



The University of Sydney

**Efficient Parallel Algorithms on Chordal
Graphs with Sparse Tree Representation**

Technical Report Number 457

February 1993

Elias Dahlhaus

ISBN 0 86758 665 6

**Basser Department of Computer Science
University of Sydney NSW 2006**

Efficient Parallel Algorithms on Chordal Graphs with a Sparse Tree Representation

Elias Dahlhaus

Basser Dept. of Computer Science

University of Sydney

NSW 2006, Australia

e-mail: dahlhaus@cs.su.oz.au

FAX: (+61)-2-692-3838

Abstract

Chordal graphs are nothing else than intersection graphs of subtrees of a tree. We present nearly optimal algorithms for certain graph problems if the input graph is given by a collection of subtrees of a tree and the subtrees are given by their leaves. This generalizes results of Olariu, Schwing, and Zhang [14] and Chen [4] concerning the parallel complexity of problems on interval graphs provided the interval structure is given. We present first parallel algorithms that construct a breadth-first search tree and a depth-first search tree. These algorithms need a linear processor number with respect to the size of the input. The time is logarithmically bounded. The basic idea to compute a breadth-first search tree and to compute a depth-first search tree is to order the tree nodes in post-order and to determine a perfect elimination ordering by sorting all subtrees with respect to the post-ordering of the roots. For the computation of a breadth-first search or a depth-first search tree, we compute for each tree node, the maximum subtree or the minimum non root subtree that contains the tree node. Next we consider the problems Maximum Independent Set and Covering by Cliques. Both problems can be solved in a quadratic logarithmic time bound with $O(m/\log m)$ processors, where m is the size of the sparse tree representation of given chordal graph, i.e. the sum of the number of leaves of all subtrees of the tree representation of given chordal graph. Here we use a procedure that is similar to tree contraction that uses the same procedure for interval graphs [14] as a subprocedure. We also prove that Minimum Coloring and Maximum Clique can be solved in logarithmic time using a linear number of processors. To find a maximum clique and a minimum coloring, we compute, by tree contraction, for any tree node, the size of the clique it corresponds to, i.e. the number of subtrees of the tree representation the tree node belongs to. Finally we consider the problem to find a minimum dominating set in a directed path graph.

0 Introduction

A graph is called chordal if every cycle of length greater than three has two nonadjacent vertices which are joined by an edge. Chordal graphs are a generalization of interval graphs (vertices are a collection of intervals, two intervals are joined by an edge if they intersect). Moreover chordal graphs

are exactly the intersection graphs of subtrees of a tree [11], i.e. the vertices are subtrees of a fixed tree and two subtrees are joined by an edge if they have a common tree vertex.

Olariu, Schwing, and Zhang [14] considered the parallel complexity of several graph problems on interval graphs if the interval structure is given. They found out that the processor time product of minimum clique cover, maximum independent set, etc. can be bounded by $O(n) \log n$ where n is the number of intervals.

Note that subtrees of a fixed tree are uniquely determined by their leaves. A generalization of the considerations of Olariu, Schwing and Zhang is that the input is a tree and a collection of subtrees and the subtrees are given by their leaves. We call this representation of chordal graphs the *sparse tree representation*. If for example the input graph is a path graph, i.e. the subtrees are paths, then the sparse tree representation is of size $O(n)$. In so far optimal parallel algorithms for chordal graphs in their sparse tree representation induce also optimal parallel algorithms for path graphs and interval graphs in their path or interval representation respectively.

As [12] we even consider the problem of the computation of a depth-first search and a breadth-first search tree.

Afterwards we consider the problems to find a maximum independent set, a minimum clique cover, and a minimum coloring.

Finally we consider the problem to find a minimum dominating set. This problem is NP-complete even for chordal graphs but can be solved efficiently for strongly chordal graphs [10]. One problem is that it is not easy to check whether a collection of subtrees represents a strongly chordal graph. That is the reason why we consider only the problem to find a minimum dominating set for directed path graphs.

In section 1 we introduce the notations and basic concepts used in this paper. In section 2 we state some basic algorithms which are used in the whole paper. In section 3 we develop optimal parallel algorithm to compute a breadth-first search tree and to compute a depth-first search tree for a chordal graph in sparse tree representation. Section 4 discusses the parallel complexity of minimum clique cover and maximum independent set. In section 5 we consider the parallel complexity of Maximum Clique and Minimum Coloring. In section 6 we present a parallel algorithm that finds a minimum dominating set of a directed path graph in polylogarithmic time with a linear processor number with respect to the number of the size of the

path representation.

1 Notation

A *graph* $G = (V, E)$ consists of a *vertex set* V and an *edge set* E . Multiple edges and loops are not allowed. The edge joining x and y is denoted by xy .

We say that x is a *neighbor* of y iff $xy \in E$.

A *tree* is a cycle free connected graph. A *rooted tree* is a directed graph whose underlying undirected graph is a tree with a distinguished vertex r (the *root*) such that for any vertex v in T , there is a directed path to r . The unique vertex v' , such that (v, v') is a directed edge in the rooted tree T , is called the *parent* of v in T . v is called a *child* of v' if v' is the parent of v . In general, v' is an *ancestor* of v iff there is a directed path from v to v' in T . v is also called a *descendent* of v' .

A *subgraph* of (V, E) is a graph (V', E') such that $V' \subset V$, $E' \subset E$.

An *induced subgraph* is an edge-preserving subgraph, that means (V', E') is an induced subgraph of (V, E) iff $V' \subset V$ and $E' = \{xy \in E : x, y \in V'\}$.

A graph (V, E) is *chordal* iff each cycle $(x_0 \dots x_{k-1} x_0)$ of length greater than 3 has an edge $x_i x_j \in E$, $j - i \neq \pm 1 \pmod k$ (which joins vertices which are not neighbors in the cycle).

