Efficient parallel algorithms for chordal graphs

Philip N. Klein*
Harvard 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.

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 [31] and in databases [3], and have been the object of much algorithmic study since the work of Fulkerson and Gross in 1965 [15].

Chordal graphs are an important subclass of the class of *perfect graphs* [4], 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 graph can be recognized in linear time.

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 [17], [26],[30], 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 previous parallel algorithms requiring, for several of these problems, n^4 processors to achieve the same time bound. The chordal graph problems we solve are:

1. recognizing chordal graphs (previous processor bound: n^2m, due to [27]),
2. finding a maximum-weight clique in a chordal graph (previous processor bound: n^4, due to [27]),
3. finding a maximum independent set (and a minimum clique-cover) in a chordal graph (previous processor bound: n^4, due to [27]),
4. finding an optimal coloring of a chordal graph (previous processor bound: n^4, due to [27]),
5. finding a breadth-first search tree of a chordal graph (processor bound for general graphs: $M(n)$, due to [18])[1],
6. finding a depth-first search tree of a chordal graph (processor bound for general graphs, using randomization: $nM(n)$, due to [2]).

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 (or using $(n + m)/logn$ processors, if randomization is permitted), 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. Recognition of interval graphs was previously shown to be in NC by Kozen, Vazirani, and Vazirani [23], but no specific time or processor bound was given. To our knowledge, no previous NC algorithm was known for interval graph isomorphism,

In related work, Novick [29] has claimed an $O(\log n)$ time, n^3 processor CRCW algorithm for recognizing interval graphs and constructing a PQ-tree representing a given interval graph. He observed [28] 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 [33] have given an efficient parallel algorithm for optimum coloring of interval graphs. Their

*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.

[1] Here $M(n)$ denotes the time required to multiply two $n \times n$ matrices. The best bound known [9] is $O(n^{2.376})$.

0272-5428/88/0000/0150$01.00 © 1988 IEEE

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.

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 [15] discovered the peo and used it to find all the maximal cliques of a chordal graph. Rose ([31]) 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 [16] 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 for finding a peo was left open by Edenbrandt [13],[14], and by Chandrasekharan and Iyengar [6], and resolved by Naor, Naor, and Schäffer ([27]) and independently by Dahlhaus and Karpinski ([11]). However, the peo algorithms of [27] and [11] required $O(n^4)$ processors. In fact, it is suggested in [27] 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. In this paper, we refute that suggestion.

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 an integer labelling of the nodes of a graph (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 2.2.

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 a variant of Miller and Reif's parallel tree contraction [26] to the elimination tree. We believe the elimination tree may also prove useful in other parallel chordal graph algorithms.

We provide only a sketch of the more difficult applications, optimal coloring and maximum independent set for chordal graphs, and interval graph recognition and isomorphism. For more details, the reader is referred to the author's Ph.D. thesis [21].

Graph Notation: Let G be a 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 [15] in 1965. Dirac had proved in [12] 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 integers). 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 $\pounds$ be two numberings of the nodes of G. We say \$ is *consistent with* $\pounds$ (and $\pounds$ is a *refinement of* \$) if $\$(v) < \$(w)$ implies $\pounds(v) < \pounds(w)$ for all nodes v and w. We say $\pounds$ is a refinement of $G_{\$}$ if we wish to emphasize the graph for which $\pounds$ is a numbering. Note that each class-component of $G_{\pounds}$ is a subgraph of some class-component of $G_{\$}$.

We call $\not\in$ a *partial numbering* if $\not\in$ assigns integers 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 numbering $\not\in$, the *refinement of \$ by* $\not\in$ is defined to be the numbering $\$\not\in$ in which $\not\in$ is used to break ties in \$. That is, $\$\not\in(v) < \$\not\in(w)$ if

- either $\$(v) < \$(w)$,
- or $\$(v) = \$(w)$, $\not\in(v)$ and $\not\in(w)$ are defined, and $\not\in(v) < \not\in(w)$.

Typically, $\not\in$ 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 $\$\not\in$ from \$ as *stratifying* the class-component C of $G_{\$}$, or as *well-stratifying* C if in addition each class-component of $G_{\$\not\in}$ 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.

[2]The notion is a generalization of one appearing in Lemma 4 of [32].

ITERATED REFINEMENT

P1 To initialize, let \$ be the trivial numbering assigning 0 to every node of G, and let $\delta = 2^{3\lceil \log_{5/4} n \rceil}$.
P2 While \$ is not one-to-one,
P3 For each nonsingleton class-component C of $G_{\$}$ in parallel:
P4 call STRATIFY($G_{\$}, C, \delta$).
P5 Let $\delta := \delta/8$.

