cliques whose union contains all the nodes). Because any independent set has size at most that of any clique cover, it follows that the above procedure has identified a *maximum* independent set and a *minimum* clique cover.

We want to simulate Gavril's sequential greedy algorithm in parallel. First suppose that the elimination tree $T(G_\$)$ is a path with leaf x. In this case, we give a simple algorithm PMIS for simulating Gavril's algorithm. For each node v, let $b[v]$ be the lowest ancestor of v in $T(G_\$)$ that is not adjacent to v in G (or v, if no such ancestor exists).

Claim 3.5 The greedy independent set consists of $x, b[x], b[b[x]]$, and so on.

This set can be determined quickly in parallel using standard pointer-jumping techniques. The implementation shown in Figure 10 requires $O(\log n)$ time, m processors, and $O(m \log n)$ space; use of more sophisticated techniques (e.g. [3], [10]) achieve the same time bound using only $m/\log n$ processors and $O(m)$ space.

Proof of Claim: Suppose we put x into $\mathcal{I}$ and delete the neighbors of x. The node $b[x]$ is by definition the poorest undeleted node. Moreover, we assert that for each undeleted node v, $b[v]$ is undeleted. If $b[v]$ were a neighbor of x, then there would be a backward path from $b[v]$ back to x, and then up the tree to v; thus v would be adjacent to $b[v]$ by the Backward Path Theorem, contradicting the definition of $b[v]$. This argument proves the assertion. The claim follows by induction on the length of the elimination path. $\square$

To generalize this procedure to the case in which $T(G_\$)$ is a tree, we use an idea of Naor, Naor, and Schäffer: eliminating terminal branches. A *terminal branch* of a tree is a maximal path of degree-two nodes ending in a leaf. Naor, Naor, and Schäffer observe that deletion of all terminal branches of a tree yields a new tree with half as many leaves. Therefore, $O(\log n)$ iterations of terminal branch elimination suffice to eliminate the entire tree. They apply this idea to the *clique tree* of a chordal graph, a tree representing the structure of intersections among maximal cliques, in order to obtain a parallel algorithm for finding a maximal independent set. However, even assuming the clique tree is given, they prove only that the number of processors required is $O(n^2)$. By applying the idea to the elimination tree, we obtain a simpler algorithm requiring only $m/\log n$ processors.

Figure 11: To splice a node out of a tree, remove the node and reattach the node's children to its parent.

Before giving the algorithm, we introduce a bit of tree-surgery, called *splicing*. To *splice* a node v out of a tree T is to remove the node and reattach any children of v to v's parent in T, as illustrated in Figure 11. If a set of nodes are to be spliced from a tree, the resulting tree does not depend on the order in which the nodes are spliced out. In fact, they can all be spliced out at once; for each node v to be spliced out, the children of v are reattached to the lowest ancestor of v that is not spliced out.

We now describe the algorithm MIS, shown in Figure 12, for constructing a maximum independent set in a chordal graph G. The algorithm maintains a set $\mathcal{I}$, the independent set under construction, and a tree T, obtained from the elimination tree $T(G_\$)$ by splicing out nodes. We prove by induction that the following invariant holds before and after each iteration of the algorithm.

Invariant

(1) $\mathcal{I}$ is an independent set.

(2) Every neighbor of a node of $\mathcal{I}$ is in fact a *richer* neighbor of some node of $\mathcal{I}$.

(3) T is obtained from $T(G_\$)$ by splicing out the nodes of $\mathcal{I}$ and their neighbors.

The algorithm terminates when T is empty, at which point $\mathcal{I}$ is an independent set such that every node of G is either in $\mathcal{I}$ or is a richer neighbor of some node in $\mathcal{I}$. Thus, as in Gavril's algorithm, the family of cliques of the form $\{x\} \cup \{\text{richer neighbors of } x\}$ for $x \in \mathcal{I}$ is a minimum clique cover, and $\mathcal{I}$ is a maximum independent set.