A subset V' of the vertex set V is *complete* iff all vertices of V' are pairwise joined by an edge.

An inclusion-maximal complete set is called a (*maximal*) *clique*.

Note that chordal graphs are a generalization of interval graphs, i.e. intersection graphs of intervals in the real line. The following characterization of chordal graphs due to Gavril [11] and Buneman [3] is essential for the whole paper.

Theorem 1 *A graph $G = (V, E)$ is chordal iff it is the intersection graph of vertices of subtrees of a tree, i.e. there is a tree T and a collection $\mathcal{S} = \{T_v : v \in V\}$ of subtrees of T such that $vw \in E$ iff the subtrees T_v and T_w have a common vertex of T .*

We call (T, \mathcal{S}) a *subtree representation* for G .

A graph is called a *path graph* if it is the intersection graph of paths of a tree.

A graph is called a *directed path graph* if it is the intersection graph of directed paths of a rooted tree. The tree together with the collection of paths is called a *directed path representation*.

In the whole paper, we assume that $G = (V, E)$ is given by its corresponding tree structure (T, \mathcal{S}) . Moreover we assume that any tree in \mathcal{S} is given by its *leaves*, i.e. by its vertices of degree one.

Let m be the sum of the number of vertices of T and the number of leaves of subtrees in \mathcal{S} . A vertex appearing as a leaf of k subtrees in \mathcal{S} is counted k times.

The parallel computation model in this paper is the concurrent read exclusive write parallel random access machine (CREW-PRAM).

2 Basic Algorithms

The first to be done is to change the input data structure in such a way that we can deal easily with the remaining problems.

We assume that the main tree T is given in such a way that each non root vertex has a pointer to its parent and a pointer to an array listing all children. The subtree T_v with leaves l_1, \dots, l_k is realized as an array containing l_1, \dots, l_k .

First we compute, for each subtree T_v represented by its leaves, the set T'_v of vertices of T_v which have more than one child in T_v . The vertices in T'_v are just those vertices in T_v that are least common ancestors of two non root leaves of T_v .

We make use of the following result of Schieber and Vishkin [15]:

Lemma 1 *The least common ancestors of m pairs of vertices of a tree T with n vertices can be computed in $O(\log n)$ time using $O(n + m) / \log n$ processors.*

To compute T'_v for a subtree T_v with k leaves, we have to take care that we have to compute only $O(k)$ least common ancestors.

We proceed as follows:

1. Let L_v be the set of leaves of T_v . If the root of T_v is a leaf of T_v then we erase it from L_v .
2. We compute on T a preorder and sort L_v with respect to the preorder.

3. We compute the least common ancestors of those vertices in L_v which are adjacent with respect to the preorder sorting.

The first step to check whether the root of T_v is in L_v is done as follows:

We select the vertex r in L_v with the smallest distance from the root of T and check whether it is an ancestor of all remaining vertices in L_v .

The distances of all vertices of T to the root can be computed in $O(\log n)$ time using $O(n/\log n)$ processors using Eulerian cycle techniques [16] combined with an optimal list ranking algorithm [6]. The vertex r of L_v with the minimum distance to the root of T can be determined in $O(\log k)$ time using $k/\log k$ processors. For all vertices in L_v simultaneously it can be checked whether r is an ancestor of all remaining vertices in L_v can be checked in $O(\log k)$ time using $O(k/\log k)$ processors after an $O(\log n)$ time $O(n/\log n)$ preprocessing on T [16].

L_v can be sorted with respect to preorder in $O(\log k)$ time using $O(k)$ processors [5]

Therefore the overall complexity to compute all vertex sets T'_v are $O(n + \sum_{v \in V} |L_v|)$ processors and $O(\log(n + \sum_{v \in V} |L_v|))$ time, i.e. $O(m)$ processors and $O(\log m)$ time.

Another step is to transform T into a binary tree. This can be done by the following standard method. Suppose c_1, \dots, c_k are the children of the vertex t . If t has at most two children then nothing is changed. If t has more than two children then we introduce new vertices t_2, \dots, t_{k-1} . The parent of c_1 is t , for $i = 2, \dots, k-1$, the parent of c_i becomes t_i , and the parent of c_k is t_{k-1} .

We have to take care that the smallest subtree of T containing the set L_v of leaves of T_v still contains all vertices in T_v . If the root of T_v is in L_v then we are done. Otherwise we have to determine the root of T_v as the least common ancestor of all vertices in L_v , which is nothing else than the least common ancestor of a pair of vertices in L_v with the smallest distance to the root of T . In that case we add the root of T_v to L_v .

Such a procedure can be implemented in $O(\log m)$ time using $O(m/\log m)$ processors.

3 Breadth-first and Depth-First Search

We first review the algorithm of P. Klein [12] to compute a breadth-first search and a depth-first search tree. We assume that a *perfect elimination ordering* $<$ on the vertices of G is known, i.e.

if $x < y, x < z, xy \in E$, and $xz \in E$ then $yz \in E$.

A depth-first search tree is determined by the parent function

$$P(x) = \min_{<} \{y | xy \in E, x < y\}$$

and a breadth-first search tree is determined by the parent function

$$Q(x) = \max_{<} \{y | xy \in E\}.$$

Suppose a subtree structure $(T, \{T_v | v \in V\})$ for the chordal graph $G = (V, E)$ is known. Then we can determine a perfect elimination ordering as follows:

1. We sort the vertex set of T in reversal order with respect to the distances to the root of T .
2. We sort the vertices $v \in V$ with respect to the numbers of the root of T_v .

It can easily be checked that such a sorting of V determines a perfect elimination ordering.

A breadth-first search tree can therefore be determined as follows:

1. For any $v \in V$, let r_v be the root of T_v .
2. $Q(v)$ is the maximal vertex w with respect to the above sorting, such that T_w contains r_v .