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

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($G\$, C, \delta$). 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($G_{\$}, C, \delta$) 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 P4 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 algorithm of Gazit [17], we can reduce the processor bound by a factor of $\log n$ without increasing the time bound.

The parameter δ is used to refine the numbering. The algorithm maintains the invariant that immediately before step P4, the number $\$(C)$ assigned to a

nonsingleton class C of $G_\$$ is at least 2δ less than the number assigned to the next richer class. In stratifying a class-component C, the procedure STRATIFY adds the value δ to the numbers of some of the nodes of C.

As an aside, we note that the initial numbering can be any valid numbering \$ of G; 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 is proved by 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. □

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 $\pounds$ is a refinement of \$, a path P that is weakly backward in $G_\pounds$ is also weakly backward in $G_\$$. If P is strictly backward in $G_\pounds$ but only weakly 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: If there is a subpath of P that is strictly backward, we replace the subpath by the edge connecting its endpoints. Continuing in this way, we obtain a uniform path P'. □

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 ϕ be a numbering of a class-component C of $G_\$$. Then the refinement of $G_\$$ by ϕ is valid if and only if the refinement of $\widehat{C}_\$$ by ϕ is valid.

Proof: Let $\$\phi$ be the refinement of $G_\$$ by ϕ, and let λ be the refinement of $\widehat{C}_\$$ by ϕ. Assume that \$ is a valid numbering of G. If $\$\phi$ 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 $G_{\$\phi}$. Conversely, suppose that λ is a valid numbering of $\widehat{C}$. For each backward path P in $G_{\$\phi}$, 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 $\$\phi$. 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 $\$\phi$, it follows that the internal nodes have a lower ϕ-number than x. Hence the internal nodes and x must all lie in C, because ϕ 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 ϕ-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. □

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 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 ϕ_i be a numbering of C^i such that the refinement of $\widehat{C^i}_\$$ by ϕ_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 ϕ_i. The numbering that $\$_{i-1}$ induces on $\widehat{C^i}$ is identical to that induced by \$, so refining $\widehat{C^i}_{\$_{i-1}}$ by ϕ_i is valid. It then fol-

lows via the Refinement Lemma that $\$_i$ is valid. We conclude that $\$_k$ is a valid refinement of \$. We can obtain $\$_k$ directly by using $\not\in_i$ to stratify C^i, for all class-components C^i in parallel. This is how steps P3 and P4 of ITERATED REFINEMENT are carried out. The algorithm STRATIFY$(G_\$, C^i, \delta)$ 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 [12].

Lemma 1.5 [Dirac] If S is a minimal set of nodes whose removal separates a 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 $\not\in$ be a valid numbering of C. Then the refinement of $G_\$$ by $\not\in$ is valid.

Proof: By the Refinement Lemma, we need only show that the refinement λ of $\widehat{C}_\$$ by $\not\in$ 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 $\not\in$. 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. □

The procedure STRATIFY$(G_\$, C, \delta)$ appears in Figure 2. If C is a nonsingleton class-component of $G_\$$, the procedure increases the numbers of some of the nodes of C by at most δ, 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 every checking for their presence. Specifically, in computing connected components of a graph

Procedure STRATIFY($G_\$, C, \delta$):
(Wealth of next richer class exceeds $\$(C)$ by $\geq 2\delta$.)

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 NONE($G_\$, C, \delta$), and end.

S4 If every node in B has at least $\frac{2}{5}|C|$ neighbors in C, call HIGHDEGREE($G_\$, C, B, \delta$), and end.

S5 If there are nodes in B with fewer than $\frac{2}{5}|C|$ neighbors in C, call LOWDEGREE($G_\$, C, B, \delta$).

Figure 2: The main procedure for finding a valid well-stratification.

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

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 δ.

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 i 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 i to v_i's number.

Figure 3: The subprocedure used to well-stratify C if C has no richer neighbors.

involving nodes of B, the procedure uses the algorithm of Shiloach and Vishkin [35], suitably modified to take into account the fact that any two nodes of B are adjacent.

Depending on the nodes in B, the procedure STRATIFY($G_\$, C, \delta$) 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 [24]. We now show that STRATIFY($G_\$, C, \delta$) 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 NONE($G_\$, C, \delta$) shown in Figure 3 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 HIGHDEGREE($G_\$, C, B, \delta$) of Figure 4 is called. The procedure arbitrarily orders the blue nodes: $B = \{v_1, \ldots, v_k\}$. For $1 \leq j \leq k$, 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 δ. 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

Procedure HIGHDEGREE($G_\$, C, B, \delta$)
(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 δ.