Initially, $\mathcal{I} = \emptyset$ and $T = T(G_\$)$, so the invariant holds trivially. Suppose the invariant holds through the first k iterations of the algorithm, and consider the $k+1^{st}$ iteration. For each terminal branch $\mathcal{B}$, the algorithm finds a maximum independent set $\mathcal{I}_\mathcal{B}$ of the subgraph induced on B, using PMIS as a subroutine. If two nodes of T lie in different terminal branches, neither is an ancestor of the other in $T(G_\$)$, and hence the two nodes are not adjacent, by Lemma 3.1. Thus $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ is an independent set in G, where the union is over all terminal branches of T. Moreover, T contains no neighbors of $\mathcal{I}$ (by part (3) of the invariant), so $\mathcal{I} \cup (\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B})$ is an independent set of G. Thus when the nodes $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ are added to $\mathcal{I}$ in step M4, part (1) of the invariant remains true.

To show that part (2) remains true, we must prove that every node w that is newly a neighbor of a node in $\mathcal{I}$ is in fact a richer neighbor of a node in $\mathcal{I}$. Our simulation PMIS of Gavril's algorithm on terminal branches $\mathcal{B}$ guarantees this property when w lies on a terminal branch. Suppose therefore that w does not lie on a terminal branch, and let $v \in \mathcal{I}$ be a neighbor of w. By Lemma 3.1, v is either a descendent or an ancestor of w in $T(G_\$)$. The set $\mathcal{I}$ consists only of

MIS

M1 To initialize, let $\mathcal{I} = \emptyset$ and let $T = T(G_{\$})$.

M2 While T is not empty,

M3 Use the algorithm PMIS to find the greedy maximum independent set $\mathcal{I}_{\mathcal{B}}$ of the subgraph induced on each terminal branch $\mathcal{B}$ of T.

M4 Add the nodes $\bigcup_{\mathcal{B}} \mathcal{I}_{\mathcal{B}}$ to $\mathcal{I}$.

M5 Splice out of T the nodes $\bigcup_{\mathcal{B}}(\mathcal{I}_{\mathcal{B}} \cup \{\text{neighbors of } \mathcal{I}_{\mathcal{B}}\})$.

Figure 12: The algorithm MIS to construct a maximum independent set $\mathcal{I}$ in the chordal graph G.

nodes in terminal branches of T and descendents of such nodes. Hence v must be a descendent of w, and a poorer neighbor.

In the last step of an iteration of the algorithm, we splice out of T all nodes newly added to $\mathcal{I}$ and their neighbors. This step ensures that part (3) holds at the end of the iteration.

Having proved that the invariant continues to hold, we now consider the implementation of the algorithm MIS. In step M3, the algorithm must identify the nodes lying in terminal branches of T. An application of the Euler tree technique [44] suffices to determine, for each node v of T, the number of leaf descendents of v. The nodes for which this number is 1 are the nodes in terminal branches. Next, the algorithm must find a maximum independent set in each terminal branch. For each node v in T, $b_0[v]$ is assigned the lowest ancestor w of v in T that is not a neighbor of v, if w lies in a terminal branch. Otherwise, $b_0[v]$ is assigned v. As in PMIS, a pointer-jumping technique is then used to mark the nodes $x, b_0[x], b_0[b_0[x]]$, and so on, for all leaves x of T. The marked nodes are added to $\mathcal{I}$ in step M4.

To implement the splicing in step M5, we again use a pointer-jumping technique; for each node v, we compute the lowest ancestor of v in T that is not to be spliced out. Each step can be implemented in $O(\log n)$ time using $m/\log n$ processors and $O(m)$ space. Each iteration removes all nodes in terminal branches and hence reduces the number of leaves in T by a factor of two; consequently, $\lceil \log n \rceil + 1$ iterations suffice, for a total of $O(\log^2 n)$ time.