A depth-first search tree can be determined as follows:

1. Let r_v be the root of T_v .
2. If v is not the maximum vertex w with $r_v = r_w$ then $P(v)$ is the immediate successor of v with respect to above sorting of V .

3. If v is the maximum vertex w with $r_v = r_w$ then $P(v)$ is the minimum vertex z such that $r_v \in T_z$ and $r_z \neq r_v$.

Therefore to determine a breadth-first search tree and to determine a depth-first search tree can be reduced to the problem to determine for each vertex t of T , a the maximum w such that $t \in T_w$ (Maximum Membership) and the minimum z such that $t \in T_z$ and $r_z \neq t$ (Minimum Membership) respectively.

We assume that for any vertex v , the leaf set L_v and the root r_v of T_v and a perfect elimination ordering as described above are known.

We can reduce the problems Maximum Membership and Minimum Membership to the same problem restricted to directed path graphs:

For each $t \in L_v$, we introduce a path $[t, r_v]$ and we improve the perfect elimination ordering in such a way that for any v , the paths $[t, r_v]$ with $t \in L_v$ appear in consecutive order, i.e. we sort all $[t, r_v]$ with $t \in L_v$ with respect to the given perfect elimination ordering on V .

Note that a maximum (minimum) $[t', r_v]$ such that t is on the path $[t', r_v]$ corresponds to a maximum (minimum) T_v such that $t \in T_v$. Therefore above reduction reduces Maximum membership and Minimum Membership correctly to Maximum Membership and Minimum Membership restricted to directed path graphs. Moreover this reduction can be done by $O(m)$ processors in constant time.

Now we assume that a binary tree T and a collection of root directed paths $\{[s_v, r_v] | v \in V\}$ is given.

Maximum Membership can be solved very easily:

We initially let $F(t) = \max\{v | s_v = t\}$. Inductively, we set

$$G(t) = \max\{F(t), G(t_1), G(t_2)\}$$

where t_1 and t_2 are the children of t . $G(t)$ is the maximum path such that t is a member. $G(t)$ can be computed simultaneously for all vertices t of T in $O(\log m)$ time using $O(m)$ processors by tree contraction [1].

Minimum Membership can be reduced to the evaluation of a sequence of $Insert(x, A)$, $Delete(x)$, $Union(A, B)$, $Min(A)$ [2].

The vertices of T correspond to the sets. Let $t_i, i = 1, \dots, n$ be an enumeration of the vertices of T in reversal order with respect to the distance from the root. At each step i , we execute a couple of operations S_{t_i} corresponding to the vertex t_i .

Let t be a vertex of T with children t_1, t_2 . Then S_t is the sequence of the following operations:

$$A_t = \text{Union}(A_{t_1}, A_{t_2}), (\text{Insert}(v), s_v = t),$$

$$(\text{Delete}(v) : r_v = t), \text{Min}(A_t).$$

$\text{Min}(A_t)$ is just the smallest path $[s_x, r_x]$ containing t and not having t as root. By [2], we can execute this step in $O(\log m)$ time using $O(m)$ processors. Note that the length of the concatenation of the sequences S_t is $O(m)$.

Therefore:

Theorem 2 *If a chordal graph $G = (V, E)$ is given as a collection of subtrees of a tree and the subtrees are given by their leaves then a breadth-first search and a depth-first search tree can be computed in $O(\log m)$ time using $O(m)$ processors.*

4 Maximum Independent Set and Covering by Cliques

First we state the problem *Covering by Cliques*:

Input: A chordal graph

Output: A minimum cardinality set of maximal cliques of G that covers all vertices of G .

Note that for any vertex t of T , the vertices v with $t \in T_v$ form a complete set. Moreover, for any complete set C of vertices in G , there is a vertex t of T such that $t \in T_v$, for all $v \in C$.

Therefore covering by cliques is equivalent to the problem to find a minimum cardinality set T' of vertices of T such that for any $v \in V$, there is a $t \in T'$ with $t \in T_v$.

We can solve the problems Covering by Cliques and Maximum Independent Set by the following procedure (I do not make any statements on the complexity): Repeat

1. Add all leaves of T to $COVER$ that are roots of any T_v and add for each new $t \in COVER$ some v to $Independent$ such that the root r_v of T_v is t .
2. Erase all v from V such that T_v contains some $t \in COVER$.
3. Erase all leaves of T from T .

until T is empty.

Our aim is to parallelize this procedure by a tree contraction technique.

We use a tree contraction scheme of [13] and [1].

number the leaves of T from left to right;

Repeat

1. erase each odd numbered leaf that is a left child;
2. identify each vertex with one child with its child;
3. erase each odd numbered leaf that is a right child;
4. identify each with one child with its child;
5. renumber the leaves from left to right;

until T is empty.

We simulate the procedure to solve Maximum Independent Set and Covering by Cliques as follows. Note that in the algorithm, each node of T corresponds to a chain of nodes of the initial tree T . We denote the initial tree T by T_0 . Each node t of T has a pointer to a list l_t of nodes of T_0 . Initially l_t consists of the node t . We assume that, for any $v \in V$, the set L_v of leaves and the root r_v are known. The idea of the algorithm is that for any leaf t , the problems Covering by Cliques and Maximum Independent Set are solved on for the intervals corresponding to subtrees with the root in l_t . Afterwards all subtrees T_v which contain a node t of T_0 corresponding to a cover clique are erased. That means if a leaf t of T is erased the we execute the algorithm of Olariu, Schwing, and Zhang [14].

To make the paper more self contained, we give a short description of the algorithm of Olariu, Schwing, and Zhang:

The input is a set \mathcal{S} of intervals $[s_v, r_v]$ of the real line.