H5 Let C' be the largest component of $G[F_{\hat{\jmath}}]$.

H6 If $\hat{\jmath} = k$, then call NONE($G_\$, C', \delta/2$).

Figure 4: The procedure HIGHDEGREE used to stratify C in case every richer neighbor of C has high degree in C.

$v_1, \ldots, v_j$ 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 NONE($G_\$, C', \delta/2$), 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.

Case III: Neither Case I nor Case II holds. In this case, the procedure LOWDEGREE($G_\$, C, B$) in Figure 5 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 $1 \leq j \leq k$, 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.

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.

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

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

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 δ.

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

L9 If $|C'| > \frac{4}{5}|C|$, then call STRATIFY($G_\$, C', \delta/2$).

Figure 5: The procedure LOWDEGREE used to stratify C in case some richer neighbor of C has low degree in C.

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 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 validity of the resulting numbering follows from the Refinement Lemma.

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 [35], suitably modified to take into account the fact that B is a clique.

2 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 [34] in the context of sparse Gaussian elimination. In Subsection 2.1, 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.

2.1 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 numbering 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 is a numbering of the nodes of a graph so that for each node v, the richer neighbors of v form a clique. Define a *cross-edge* to be an edge of G whose endpoints are not ancestors of each other in $T(G_\$)$. If there are no cross-edges, we call $T(G_\$)$ a *depth-first search tree*. The existence of a cross-edge between nodes u and 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 2.1 Let G be a chordal graph. If \$ is a peo of G, then $T(G_\$)$ has no cross-edges.

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 6, removing v and its richer neighbors separates the subtrees rooted at children of v from the remainder of the graph.

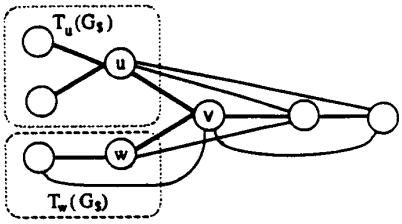

Figure 6: When the node v and its richer neighbors are removed, the subtrees rooted at children of v become separated from each other and from the remainder of the graph.

Lemma 2.2 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. □

2.2 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 [32]. 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. It is easy to prove that the numbering \$ is a peo if and only if no verification step fails. 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.

2.3 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 clique may easily be determined from \$.

2.4 Depth-first and breadth-first search trees

We observed in Subsection 2.1 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) to be the richest neighbor of v. Let T be the resulting tree. We shall show that T is a breadth-first search tree. Let the peo be $v_1 \ldots v_n$. For a node v_i, let $d(v_i)$ denote the depth of v_i in T, and let $p(v_i)$ denote the parent of v_i in T.

We first observe that $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$ because the parent of each node is a richer node. To prove that T is a breadth-first search tree, we must verify that, for each node v_i and each neighbor w of v_i, $d(v_i) \leq 1 + d(w)$.

By choice of v_i's parent $p(v_i)$, w is no richer than $p(v_i)$, so $d(w) \leq d(p(v_i))$. By definition of depth, $d(v_i) = 1 + d(p(v_i))$, which proves $d(v_i) \leq 1 + d(w)$.

2.5 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 $C_1, \ldots, C_s$ of $G - K$ are all "small," recursively color each subgraph $G[C_i \cup K]$, 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. The difficulty is that the subgraphs on which the algorithm recurses are not disjoint—they share the nodes in K—so we cannot hope to make do with only one processor per edge. Their algorithm, in fact, requires n^3 processors even if all maximal cliques are provided.

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.

COLOR(G, K_0):
Input: Graph G consisting of clique K_0 and subgraph H, such that every node of K_0 has a neighbor in H.
Output: Optimal coloring of G.

C1 If H 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 7: The recursive algorithm COLOR for finding an optimal coloring of a chordal graph G.

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 $O(\log^2 t)$ 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 7. 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 parallel prefix computation and small-integer sorting. 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 [36], choose the lowest node $\hat{v}$ in $T(G_\$)$ that has $> p/2$ descendents, where $p = |G - K_0|$. Let $v_1, \ldots, v_s$ be the children of v in $T(G_\$)$; then each subtree $T_{v_i}(G_\$)$ has $\leq p/2$ nodes. We let K be the clique $\{\hat{v}\} \cup \{\text{richer neighbors of } \hat{v}\}$, and, for $i = 1, \ldots, k$, we let $H_i = G[T_{v_i}(G_\$)]$. By Lemma 2.2, 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 $\leq p/2$ nodes.