3.6 Optimal coloring

Gavril showed that applying the greedy coloring algorithm to the nodes of G in reverse order of \$ yields an optimal coloring. Our basic approach to coloring the graph in parallel is as follows: choose a clique K such that the components $H_1, \ldots, H_s$ of $G[G - K]$ are all "small," recursively color each subgraph $G[H_i \cup K]$, repair the colorings by making them consistent on the nodes of K, and merge the repaired colorings.

Naor, Naor, and Schäffer use essentially this approach in their coloring algorithm. Their algorithm, however, uses n^3 processors even if all maximal cliques are provided. One apparent difficulty is that the subgraphs on which the algorithm recurs are not disjoint—they share the nodes in K—so we cannot hope to make do with only one processor per edge.

COLOR(G, K_0):
Input: Connected graph G containing a clique K_0, such that every node of K_0 has a neighbor in $G - K_0$.
Output: Optimal coloring of G.

C1 If $G - K_0$ consists of a single node v, then G is a clique; assign the first $|V(G)|$ colors to its nodes, and end.

C2 Break $G - K_0$ into subgraphs $H_0, \ldots, H_s$ such that

- each subgraph has size at most half that of $G - K_0$;
- $H_1, \ldots, H_s$ are distinct components of $G - K_0 - H_0$; and
- for $1 \leq i \leq s$, the neighborhood of H_i in $G - H_i$ is a clique K_i.

C3 For $i = 0, \ldots, s$ in parallel,
call COLOR($\widehat{H}_i, K_i$), where $\widehat{H}_i = G[H_i \cup K_i]$, to get an optimal coloring c_i of $\widehat{H}_i$.

C4 For $i = 1, \ldots, s$ in parallel, modify the coloring c_i to be consistent with c_0 on the nodes $V(K_i)$ they have in common.

C5 Merge the colorings to obtain a coloring of G.

Figure 13: The recursive algorithm COLOR for finding an optimal coloring of a chordal graph G.

In coping with this difficulty, we use the same idea that made our peo algorithm efficient. Given the knowledge that K is a clique, we need not inspect the edges between nodes of K during the recursive calls. We recursively solve the following problem: given a chordal graph G and a clique K_0 contained in G, find an optimal coloring of G. We solve this problem in $O(\log t \log |G - K_0|)$ time using t processors, where t is the number of edges with at least one endpoint in $G - K_0$, or using $t/\log t$ processors of a randomized P-RAM. The algorithm, COLOR(G, K_0), is shown in Figure 13. There are $1 + \log |G - K_0|$ levels of recursion. We shall show that each level can be implemented in $O(\log t)$ time. To find a coloring in the original graph G, we call COLOR($G, \emptyset$).

Step C4, in which we modify the colorings to be consistent, can be implemented using a parallel prefix computation. We shall give more details later. The idea in implementing step C2, in which we divide up the graph, is as follows. We inductively assume we have an elimination tree $T(G_\$)$ in which the nodes of K_0 are the richest nodes. Using the Euler-tour technique [44], choose the lowest node $\hat{v}$ in $T(G_\$)$ that has more than $p/2$ descendents, where $p = |G - K_0|$. Let $v_1, \ldots, v_s$ be the children of $\hat{v}$ in $T(G_\$)$; then each subtree $T_{v_i}(G_\$)$ has at most $p/2$ nodes. We let K be the clique $\{\hat{v}\} \cup \{$richer neighbors of $\hat{v}\}$, and let $H_i = G[T_{v_i}(G_\$)]$. By Lemma 3.2, the subgraphs $H_1, \ldots, H_s$ are connected components of $G - K$. The neighborhood of each H_i in $G - H_i$ is contained in the clique K, and is therefore itself a clique K_i. Let $H_0 = G - K_0 - \bigcup_{i=1}^{s} H_i$. By choice of $\hat{v}$, H_0 has at most $p/2$ nodes.