1. We compute an interval $I_0 = [r_v, s_v]$ such that r_v is minimal.
2. For each interval $[s_v, r_v] \in \mathcal{S}$, let $right[s_v, r_v]$ be an interval $[s_w, r_w] \in \mathcal{S}$ such that $r_v < s_w$ and r_w is minimal.
3. A maximum independent set *Independent* is determined by multiple application of *right* to I_0 and a minimum covering by cliques is determined by taking all r_w into *Cover* such that $[r_w, s_w] \in Independent$.

Olariu, Schwing, and Zhang proved that this algorithm can be implemented in $O(\log |\mathcal{S}|)$ time using $|\mathcal{S}|$ processors. It is easily seen that in case of interval graphs, this procedure does the same as above general procedure to compute a Covering by Cliques.

The algorithm to compute a maximum independent set and a minimum clique cover for any chordal graph in sparse representation works as follows:

For any $t \in T_0$, let $S_t = \{v \in V | t \in L_v\}$ and $R_t := \{v \in V | r_v = t\}$;
repeat:

1. for each odd numbered leaf t of T which is a left child:
 - (a) number the elements of the list l_t by a list ranking;
 - (b) compute a covering C_t by cliques and a maximum independent set I_t for those $v \in V$ with $r_v \in l_t$ by the algorithm of [14] (note that this is the computation of a maximum independent set and of a covering by cliques in an interval graph);
 - (c) add C_t to *COVER* and I_t to *INDEPENDENT*;
 - (d) compute the $t' \in C_t$ with the largest number;
 - (e) for $v \in \bigcup_{s \in l_t, s \text{ is a descendent of } t'} S_s$,
erase v from V and any set $S_{t'}$, $t' \in L_v$ and from R_{r_v} .
 - (f) if $t'' \in l_t$ is a proper ancestor of t' and $t'' \in L_v$ then :
erase t'' from L_v and add the first element $fP(t)$ of the list $l_{P(t)}$ of the parent $P(t)$ of t in T to L_v ; add v to $S_{fP(t)}$;
 - (g) erase t from T ;
2. for each node t of T with exactly one child t_1 , concatenate the list l_{t_1} with the list l_t , make the parent of t the parent of t_1 and erase t from T ;

3. for each odd numbered leaf t which is a right child:
 - (a) number the elements of the list l_t by a list ranking;
 - (b) compute a covering C_t by cliques and a maximum independent set I_t for those $v \in V$ with $r_v \in l_t$ by the algorithm of [14] (note that this is the computation of a maximum independent set and of a covering by cliques in an interval graph);
 - (c) add C_t to *COVER* and I_t to *INDEPENDENT*;
 - (d) compute the $t' \in C_t$ with the largest number;
 - (e) for $v \in \bigcup_{s \in l_t, s \text{ is a descendent of } t} S_s$,
erase v from V and any set $S_{t'}$, $t' \in L_v$ and from R_{r_v} .
 - (f) if $t'' \in l_t$ is a proper ancestor of t' and $t'' \in L_v$ then :
erase t'' from L_v and add the first element $fP(t)$ of the list $l_{P(t)}$ of the parent $P(t)$ of t in T to L_v ; add v to $S_{fP(t)}$;
 - (g) erase t from T ;
4. for each node t of T with exactly one child t_1 , concatenate the list l_{t_1} with the list l_t , make the parent of t the parent of t_1 and erase t from T ;

until T is empty;

output *COVER* and *INDEPENDENT*.

Following the argument that the algorithm of [14] correctly simulates the first Maximum Independent Set and Covering by Cliques algorithm, it is easily seen that last algorithm simulates first Maximum Independent Set and Covering by Cliques algorithm correctly.

The complexity can be checked as follows:

The depth of the loop is $O(\log n)$ [1] where n is the number of vertices of T .

The concatenations of lists l_t need an overall time of $O(\log n)$ and an overall processor number of $O(n/\log n)$, because one concatenation step needs one time unit and the overall processor and time bound is the same as the processor and time bound for tree contraction [1].

The number of insertions of v into some S_t is bounded by $O(\log n)$ simultaneous insertions of k elements v into some S_t needs $O(\log k)$ time and

$O(k)$ workload. Therefore we can bound the time for insertions of elements $v \in V$ into sets S_t by $O(\log^2 m)$ and the processor number by $O(m/\log m)$.

Erasing v from V and S_t needs constant time. The overall complexity for these kind of operations is therefore $O(\log m)$ time and $O(m/\log m)$ processors.

Let \mathcal{S}_t be the set of (nonerased) v such that the root r_v of T_v is in the list l_t . Then Covering by Cliques and Maximum Independent Set restricted to the interval structure \mathcal{S}_t can be solved in $O(\log |\mathcal{S}_t|)$ time with $O(|\mathcal{S}_t|)$ processors. Since each $v \in V$ appears only in one \mathcal{S}_t , the overall complexity of the applications of interval graph Maximum Independent Set and Covering by Cliques is of $O(m/\log m)$ processors and $O(\log^2 m)$ time.

Therefore:

Theorem 3 *For chordal graphs in sparse representation of size m , Maximum Independent Set and Covering by Cliques can be solved in $O(\log^2 m)$ time using $O(m/\log m)$ processors.*

5 Minimum Coloring

First we compute, for each vertex t of T , the number of T_v with $t \in T_v$. This gives us also a solution of the maximum clique problem. We assume that for any $v \in V$, the set L_v of leaves of T_v and the set T'_v of vertices of T_v is known which have both children in T_v , i.e. which are least common ancestors of some vertices in L_v .

We determine, for each $t \in T$, the number l_t of $v \in V$ such that $t \in L_v \cup T'_v$, the number r_t of $v \in V$ such that t is the root of T_v , and the number l'_t of $v \in V$ such that $t \in T'_v$.