There are $\log n$ levels of recursion, and each level

can be implemented in $O(\log n)$ time, for a total of $O(\log^2 n)$ time. The number of processors required is $O(m)$, or $O(m/\log m)$ using Reif's randomized small-integer sorting algorithm [30].

2.6 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 the form $\{v\} \cup \{\text{richer neighbors of } v\}$ for $v \in \mathcal{I}$ 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 tree $T(G_\$)$ were a path with leaf x. For each node v, let $b_0[v]$ be the lowest ancestor of v in $T(G_\$)$ that is not adjacent to v in G (or $\perp$, if no such ancestor exists). We claim that the greedy independent set consists of $x, b_0[x], b_0[b_0[x]]$, and so on. For suppose we put x into $\mathcal{I}$ and delete the neighbors of x. The node $b_0[x]$ is by definition the poorest undeleted node. Moreover, we assert that for each undeleted node v, $b_0[v]$ is undeleted. If $b_0[v]$ were a neighbor of x, then there would be a backward path from $b_0[v]$ back to x, and then up the tree to v; thus v would be adjacent to $b_0[v]$ by the Backward Path Theorem, contradicting the definition of $b_0[v]$. This argument proves the assertion; it follows by induction that the greedy independent set consists of $x, b_0[x], b_0[b_0[x]]$, and so on. This set can be determined quickly in parallel using standard pointer-jumping techniques as follows. Phase A: For stages $i = 0, \ldots, \lceil \log n \rceil - 1$, we let $b_{i+1}[v] := b_i[b_i[v]]$ for each node v. Phase B: mark x as being in $\mathcal{I}$, and, for stages $i = \lceil \log n \rceil - 1, \lceil \log n \rceil - 2, \ldots, 0$, for each marked node v we mark $b_i[v]$. Thus we identify $\mathcal{I}$ in $O(\log n)$ time.

To generalize this procedure to the case in which $T(G_\$)$ is a tree, we use the ideas of parallel tree contraction, a technique due to Miller and Reif [26]. The pointer-jumping steps are replaced with some more complicated operations, and are alternated with "raking" steps that remove leaves. An analysis similar to that of [26] shows that $O(\log n)$ stages are sufficient. Each stage can be carried out in $O(\log n)$ time, for a total of $O(\log^2 n)$ time, using $(n+m)/\log n$ processors.

2.7 PQ-trees

The results in this subsection are not an application of the peo algorithm, but they make possible the applications discussed in the next subsection.

A PQ-tree is a data structure developed by Booth and Lueker [5] for representing large sets of orderings of a ground set S. For this section, let $n = |S|$. A PQ-tree T for S has the elements of S as its leaves, and every internal node has at least two children. Hence T has at most $2n - 1$ nodes. The *universal* PQ-tree for S represents the set of all orderings of S; the *null* PQ-tree represents the empty set. Given a PQ-tree T and a subset A of S, the operation REDUCE(A) transforms T into a PQ-tree T' representing exactly those orderings σ such that σ is represented by T and in addition the elements of A occur consecutively in σ. Each PQ-tree for S can be obtained from the universal PQ-tree by a sequence of calls to REDUCE. Moreover, given any non-null PQ-tree, it is easy to read off one of the orderings represented.

Booth and Lueker gave an implementation of REDUCE(A) that ran in time $O(|A|)$. Klein and Reif [22] defined the operation MREDUCE$(\{A_1, \ldots, A_k\})$ that reduced T with respect to all the sets A_i simultaneously. They gave a parallel implementation of MREDUCE that ran in $O(\log^2 n)$ time using n processors, *provided* the A_i's were disjoint.

However, [22] left unresolved the problem of efficiently computing MREDUCE$(T, \{A_1, \ldots, A_k\})$ for overlapping sets A_i. In this abstract, we hint at how to handle this case; details are provided in [21]. In particular, let $t = \sum_i |A_i|$; we can carry out non-disjoint MREDUCE in $O(\log n \cdot (\log n + \log t))$ time using $n + t$ processors. We make use of three observations, the first of which appeared in [5]. Suppose we reduce T with respect to a subset $\mathcal{E}$ of the ground set S. Then in the resulting tree, there is (essentially) a sub-PQ-tree, which we denote by $T|\mathcal{E}$, whose ground set is $\mathcal{A}$. When this subtree is removed from T, the remaining sub-PQ-tree, denoted $T/\mathcal{E}$, has ground set $S - \mathcal{E}$.

