Efficient Parallel Modular Decomposition (Extended Abstract)

Elias Dahlhaus

Basser Dept. of Computer Science,
University of Sydney,
NSW 2006, Australia,
e-mail: dahlhaus@cs.su.oz.au and dahlhaus@cs.uni-bonn.de

Abstract. Modular decomposition plays an important role in the recognition of comparability graphs and permutation graphs [12]. We prove that modular decomposition can be done in in polylogarithmic time with a linear processor bound.

Keywords. Parallel algorithms, graph theory, graph algorithms

1 Introduction

By a *module* of a graph $G = (V, E)$, we define a subset V' of the vertex set V such that all vertices in V' have the same neighbors outside V'. The decomposition of a graph into modules has many applications. First, modular decomposition allows a more compact representation of a graph. Modular decomposition plays also an important role in the recognition of certain graph classes like interval graphs, comparability graphs, and permutation graphs (see for example [12]). One only has to check that the 'module components' are permutation, interval, or comparability graphs respectively. Modular decomposition is also helpful to solve classical graph problems like graph coloring or maximum clique. The best known sequential algorithms for modular decomposition have a time bound of $O(n + m)$ [9, 4]. The only problem of these optimal algorithms is that a big mathematical effort is necessary to understand them. Previous efficient (not linear time) algorithms are due to [10] and [14]. Parallel algorithms with a non linear processor bound are due to M. Morvan and L. Viennot and to M. Novick (both unpublished).

Very related to the problem of modular decomposition is the problem to recognize cographs [3]. In some sense, cographs are those graphs which are totally decomposable into modules, i.e. each induced subgraph with at least four vertices has a module of size at least two.

The goal of this paper is to develop a parallel algorithm for modular decomposition. The processor number is linear, and the time is polylogarithmic. Compared to the algorithms of [9, 4], the algorithm presented here is quite simple. We combine some ideas of the parallel cograph recognition algorithm of [5] and the sequential divide and conquer modular decomposition method of [6].

As in [5], we distinguish between low degree, middle degree, and high degree vertices. In case of the existence of a middle degree vertex, we consider its neighbors, non neighbors, do modular decompositions separately, and combine the modular decomposition to a big one. If there are only low degree vertices then we compute a spanning tree and compute a first approximation of an elimination ordering as done by P. Klein [8]. Up to modules containing the root of the spanning tree, all modules appear only in one level. We make a modular decomposition in each level and combine these. In case that all vertices have high degree we only have to consider the complement.

The only situation that is left is that we have low degree vertices and high degree vertices but no middle degree vertices. There we distinguish between small size and large size modules. We shall find out that first contain only low degree vertices or only high degree vertices. Large size modules have the property that the neighborhood consists only of large degree vertices and the non neighborhood consists only of low degree vertices.

Section 2 introduces the notations and basic notions that are of relevance in the whole paper. Section 3 discusses describes the algorithm. Section 4 presents a complexity analysis of the algorithm.

2 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 $xy \in E$. The *neighborhood* of x is the set $\{y : xy \in E\}$ consisting of all neighbors of x and is denoted by $N(x)$. The neighborhood of a set of vertices V' is the set $N(V') = \{y|\exists x \in V', yx \in E\}$ of all neighbors of some vertex in 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, i.e. (V', E') is an induced subgraph of (V, E) iff $V' \subset V$ and $E' = \{xy \in E : x, y \in V'\}$.

By a *module* of $G = (V, E)$, we mean a subset V' of V such that with $y \in V \setminus V'$ and $u, v \in V'$, both uy and vy are in E or none of uy and vy is in E.

We compute not all modules but those modules X which do not *overlap* with other modules, i.e. there is no module Y that has a nonempty intersection with X and neither $X \subset Y$ nor $Y \subset X$. We call such modules also *overlap free*. Note that the overlap-free modules of any graph form a tree with respect to set inclusion, i.e. two overlap free modules are disjoint or comparable with respect to set inclusion. The notions of parent and child modules can be defined in an obvious way. The parent $P(X)$ of an overlap-free module X is the unique smallest overlap free module that is a proper superset of X. Y is a child module of X if and only if Y is an inclusion maximal proper overlap free submodule of X. The algorithm we present later computes, for a given graph G, a root directed tree T_G representing the overlap free modules in their tree like ordering with respect to set inclusion.