For all leaves t of T , l_t is the number c_t of $v \in V$ with $t \in T_v$.

Let t be a nonleaf vertex with the children t_1 and t_2 . Then the number c_t of $v \in V$ with $t \in T_v$ is $c_{t_1} - r_{t_1} + c_{t_2} - r_{t_2} + l_t - l'_t$. Note that $c_{t_i} - r_{t_i}$ is the number of $v \in V$ such that t and t_i are in T_v . This number has to be added to l_t . To mind that a $v \in V$ is counted twice we have to subtract l'_t .

Such a procedure can be done by tree contraction in $O(\log m)$ time using $O(m/\log m)$ processors.

We begin with a first rough algorithmic description to compute a minimum coloring on V .

1. determine the maximum size χ of a clique and make a vertex of T corresponding to a maximum clique the root of T ;
2. for each vertex t of T , sort the clique $C_t = \{v \in V | t \in T_v\}$ corresponding to t to a sequence S^t in such a way that first those $v \in C_t$ appear, such that t is not the root of T_v .
3. for each vertex t' of T with t as its parent, sort C_t to a sequence $S_{t'}$ such that those $v \in C_t$ appear first that also appear in $C_{t'}$;
4. for each $v \in V$ with root r_v and with index i in the sequence S^{r_v} , determine the first ancestor t'_v with parent t_v such that $C_{t'_v}$ has a cardinality of at least i and let $F(v)$ be the i^{th} element of $S_{t'_v}$
5. for any $v \in V$, let $G(v)$ be the $v' = F^k(v)$ such that the root of $T_{v'}$ is the root of T ;
6. color each $v \in V$ with the color of $G(v)$.

Note that T_v and $T_{F(v)}$ are disjoint. Note that for any v with number i in S^{r_v} , no C_t between r_v and t_v has cardinality $\geq i$. Therefore no v' , such that $r_{v'}$ is between r_v and t_v , can have the property $F(v) = F(v')$. Since $t_v \in T_{F(v)}$, all $T_{v'}$, such that $r_{v'}$ is between t_v and $r_{F(v)}$, intersect $T_{F(v)}$ and therefore $F(v') \neq F(v)$. Therefore if $F(v) = F(v')$ then r_v is not an ancestor of $r_{v'}$ and vice versa. Therefore if $F(v) = F(v')$ then T_v and $T_{v'}$ are disjoint. Moreover if r_v is an ancestor of $r_{v'}$ and $G(v) = G(v')$ then v is of the form $F^k(v')$. Therefore if $G(v) = G(v')$ then T_v and $T_{v'}$ are disjoint.

It remains to improve this algorithm to a parallel efficient algorithm with respect to a sparse representation.

We assume that that T is binary and that for any $v \in V$, the set L_v of leaves and the set T'_v of two children vertices of T_v are known. In S_t , we only can number those $v \in C_t$ which have t as its root. Just in the maximum clique algorithm, we determined the number R_t of those T_v which have t as its root. Even we determined $|C_t|$. Therefore we can determine the number $n_t = |C_t| - R_t$ of those $v \in C_t$ such that t is not the root of T_v . We number all v with $t = r_v$ from $n_t + 1$ upwards. Therefore S^t restricted to those v with t as root of T_v can be computed in $O(\log m)$ time using $O(m)$ processors.

To find, for any v with number i in S^t , the first ancestor t'_v of r_v , such that the clique corresponding to the parent t_v of t'_v has at least i elements,

we use the same binary search strategy as in [8] to compute, for any edge of a weighted tree, the first ancestor edge with a larger weight. This can be done in $O(\log m)$ time using $O(m)$ processors.

To determine $F(v)$, for all $v \in V$ with $t'_v = t'$, we proceed as follows:

1. Determine the maximum number in S^{r_v} such that $t'_v = t'$. Denote this number by $\max_{t'}$.
2. Find $\max_{t'}$ many $v \in V$ such that the parent t of t' belongs to T_v but not t' itself.
3. Determine an $S'_{t'}$ -numbering for these T_v .

First and third step can be done easily in $O(\log m)$ time using $O(m/\log m)$ processors for all t' simultaneously.

It remains to fill out the second step.

First we find as many $v \in V$ as $t \in L_v$ and $r_v \neq t$. If this are not enough then we find as many $v \in V$ such that $r_v = t \in L_v$ and t' is not an ancestor of any $s \in L_v$ with $s \neq t$, i.e. t is the only vertex of T_v or the sibling t'' of t' is an ancestor of some $s \in L_v \setminus \{r_v\}$. This steps can be done in $O(\log m)$ time with $O(m)/\log m$ processors. If we still did not find enough $v \in C_t \setminus C_{t'}$ then we have to find $v \in V$ such that some $s \in T'_v \cup L_v$ is a descendent of t'' and some $s' \in T'_v \cup \{r_v\}$ is a proper ancestor of t .

It remains to find enough $v \in V$ such that T_v passes the edge $t''t$ of T , t is not the root of T_v and T_v does not contain t' . This is equivalent to the problem to find enough $v \in V$ such that $t \notin T'_v$ and T_v passes the edge $t''t$ of T .

For each $s \in L_v \cup T'_v \setminus \{r_v\}$, we determine the first ancestor $P'_v(s)$ of s in $T'_v \cup L_v$. This can be done (in $O(\log m)$ time using $O(m)$ processors) by determining, for each $s' \in T'_v$, the left and the right immediate descendent of s' in T'_v , i.e. by determining the immediate successor of s' in T_v with respect to preorder on T and with respect to postorder on T .

Let $P''_v(s)$ be the child of $P'_v(s)$ that is an ancestor of s . We consider the paths $[s, P''_v(s)]$ for all vertices $s \in T$ $s \in T'_v$. Note that any $s \in T_v$ appears in at least one of the paths $[s, P''_v(s)]$.