Our first new observation is as follows: Assume T is reduced with respect to $\mathcal{E}$. For sets $A_1, \ldots, A_k$, let $A_i|\mathcal{E} = A_i \cap \mathcal{E}$, and let $A_i/\mathcal{E} = A_i - \mathcal{E}$. Then reducing T with respect to $A_1, \ldots, A_k$ is equivalent to (1) reducing a slightly modified version of $T|\mathcal{E}$ with respect to $A_1|\mathcal{E}, \ldots, A_k|\mathcal{E}$, (2) reducing a slightly modified version of $T/\mathcal{E}$ with respect to $A_1/\mathcal{E}, \ldots, A_k/\mathcal{E}$, and then (3) piecing the resulting trees together.

Our second new observation is as follows: Suppose the sets $A_1, \ldots, A_\ell$ are consecutive in the ordering σ. Then the intersection $\bigcap_{i=1}^{\ell} A_i$ is consecutive in σ. Moreover, if the intersection graph of $A_1, \ldots, A_\ell$ is connected, then the union $\bigcup_{i=1}^{\ell} A_i$ is also consecu-

tive in σ.

The above observations are used in our algorithm for non-disjoint MREDUCE. The idea is to recurse on the size n of the ground set. Given $A_1, \ldots, A_k$, we find a suitable union or intersection $\mathcal{E}$ of some of these sets such that $|\mathcal{E}|$ is a constant fraction of n. In accordance with the second new observation, we reduce T with respect to the set $\mathcal{E}$. Then we use the first new observation; we construct the modified versions of $T|\mathcal{E}$ and $T/\mathcal{E}$ and recurse on them, and then piece the results together. By a suitable choice of $\mathcal{E}$, the recursion depth is $O(\log n)$. A stage can be carried out in $O(\log n + \log \sum_i |A_i|)$ time using $O(n + \sum_i |A_i|)$ processors. A logarithmic-factor reduction in the processor bound can be achieved using randomization, through use of Reif's small-integer sorting algorithm [30].

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 labelling 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 [8], 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.

2.8 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 Section 2.3). For each node v, let A_v be the set of maximal cliques containing v. It can be shown [15] 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 [19] 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, or $O((n + m)/\log n)$ processors using randomization.

Booth and Lueker showed ([25]; also see [7]) that if the PQ-tree T_G derived from G is augmented with some labels depending on the graph, isomorphic interval graphs lead to isomorphic labelled PQ-trees and vice versa. 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 labelling 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 [25] and the proof of Lemma 3.1 of [7] that the resulting labelled PQ-tree uniquely represents the interval graph G up to isomorphism. We can use our parallel PQ-tree isomorphism algorithm (sketched in Subsection 2.7 to compare two such labelled PQ-trees in $O(\log^2 n)$ time using a "priority" CRCW P-RAM; the number of processors is $O((n + m)/\log n)$. Hence we can test isomorphism of interval graphs in $O(\log^2 n)$ time using $O(n + m)$ processors of a "priority" CRCW P-RAM, or using $O((n + m)/\log n)$ processors if randominess is permitted.

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," *19th STOC* (1987), pp. 325-334.

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

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

[5] 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. 13:3* (1976), pp. 335-379.

[6] 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).

[7] C. J. Colbourn, K. S. Booth, "Linear time automorphism algorithms for trees, interval graphs, and planar graphs," *SIAM J. Comp. 10:1* (1981), pp. 203-225.

[8] R. Cole, "Parallel merge sort," *27th FOCS* (1986), pp. 511-516.

[9] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *19th STOC* (1987), pp. 1-6.

[10] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *27th FOCS* (1986), pp. 492-501

[18] H. Gazit, G. L. Miller, "An improved parallel algorithm for BFS of a directed graph," manuscript, USC (1987).

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

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

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

[22] P. N. Klein and J. H. Reif, "An efficient parallel algorithm for planarity," *J. Comp. Sys. Sci.*, to appear. A preliminary version appeared in 27th FOCS (1986), pp. 465-477.

[23] 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.

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

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

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

[27] J. Naor, M. Naor, and A. A. Schäffer, "Fast parallel algorithms for chordal graphs," *19th STOC* (1987), pp. 355-364; also submitted to SIAM J. Comput.

[28] M. Novick, personal communication (1987).

[29] M. Novick, personal communication (1988).

[30] J. H. Reif, "An optimal parallel algorithm for integer sorting," *Proc. 26th FOCS*, pp. 496-503.

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

[32] 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.

[33] 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).

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

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

[36] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *25th FOCS* (1984), pp. 12-22.