We shall inductively ensure that for $i = 0, \ldots s$,

(I) c_i is an optimal coloring of $G[H_i \cup K_i]$;
(II) the maximum color used by c_i equals the number of colors used; and
(III) the coloring c_i assigns the first $|K_i|$ colors to the nodes of K_i.

These conditions are easy to establish at the base of the recursion, step C1. Condition (III) is automatically preserved in going from one level of recursion to the next higher level: the colors assigned by c to the nodes of K_0 are exactly those assigned by c_0. Assume (I), (II), and (III) hold for $c_0, \ldots, c_s$. We must ensure that (I) and (II) hold for the coloring c of G that we construct. Namely, we must color G with colors 1 through x, where x is the minimum number of colors needed to color G. We shall in fact construct a coloring c of G with maximum color equal to

$$\max\{\text{number of colors used by } c_i \ : \ i = 0, \ldots, s\}. \tag{1}$$

Since c_i is an optimal coloring of the subgraph $G[H_i \cup K_i]$ of G, the value given by (1) is clearly a lower bound on x. Thus by achieving this lower bound, we ensure that our coloring c of G is optimal.

The colors c assigns to nodes of $H_0 \cup K_0$ are exactly the colors c_0 assigns to these nodes. Therefore, for any node $v \in H_0 \cup K_0$, the color assigned to v is no more than the value of (1). It remains to determine the colors c assigns to nodes of H_i, for $i = 1, \ldots, s$.

Let x_i be the number of colors used by c_i. The colors 1 through $|K_i|$ are assigned by c_i to the nodes of K_i. The nodes of H_i are asigned colors $|K_i|+1$ through x_i. For each q, $|K_i|+1 \leq q \leq x_i$, let

$$A_i[q] := |\{c_0(v) < q : v \in V(K_i)\}|.$$

(The values $A_i[\cdot]$ can be computed using a parallel prefix computation.) For each node $v \in H_i$, define

$$c(v) := c_i(v) - |K_i| + A_i[c_i(v)]$$

Thus the colors of nodes of H_i are remapped to colors starting at 1, with gaps only for colors already assigned to nodes of K_i. This assignment ensures that, for any node $v \in H_i$, colors 1 through $c(v)$ all appear in the coloring c_i' induced by c on H_i. Since c_i' can be obtained from c_i by merely permuting colors, it follows that $c(v)$ is no more than the value of (1). We have completed the proof of correctness of the algorithm.

The only nontrivial computation in implementing step C4 is computing the $A_i[q]$ values. For each $1 \leq i \leq s$, we identify the colors c assigns to nodes of K_i, and then perform a parallel prefix computation of length $x_i \leq |H_i|$. The total work is proportional to $\sum_i |K_i| + |H_i|$.

Let t_i be the number of edges that either lie in H_i or connect H_i to K_i. Since H_i is connected, the number of nodes in H_i is at most one more than the number of edges in H_i. Since every node in K_i is a neighbor of some node in H_i, the number of nodes in K_i is at most the number of edges connecting H_i to K_i. Thus $|H_i \cup K_i| \leq t_i + 1$. so the total work is proportional to $\sum_i t_i$, which is just the number t of edges that either lie within $G - K_0$ or connect $G - K_0$ to G.

Assume inductively that $O(\log t_i \log |H_i|)$ time and $O(t_i / \log t_i)$ processors are sufficient to recursively color $G[H_i \cup K_i]$. Then $O(t/\log t)$ processors are sufficient to recursively color all the subgraphs $G[H_i \cup K_i]$ in $O(\log t(\log(|G - K_0|) - 1))$ time and to combine the colorings in $O(\log t)$ time, for a total of $O(\log t \log |G - K_0|)$ time.