To find enough T_v passing t'' and t but not passing t' and not having t as root is equivalent to the problem to find the same number of paths $[s, P''_v(s)]$ passing t'' and t .

Therefore it remains to solve the following problem:

Input: A binary tree T , a collection P of root directed paths on T , and a labeling l of the edges of T by positive integers.

Output: For each edge e of T , find $l(e)$ paths of P passing e .

Here we again use the tree contraction principle of [1]. Note that, in each step, disjoint pairs of neighbor vertices are each unified to one vertex. We define a new tree structure Tr with parent function Pa . Let t_1 and t_2 be unified to a vertex t then t is set to be the parent $Pa(t_1) = Pa(t_2)$ of t_1 and t_2 .

This tree Tr has the following properties:

1. The depth of Tr is $O(\log m)$.
2. The leaves of Tr are the vertices of T .
3. For each vertex tr of Tr , the leaves of the underlying subtree of tr with root tr form a subtree of T . We denote the set of leaves of the subtree of Tr with root tr by $Cl(tr)$ (the cluster corresponding to tr).

Note that for children tr_1 and tr_2 of tr in Tr , there is an edge t_1t_2 of T with $t_1 \in Cl(tr_1)$ and $t_2 \in Cl(tr_2)$. If t_2 is the parent of t_1 then tr_2 is called the *larger child* of tr and tr_1 is called the *smaller child* of tr . t_1t_2 is called the separating edge of tr and is denoted by e_{tr} .

Step 1: We determine, for each root directed path $[t_1, t_2]$ of P , the least common ancestor $A(t_1, t_2)$ of t_1 and t_2 in Tr .

Let tr be an ancestor of t_1 in Tr and a proper descendent of $A(t_1, t_2)$ in Tr . Moreover, we assume that the smaller child tr_1 of tr is an ancestor of t_1 . Then the path $[t_1, t_2]$ covers the path from e_{tr} to the root of $Cl(tr)$ in T .

We call the path of T from e_{tr} to the root of $Cl(tr)$ including e_{tr} also the *main path* of tr and is denoted by m_{tr} .

Note that $[t_1, t_2]$ is the union

1. of all m_{tr} such that the smaller child of tr is an ancestor of t_1 and tr is a proper descendent of $A(t_1, t_2)$ and
2. of an initial segment p of $m_{A(t_1, t_2)}$.

Step 2: If $[t_1, t_2] \in P$, tr and its smaller child tr_1 are proper descendents of $A(t_1, t_2)$ and ancestors of t_1 then we put $[t_1, t_2]$ into the set $M(tr)$, assigning that $[t_1, t_2]$ covers the main path of tr .

Step 3: We put $[t_1, t_2]$ into the set $Part_{A(t_1, t_2)}$ assigning that $[t_1, t_2]$ partially covers the main path of $A(t_1, t_2)$.

Step 4: For each vertex tr of Tr , we sort the paths $[t_1, t_2]$ in $Part_{tr}$ with respect to the distance of t_2 to the root of T .

Step 5: For each edge e on the main path of tr , we determine the number $n(e, tr)$ of $[t_1, t_2] \in Part_{tr}$ such that t_2 is an ancestor of e , i.e. $[t_1, t_2]$ passes e . This can be done by parallel prefix computation: Let e' be the immediate ancestor edge of e and let $e' = t't''$. Then $n(e, tr) = n(e', tr) + |\{[t_1, t_2] \in Part(tr) | t_2 = t''\}|$.

Step 6: For each edge T -edge $e = t_1t_2$, we determine all ancestors of the least common ancestor of t_1 and t_2 in Tr and select those tr such that e is on the main path of tr . Let tr_1^e, \dots, tr_k^e be an enumeration of those tr . To find $l(e)$ paths passing through e , we continue as follows:

Set $l_1(e) = l(e)$;
for $i = 1, \dots, k$, do:

1. If $l_1(e) > |M(tr_i^e)|$ then put all paths of $M(tr_i^e)$ into $Path(e)$ and set $l_1(e) = l_1(e) - |M(tr_i^e)|$,
else put $l_1(e)$ paths in $M(tr_i^e)$ into $Path(e)$ and leave the loop.
2. If $l_1(e) > n(e, tr_i^e)$ then then put the first $n(e, tr_i^e)$ paths in $Part(tr_i^e)$ into $Path(e)$ and set $l_1(e) = l_1(e) - n(e, tr_i^e)$ else put the first $l_1(e)$ paths of $Part(tr_i^e)$ into $Path(e)$ leave the loop.

$Path(e)$ is the collection of $l(v)$ paths passing e .

It remains to do a complexity analysis.

Step 1 can be done in $O(\log m)$ time using $O(m/\log m)$ processors since the least common ancestor of at most $O(m)$ pairs on a tree with $O(m)$ vertices can be determined in these bounds.

In step 2, we determine, for each t_1 , all $O(\log m)$ ancestors (in $O(\log m)$ time). Then we check, for each ancestor tr of t_1 whether it is a proper descendent of $A(t_1, t_2)$ and whether the smaller child is an ancestor of t_1 (in constant time for each t_1 and each tr and therefore in $O(\log m)$ time with $O(m)$ processors for all $[t_1, t_2]$ simultaneously). For each $[t_1, t_2]$, we insert $O(\log m)$ copies of $[t_1, t_2]$ into sets $M(tr)$ and therefore we have a time bound

of $O(\log m)$ and a processor bound of $O(m)$, since simultaneous insertion of $O(k)$ elements into sets needs $O(\log k)$ time and $O(k/\log k)$ processors.

Step 3 needs $O(m/\log m)$ processors and $O(\log m)$ time.

Step 4 needs $O(m)$ processors and $O(\log m)$ time because sorting of m items needs this bound [5].