There are the following types of overlap free modules:

Prime Modules First define a graph $G = (V, E)$ with more than two vertices as *prime* if all the only modules are V and the subsets of V containing exactly one vertex. An overlap free module $X \subseteq V$ is called *prime* if the graph that results from $G[X]$ after identification of all vertices in the same child module of X is prime.

Degenerated Modules An overlap free module X is *degenerated* if the graph that results from $G[X]$ after identification of all vertices appearing in the same child module is complete or forms an independent set. X is called *positively degenerated* or also called a 1-module if the resulting graph is complete, i.e. all vertices are pairwise adjacent. Otherwise, if the resulting graph forms an independent set, the degenerated submodule X is called *negatively degenerated module* or a 0-module.

Note that any module, overlap free or not, is a prime module or a subset of a degenerated module X with the property that each child module is a subset of X or disjoint with X (compare [9]).

The goal of this algorithm is, given a graph (V, E), to compute a rooted tree T_G such that the leaves are the vertices of G and each inner node t represents an overlap free module, i.e. the leaves that are descendents of t form an overlap free module of G. We call this tree T_G also the *modular tree* of G.

The computation model is the CRCW-PRAM. We measure each primitive operation by one time unit.

3 The Algorithm

The algorithm is a divide and conquer strategy. We reduce the problem to subgraphs with at most $3/4|V|$ vertices to get a logarithmic recursion depth. We distinguish between vertices of low degree, middle degree, and high degree. A vertex v of $G = (V, E)$ is of *low degree* if it is of degree at most $1/4|V|$. If the degree is at least $3/4|V|$ then the vertex is called a *high degree* vertex. If a vertex is neither a high degree nor a low degree vertex then it is called a *middle degree* vertex.

3.1 First Case: There is a Middle Degree Vertex

Suppose there is a middle degree vertex u. Clearly each module X that does not contain u has the property that all vertices of X are in the neighborhood of u or none of them. Therefore every module not containing u is a prime module of the neighborhood or the non neighborhood of u or a subset of a degenerate module of the neighborhood or of the non neighborhood of u.

We first find out the modules that do not contain u. Let L_1 be the set of neighbors of u and L_2 be the set of non neighbors of u. Note that each module that does not contain u is a subset of L_1 or a subset of L_2. Moreover, each module in L_1 or L_2 has the property that all vertices have the same vertices outside L_1 or L_2 respectively. A prime module of the whole graph that is in

L_1 or L_2 is also a prime module of G restricted to L_1 or L_2 respectively. A degenerated module of the whole graph that appears as a subset of L_1 or L_2 is a subset of a degenerated module of G restricted to L_1 or L_2 respectively (not necessarily the whole degenerated module).

We assume that modular trees T_1 and T_2 of G restricted to L_1 and G restricted to L_2 respectively are computed recursively.

We first find out those overlap-free modules of G restricted to L_1 and L_2 that remain modules of the whole graph. We call a module of L_1 or L_2 *inhomogeneous* if it has descendents in the modular tree with different neighborhoods outside L_1 or L_2 respectively. We find out inhomogeneous modules, i.e. the tree nodes that represent inhomogeneous modules as follows. We assume that the trees T_1 and T_2 are embedded into the plane and the leaves of T_1 and T_2, i.e. the vertices of L_1 and L_2 are enumerated from left to right, say $L_1 = \{x_1, \dots x_k\}$ and $L_2 = \{y_1, \dots y_l\}$. Such enumerations can be computed in $O(\log n)$ time with $O(n/\log n)$ processors using Eulerian cycle techniques on trees (see for example [15, 2]). We assign those (x_i, x_{i+1}) and (y_i, y_{i+1}) that distinguish in the neighborhood in L_2 and L_1 respectively. We determine the least common ancestors s_i and t_i of all assigned pairs (x_i, x_{i+1}) and (y_i, y_{i+1}) in $O(\log n)$ time with $O(n/\log n)$ processors [11]. We assign all tree nodes as inhomogeneous that are ancestors of at least one assigned tree node s_i or t_i in $O(\log n)$ time with $O(n/\log n)$ processors [15].