3.7 PQ-tree intersection

We can also test two leaf-labelled PQ-trees for isomorphism. The idea is to use Edmonds' tree isomorphism algorithm (see [1]), which proceeds in stages from the leaves to the roots, level by level. In general, the number of levels may be large, so we instead apply Edmonds' algorithm to *decomposition trees* for the original PQ-trees. The decomposition tree of a tree T is formed by breaking T into subtrees of half the size by removing the edges from a node v to its children, recursively finding decomposition trees of the subtrees, and hanging the recursive decomposition trees from a common root. By labeling the decomposition tree during its construction, one can ensure that it uniquely represents T up to isomorphism. The decomposition trees for n-leaf PQ-trees have $O(\log n)$ levels, so $O(\log n)$ stages of Edmonds' algorithm suffice. Each stage involves sorting strings, which can be done in $O(\log n)$ time on a "priority" type CRCW P-RAM using Cole's algorithm [9], for a total time bound of $O(\log^2 n)$. To achieve this time bound, $(n+t)/\log n$ processors are sufficient, where t is the sum of the lengths of the leaf-labels.

3.8 Interval graphs

The algorithm of Booth and Lueker for recognition of interval graphs is as follows: Find a peo of the input graph G; if there is none, the graph is certainly not an interval graph. Otherwise, we can obtain all the maximal cliques (there are at most n; see Subsection 3.3). For each node v, let A_v be the set of maximal cliques containing v. It can be shown [17] that $\sum_v |A_v| = O(m+n)$. Let T be the universal PQ-tree whose ground set is the set of maximal cliques. Reduce T with respect to the sets A_v. If the resulting PQ-tree T_G is the null tree, then it follows from a theorem of Gilmore and Hoffman [22] that G is not an interval graph. Otherwise, an ordering represented by T yields a representation of G as an intersection of intervals.

Since we have given efficient parallel algorithms for finding a peo and for PQ-tree multiple (non-disjoint) reduction, the above algorithm is parallelizable; each step takes $O(\log^2 n)$ time using $O(n+m)$ processors.

Booth and Lueker showed ([32]; also see [8]) that if the PQ-tree T_G derived from G is augmented with some labels depending on the graph, isomorphic interval graphs correspond to isomorphic labelled PQ-trees. Booth and Lueker then showed that such labelled PQ-trees could be tested for isomorphism in linear time, proving that interval graph isomorphism testing could be done in linear time. We show that this approach can be parallelized.

Let T_G be the PQ-tree for an interval graph G. We augment T_G with labels as follows: Each leaf x of T_G corresponds to a maximal clique $\mathcal{C}_x$; we create a label for x by sorting the degrees of the nodes in $\mathcal{C}_x$. The sum of the lengths of the strings labeling T_G is just the sum of the sizes of the maximal cliques, which is $O(n+m)$. The labels can be found in $O(\log n)$ time using $O(n+m)$ processors, by means of small-integer sorting. It follows from Theorem 1 of [32] and the proof of Lemma 3.1 of [8] that the resulting labelled PQ-tree uniquely represents the interval graph G up to isomorphism.

The number of nodes in the augmented PQ-tree is $O(m+n)$, and the augmentation can easily be carried out in parallel. It remains to show how such augmented PQ-trees may be tested for isomorphism in $O(\log^2 n)$ time using $m+n$ processors. To achieve this time bound, we require a powerful model of parallel computation, the "priority" CRCW P-RAM. Multiple processors are permitted to write to the same location in the same time step; the value stored in the location is the value written by the lowest-numbered processor.

The standard (sequential) approach to tree isomorphism (see [1], [26]) and the approach used in [32] is to process the two trees from the leaves up, essentially canonicalizing subtrees at each successive level by sorting. The problem with a direct parallelization of this approach is that there may be too many levels. Therefore, our approach is to test isomorphism not of the original trees but of their decomposition trees, which are guaranteed to have only a logarithmic number of levels.

In constructing a decomposition tree for an augmented PQ-tree, we must for the recursion construct decomposition trees for slightly more general trees: trees that are obtained from augmented PQ-trees by deleting some subtrees and assigning numbers to some (resulting) leaves. Let T be such a modified PQ-tree. Note that, for example, T may contain P-nodes and Q-nodes having no children. The leaves of the decomposition tree for T will correspond to the nodes of T.