Step 5 is a parallel prefix computation and is therefore bounded by $O(\log m)$ time and $O(m/\log m)$ processors.

The loop of step 6 is applied $O(\log k)$ times. If we proceed straight forward then each application of the loop needs $O(\log m)$ time. Note that the else statements are applied only once. The then-part can be executed in that way that in the loop, we only set a marker that all paths in $M(tr_i^e)$ and the first $l(e)$ paths in $Part(tr_i^e)$ have to be taken. The real insertion step will be done for all marked items in parallel with $l(e)$ processors in $O(\log m)$ time and therefore, for all e simultaneously, we get a processor bound of $\sum_e l(e) \leq O(m)$ and a time bound of $O(\log m)$.

Therefore:

Theorem 4 *For chordal graphs in sparse tree representation of size m , a minimum coloring can be determined in $O(\log m)$ time using $O(m)$ processors.*

6 Domination Problems

Domination is the problem to find a minimum subset D of the vertex set of a graph such that each vertex is a neighbor of some vertex in D or belongs to D .

This problem is NP-complete, even for chordal graphs [10]. For strongly chordal graphs, an efficient sequential algorithm is due to [10] and an efficient parallel algorithm is due to [9]. The problem is that we have not an easy characterization of subtree representations of strongly chordal graphs.

An interesting subclass of strongly chordal graphs are *directed path graphs*. A graph $G = (V, E)$ is called a directed path graph iff there is a rooted tree T and a collection $\mathcal{P} = \{P_v | v \in V\}$ of root directed paths such that $xy \in E$ iff P_x and P_y share at least one vertex. (T, \mathcal{P}) is also called the *path representation* of G . Any path p in \mathcal{P} is determined by its two border vertices s_p and t_p . s_p is the border vertex of p that is closer to the root of T . s_p is also called the root of p .

Our goal is to reduce Domination to the problem to find a minimum number of paths in \mathcal{P} that cover the whole tree.

For this reason, we eliminate all $t \in T$ that do not represent maximal cliques. We first erase all vertices t of T that are not roots of some $p \in \mathcal{P}$, i.e. all paths containing t contain also the parent of t . This is done by the following algorithm.

1. We mark all vertices of T that are roots of some $p \in \mathcal{P}$.
2. For any marked $t \in T$, $parent'(t)$ is the next proper ancestor of t that is marked.
3. The new tree T' consists of all marked vertices of T and is determined by $parent'$ as parent function.

Lemma 2 *With $p' = p \cap T'$ and $\mathcal{P}' = \{p' | p \in \mathcal{P}\}$, (T', \mathcal{P}') represents the same directed path graph as (T, \mathcal{P}) .*

Proof: Trivially p' is a root directed path in T' , for any $p \in \mathcal{P}$. Since p_1 and p_2 are root directed, p_1 and p_2 share a vertex if the root of p_1 is in p_2 or the root of p_2 is in p_1 . Therefore if p_1 and p_2 share a vertex in T then they share a vertex in T' . \square

Secondly, we erase those $t \in T$ such that there is a child t' of t with the property that all $p \in \mathcal{P}$ that contain t also contain t' . Under the assumption that the graph G is connected, t' is the only child. *Therefore we have to erase those $t \in T$ that have only one child and there is no $p \in \mathcal{P}$ such that $t_p = t$.*

Proposition 1 *Suppose for all $p \in \mathcal{P}$, s_p and t_p are known and $n = |T| + |\mathcal{P}|$. Then we can compute a directed path representation on a CREW-PRAM in $O(\log n)$ time with $O(n)$ processors that represents the same graph such that the tree vertices correspond to the maximal cliques.*

Proof: As mentioned before, we first eliminate non roots. Above procedure can be implemented in constant CRCW-time with $O(n)$ processors and therefore in $O(\log n)$ CREW-time with $O(n)$ processors. The update of the parent function (find the next marked ancestor) can be done in $O(\log n)$ time with $O(n/\log n)$ processors by Euler cycle techniques [16]. Even we have to

update t_p if t_p is not marked. The new vertex t_p is the next marked ancestor of the old vertex t_p . We need $O(n/\log n)$ processors and $O(\log n)$ time.

The second step is to eliminate one child vertices that are not of the form t_p . We mark all $t \in T$ that have more than one child or there is some $p \in \mathcal{P}$ such that $t = t_p$. This is done in $O(\log n)$ time with $O(n)$ processors on a CREW-PRAM. For any unmarked $t \in T$, we find, by Eulerian cycle techniques, the first marked descendent $m(t)$ of t . If the parent of t is marked then the new parent of $m(t)$ is the parent of t . Note that all t with $m(t) = t'$ appear consecutively on a directed path that begins with the parent of t' . For any $p \in \mathcal{P}$, if s_p is not marked then s_p is updated by $m(s_p)$. \square

We first present a simple sequential algorithm that computes a minimum number of paths that cover the whole tree.

1. For each leaf t of T , select a path $p_t \in \mathcal{P}$ of maximal length that contains t . Add p_t to *COVER*.
2. Mark all $t \in T$ that are not contained in some p_t such that t is a leaf of T .
3. Update T : The parent of each marked vertex $t \in T$ is the next marked ancestor of t . All unmarked vertices of T are erased from T .
4. The previous steps are repeated until T is empty.

Lemma 3 *COVER is a minimum set of paths in \mathcal{P} that covers all vertices of the initial tree T .*

Proof: Each leaf of T must be covered by some $p \in \mathcal{P}$. Since \mathcal{P} is a collection of root directed paths, a path p_t of maximum length containing a leaf t is also an inclusion maximal path containing t . Moreover, by the same reason, no two leaves of T share a path. Therefore the collection of p_t , t is a leaf, is a minimum collection of paths covering all leaves of T and the vertices of T that are in some path p_t are exactly those vertices that share a path with some leaf of T . Therefore we may reduce the problem to find a minimum set of paths that covers all vertices of T to all vertices that are not in some p_t . \square

It remains to parallelize this procedure.

As in the problem of Covering by Cliques, we consider leaf chains instead of leaves.

To get a minimum cover of the leaf chains such that a maximum number of remaining vertices of T is covered, we proceed as follows.

1. For each $t \in T$, we select a path q_t that contains t and a maximum number of ancestors of t , i.e. the root of q_t has a minimum distance from the root of T .
2. For each leaf chain $L = \{t_1, \dots, t_k\}$, compute a maximal list

$$s_1, p_1, s_2, p_2, \dots, s_l, p_l$$

such that $s_i \in L$, $s_1 = t_1$, $p_i = q_{s_i}$, and $s_{i+1} = \text{parent}_T(\text{root}(p_i))$. Mark all vertices that appear in some p_i and add all p_i to *COVER*.

3. Erase all $t \in T$ that are marked. Update the parent function of T (first unmarked ancestor) and for any $p \in \mathcal{P}$, t_p (first unmarked ancestor if marked) and s_p (last unmarked ancestor of t_p that is a descendent of s_p).

To get a minimum covering of all vertices of T by paths of \mathcal{P} , we apply last procedure $O(\log n)$ times.

We continue with a complexity analysis. First step has been considered in the computation of a breadth-first search tree and needs $O(\log n)$ time and $O(n/\log n)$ processors. Second step can be done by tree contraction on a tree T_L where the vertices of T_L are vertices of L and paths intersecting L and the parent of a path p in T_L is $\text{parent}_T(\text{root}(p))$ and the parent of a vertex $t \in L$ in T_L is q_t . Therefore we have a time bound of $O(\log n)$ and a processor bound of $O(n/\log n)$ for the second step. The computations in the third step can be done by Eulerian cycle techniques [16] in $O(\log n)$ time with $O(n/\log n)$ processors.

We apply these steps $O(\log n)$ times and get the following result.

Theorem 5 *Domination restricted to directed path graphs can be done in $O(\log^2 n)$ time with $O(n/\log n)$ processors on a CREW-PRAM.*

7 Conclusions

We considered problems which have linear time solutions for chordal graphs in general and found out that, even in a sparse tree representation, they could

be solved in polylogarithmic time with a linear processor number. Even we considered the problem to find a minimum dominating set for directed path graphs. It might be interesting to find an efficient parallel algorithm to check whether a sparse tree representation describes a strongly chordal graph. A parallel algorithm that recognizes strongly chordal graphs in polylogarithmic time with a linear processor number with respect to the number of vertices and edges is due to [7]. This algorithm is not easy. It would be interesting to find a simpler parallel algorithm that recognizes strongly chordal graphs.

References

- [1] K. Abrahamson, N. Dadoun, D. Kirkpatrick, T. Przytycka, A Simple Parallel Tree Contraction Algorithm, *Journal of Algorithms* 10 (1988), pp. 287-302.
- [2] M. Atallah, M. Goodrich, S.R. Kosaraju, Parallel Algorithms for Evaluating Sequences of Set-Manipulation Operations, preprint, preliminary version in *VLSI-Algorithms and Architectures, 3^d Agean Workshop on Computing, AWOC 88*, LNCS 319 (1988), pp. 1-10.
- [3] P. Buneman, A Characterization of Rigid Circuit Graphs, *Discrete Mathematics* 9(1974), pp. 205-212.
- [4] L. Chen, Optimal Parallel Time Bounds for the Maximum Clique problem on Intervals, *Information processing letters* 42 (1992), pp. 197-201.
- [5] R. Cole, Parallel Merge Sort, *27th Symposium on Foundations of Computer Science* (1986), pp. 511-516.
- [6] R. Cole, U. Vishkin, Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems, *27th Annual Symposium on Foundations of Computer Science* (1986), pp. 478-491.
- [7] E. Dahlhaus, Chordale Graphen im besonderen Hinblick auf parallele Algorithmen (German), Habilitation Thesis, University of Bonn.
- [8] E. Dahlhaus, Fast parallel algorithm for the single link heuristics of hierarchical clustering, *Proceedings of the fourth IEEE Symposium on Parallel and Distributed Processing* (1992), pp. 184-186.

- [9] E. Dahlhaus, P. Damaschke, The Parallel Solution of Domination Problems on Chordal and Strongly Chordal Graphs, to appear.
- [10] M. Farber, Domination, Independent Domination and Duality in Strongly Chordal Graphs, *Discrete Applied Mathematics* 7(1984), pp. 115-130.
- [11] F. Gavril, The Intersection Graphs of Subtrees in Trees are Exactly the Chordal Graphs, *Journal of Combinatorial Theory Series B* 16 (1974), pp. 47-56.
- [12] P. Klein, Efficient Parallel Algorithms for Chordal Graphs, *29th Symposium on Foundation of Computer Science* (1988), pp. 150-161.
- [13] S.R. Kosaraju, A. Delcher, Optimal Parallel Evaluation of Tree-Structured Computations by Raking, *VLSI-Algorithms and Architectures, 3^d Agean Workshop on Computing, AWOC 88*, LNCS 319 (1988), pp. 101-110.
- [14] S. Olariu, J. Schwing, J. Zhang, Optimal Parallel Algorithms for Problems Modeled by a Family of Intervals, *IEEE Transactions on Parallel and Distributed Systems* 3 (1992), pp. 364-374.
- [15] B. Schieber, U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM Journal on Computing* 17 (1988), pp. 1253-1262.
- [16] R. Tarjan, U. Vishkin, Finding Biconnected Components in Logarithmic Parallel Time, *SIAM-Journal on Computing* 14 (1985), pp. 862-874.