We form T's decomposition tree $\mathcal{D}_j(T)$ as follows: if T consists of a single node x, then $\mathcal{D}_j(T)$ also consists of a single vertex v. The vertex v is labelled with P,Q, L, or D, depending on whether x is a P-node, a Q-node, a leaf, or a degree node in the original augmented PQ-tree. Moreover, if x was numbered, then v is labelled with the same number.

Suppose T contains at least two nodes. There is a unique node $\hat{x}$ in T that is the lowest node of T having more than $n/2$ descendents (including itself). The choice of $\hat{x}$ is invariant under automorphisms of T. Moreover, $\hat{x}$ is not a leaf. Removal of the edges joining $\hat{x}$ to its children disconnects T into subtrees each having at most $n/2$ nodes. Let these subtrees be $T_0, \ldots, T_k$, where T_0 is the subtree containing $\hat{x}$, and $T_1, \ldots, T_k$ are ordered just as their roots are ordered as children of $\hat{x}$ in T. Modify T_0 by assigning the number of nodes of T to the leaf $\hat{x}$. Then $\mathcal{D}(T)$ is defined to be the tree obtained from $\mathcal{D}(T_0), \ldots \mathcal{D}(T_k)$ by introducing a new vertex v to be the parent of the roots $v_0, \ldots, v_k$ of these $k+1$ decomposition trees, in this order. The vertex v is labelled with P,Q, L, or D as before.

The assignment of integers to leaves of the modified PQ-trees and to leaves of the decomposition trees makes it possible to uniquely reconstruct a PQ-tree T from its decomposition tree $\mathcal{D}(T)$, while at the same time establishing the correspondence between the nodes of T and the leaves of $\mathcal{D}(T)$. Let v be the root of $\mathcal{D}(T)$. Recursively reconstruct the modified PQ-trees whose decomposition trees are rooted at the children of v in $\mathcal{D}(T)$. Let $T_0, \ldots, T_k$ be the resulting PQ-trees in order. There is a unique leaf $\hat{x}$ in T_0 labelled with the integer $\sum_i = 0^k |T_i|$. To construct T, let the roots of $T_1, \ldots, T_k$ be the children of $\hat{x}$ in order, and remove the label from $\hat{x}$. It follows that two decomposition trees are isomorphic if and only if their decomposition trees are isomorphic.

It remains to describe how to test decomposition trees for isomorphism. The approach used is essentially the standard approach, but we must take care to respect the P,Q,L,D labels, the integer labels assigned during augmentation, the integer labels assigned during decomposition, and the order of children of Q-nodes. It follows from the construction of the decomposition trees that only leaves have integer labels.

The isomorphism algorithm proceeds top-to-bottom, from the leaves of the decomposition trees to their roots. The *height* of a vertex in a tree is defined to be the maximum distance down the tree to a leaf. We initialize by assigning a string to each leaf (height-0 vertex) of each tree. The string combines all the labels of the leaf.

For the general step of the algorithm, we are given an assignment of strings to the height-h vertices of the trees. We sort these strings, eliminate duplicates, and assign to each string the ordinal number of the string in the set of strings. If the multiset of strings associated with height-h vertices of one tree does not match the corresponding multiset of the other tree, we terminate the algorithm because the trees are not isomorphic. We also terminate the algorithm if one tree has height-$(h+1)$ vertices and the other does not.

Otherwise, if neither tree has height-$(h+1)$ vertices, we conclude that the trees are isomorphic, and if both do, we next assign strings to these vertices as follows: let v be a height-$(h+1)$ vertex. It follows from the construction of the decomposition tree that v corresponds to an internal node of the original augmented PQ-tree, and is therefore labelled with P,Q, or L. If v is labelled with

P or L, we form its string as follows: the first element of the string is either P or L, whichever is appropriate. The second element of the string is the integer assigned to v first child. The remaining elements are obtained by sorting the collection of integers assigned to its remaining children. If v is labelled with Q, we proceed somewhat differently. As before, the first element of v's string is the label Q, and the second element is the integer assigned to v's first child. To determine the remainder of the string, we consider two sequences, one consisting of the integers assigned to the remaining children of v in order, the second being the reverse of the first. The remainder of v's string is whichever of these two sequences is lexicographically less.

It is a simple induction to see that two height-h vertices receive the same string or same integer iff the subtrees rooted at these vertices are isomorphic. This shows that the isomorphism algorithm is correct.

There are $\leq 1 + \log t$ levels in the decomposition trees for t-node trees. Therefore, the above algorithm has $O(\log t)$ stages, where $t = O(n+m)$. In each stage, the most difficult step is sorting strings. The sum of the lengths of the strings is $O(t)$ in each stage. Therefore, we can assign a processor to each symbol of each string. To compare two strings, processors associated with corresponding symbols communicate to compare their symbols. Using the powerful concurrent-write capability, the processors associated with the two strings can determine in constant time the minimum index at which the strings differ, and hence which string comes first in lexicographic order. Using a logarithmic-time comparison sort [9], the strings can then be sorted in $O(\log t)$ time using $O(t)$ processors. Hence t-node tree isomorphism can be tested in $O(\log^2 t)$ time using t processors.

Acknowledgements

Many thanks to David Shmoys, who advised the thesis based in part on this research. Thanks also to others with whom I discussed this research, including Dina Kravets, Tom Leighton, Charles Leiserson, George Lueker, Mark Novick, James Park, John Reif, Cliff Stein, and Joel Wein.

References

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts (1974).

[2] A. Aggarwal and R. Anderson, "A random NC algorithm for depth first search," *Combinatorica 8* (1988), pp. 1-12.

[3] R. J. Anderson and G. L. Miller, "A simple randomized parallel algorithm for list-ranking," *Inf. Proc. Let. 33* (1990), pp. 33.

[4] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, "On the desirability of acyclic database schemes," *J. ACM 30* (1983), pp. 479-513.

[5] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam (1973).

[6] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comp. Sys. Sci. 13* (1976), pp. 335-379.

[7] N. Chandrasekharan and S. S. Iyengar, "NC algorithms for recognizing chordal graphs and k-trees," Tech. Rept. #86-020, Dept. of Comp. Sci., Louisiana State Univ. (1986).

[8] C. J. Colbourn, K. S. Booth, "Linear time automorphism algorithms for trees, interval graphs, and planar graphs," *SIAM J. Comp. 10* (1981), pp. 203-225.

[9] R. Cole, "Parallel merge sort," *SIAM J. Comp. 17* (1988), pp. 770-785.

[10] R. Cole and U. Vishkin, "Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time," *SIAM J. Comp. 17* (1988), p. 128-142.

[11] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *SIAM J. Comp. 11* (1982), pp. 472-492.

[12] E. Dahlhaus and M. Karpinski, "The matching problem for strongly chordal graphs is in NC," Tech. Rept. 855-CS, Institut für Informatik, Universität Bonn, 1986.

[13] E. Dahlhaus and M. Karpinski, "Fast parallel computation of perfect and strongly perfect elimination schemes," Tech. Rept. 8519-CS, Institut für Informatik, Universität Bonn, 1987, and IBM Research Report RJ5901, October 1987.

[14] G. A. Dirac, "On rigid circuit graphs," *Abh. Math. Sem. Univ. Hamburg 25* (1961), pp. 71-76.

[15] A. Edenbrandt, *Combinatorial Problems in Matrix Computation*, TR-85-695 (Ph.D. Thesis), Department of Computer Science, Cornell University (1985).

[16] A. Edenbrandt, "Chordal graph recognition is in NC," *Inf. Proc. Let. 24* (1987), pp. 239-241.

[17] D. Fulkerson and O. Gross, "Incidence matrices and interval graphs," *Pac. J. Math 15* (1965), pp. 835-855.

[18] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM J. Comp. 1* (1972), pp. 180-187.

[19] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *SIAM J. Comp.*.

[20] H. Gazit, G. L. Miller, "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Inf. Proc. Let. 28* (1988), pp. 61-65.

[21] J. R. Gilbert, D. J. Rose, and A. Edenbrandt, "A separator theorem for chordal graphs," *SIAM J. Alg. Disc. Meth. 5* (1984), pp. 306-313.

[22] P. C. Gilmore and A. J. Hoffman, "A characterization of comparability graphs and of interval graphs," *Can. J. Math 16* (1964), pp. 539-548.

[23] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York (1980).

[24] C.-W. Ho and R. C. T. Lee, "Efficient parallel algorithms for finding maximal cliques, clique trees, and minimum coloring on chordal graphs," *Inf. Proc. Lett. 28* (1988), pp. 301-309.

[25] C.-W. Ho and R. C. T. Lee, "Counting clique trees and computing perfect elimination schemes in parallel," *Inf. Proc. Lett. 31* (1989),pp. 61-68.

[26] J. Hopcroft and J. Wong, "Linear time algorithm for isomorphism of planar graphs," *Sixth Annual ACM Symp. on Theory of Computing* (1974), pp.172-184.

[27] P. N. Klein, *Efficient parallel algorithms for planar, chordal, and interval graphs*, TR-426 (Ph.D. thesis), Laboratory for Computer Science, MIT (1988).

[28] P. N. Klein, "Efficient parallel algorithms for chordal graphs," *Proc. 29th Symp. Found. of Comp. Sci.* (1989), pp. 150-161.

[29] P. N. Klein and J. H. Reif, "An efficient parallel algorithm for planarity," *J. Comp. Sys. Sci.37* (1988), pp. 190-246.

[30] D. Kozen, U. Vazirani, and V. Vazirani, "NC algorithms for comparability graphs,and testing for unique perfect matching," *Proc. 5th Symp. Found. of Software Technology and Theor. Comp. Sci.*, published as *Lecture Notes in Computer Science 206*, Springer-Verlag, New York (1985), pp. 496-503.

[31] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM 27:4* (1980), pp. 831-838.

[32] G. S. Lueker and K. S. Booth, "A linear time algorithm for deciding interval graph isomorphism," *J. ACM 26:2* (1979), pp. 183-195.

[33] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," *26th FOCS* (1985), pp. 478-489.

[34] J. Naor, M. Naor, and A. A. Schäffer, "Fast parallel algorithms for chordal graphs," *SIAM J. Comput. 18* (1989), pp. 327-349.

[35] M. B. Novick, personal communication (1987).

[36] M. B. Novick, personal communication (1988).

[37] M. B. Novick, 'Logarithmic time parallel algorithms for recognizing comparability and interval graphs," manuscript.

[38] J. H. Reif and S. Rajasekaran, "Optimal and sublogarithmic time randomized parallel sorting algorithms," *SIAM J. Comput. 18* (1989), pp. 594-607.

[39] D. J. Rose, "Triangulated graphs and the elimination process," *J. Math Anal. Appl. 32* (1970), pp. 597-609.

[40] D. J. Rose, R. E. Tarjan, and G. S. Lueker, "Algorithmic aspects of vertex elimination on graphs," *SIAM J. Comp. 5* (1976), pp. 266-283.

[41] J. E. Savage and M. G. Wloka, "A parallel algorithm for channel routing," *Proceedings of WG '88, Graph-theoretic Concepts in Computer Science*, published as *Lecture Notes in Computer Science*, Springer-Verlag, New York (1988).

[42] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. on Mathematical Software 8:3* (1982), pp. 256-276.

[43] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms 3* (1982), pp. 57-67.

[44] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *SIAM J. comput. 14* (195), pp. 862-874.