The second step is to find out new degenerate modules, i.e. homogeneous subsets of inhomogeneous degenerated modules of L_1 and of L_2. We consider vertices of L_1 or L_2 as *equivalent* if they have the same neighbors outside L_1 or L_2 respectively. One can find the equivalence classes by lexicographic sorting in $O(\log n)$ time using $O(n+m)$ processors (to compare the neighborhoods of two vertices with degrees d_1 and d_2, one needs $d_1 + d_2$ processors and constant time on a CRCW-PRAM. To sort n elements, one needs $O(n)$ processors and $O(\log n)$ time [1]). Let A be an equivalence class and X be an inhomogeneous degenerated module and $X_1, \dots, X_k$ be the child modules of X that are subsets of A. If X contains more than one such child module in A then we create a new module X_A with $X_1, \dots, X_k$ as its children, i.e. suppose t has been assigned as inhomogeneous and represents a degenerate module. Suppose t has more than one child that are not assigned as inhomogeneous. For each non inhomogeneous child t' of t, we pick a vertex $v_{t'}$ of G that is a descendent of t' ($v_{t'}$ is a leaf). This can be done in $O(\log n)$ time with $O(n/\log n)$ processors by list ranking along the Eulerian cycle along the double edged tree. Consider children t' and t'' of t as equivalent if $v_{t'}$ and $v_{t''}$ are equivalent. We determine the equivalence classes by sorting (in $O(\log n)$ time with $O(n)$ processors for all t simultaneously). For each equivalence class A, we create a new node with the children of t in A as children.

It remains to find out the modules that contain u. Let X be a module containing u. Note that every neighbor of X that is not in X is in the neighborhood of u. Let L_1 be the set of neighbors of u and L_2 be the set of non neighbors of u. Since all neighbors of X not being in X are in L_1, $X \cap L_2$ is a union of connected components of G restricted to L_2. Note that each module of G is also a module of the complement of G. Therefore for symmetry reasons, $X \cap L_1$ is the union of connected components of the complement of G restricted to L_2.

Shortly we call connected components of G restricted to L_2 *2-components* and connected components of the complement of G restricted to L_1 *1-components.* Note that all 1-components and 2-components are known recursively as modules of G restricted to L_1 and L_2 respectively.

We consider the situation that D_1 is a 2-component not in the module X and D_2 is a 2-component in X. Let C_1 be a 1-component, such that there is an edge joining some vertex in $x \in D_1$ with some vertex $y \in C_1$. Then also C_1 is not in X. Since C_1 is a subset of the neighborhood of u, and therefore in the neighborhood of X, all vertices in X are adjacent to all vertices in C_1. Therefore also all vertices in D_2 are adjacent to all vertices in C_1.

A 1-component C is called *strongly adjacent* to a 2-component D if each vertex of C is joined by an edge with each vertex of D. We call C also a *strong neighbor* of C and vice versa. A 1-component C is called *weakly* adjacent to a 2-component D if there is an edge joining some vertex of C with some vertex of D. We call D also a *weak neighbor* of C and vice versa.

We can reformulate above observation as follows.

Let X be a module containing u, D_1 be a 2-component not in X, and D_2 be a 2-component in X. Then all weak neighbors of D_1 are strong neighbors of D_2.

Next observe that any 1-component C with the property that some 2-component in X is not a strong neighbor of C must be in X and any 1-component that is a weak neighbor of some 2-component not in X must not be in X.

Theorem 1. *Let $u \in X \subseteq V$.*
Then X is a module if and only if

1. *For all 1- and 2-components D, $D \subseteq X$ or $D \cap X = \emptyset$*
2. *For all 2-components $D_1 \not\subseteq X$, $D_2 \subseteq X$, all weak neighbors of D_1 are strong neighbors of D_2,*
3. *all 1-components not in X are strong neighbors of all 2-components in X, and*
4. *all 1-components being weak neighbors of some 2-component not in X are not in X.*

Proof. The second statement follows from the first by previous discussions.

Suppose now, the second condition is satisfied. First we show that all neighbors of X not in X are in L_1. Since $L_1 \cap X$ is a union of connected components of G restricted to L_2, no edge appears between a vertex of $L_2 \cap X$ and a vertex of $L_2 \setminus X$. Since all 1-components being weak neighbors of some 2-component not in X are not in X, there is also no edge between a vertex in $L_2 \setminus X$ and a vertex in $L_1 \cap X$. Therefore no neighbor of X in L_2 exists.

Since every connected component of the complement of G restricted to L_1 is either completely in X or disjoint with X, every vertex in $X \cap L_1$ has all vertices in $L_1 \setminus X$ as a neighbors. Therefore all vertices in $X \cap L_1$ have all the same neighbors outside X.

Since all 1-components not in X are strong neighbors of all 2-components in X, $L_1 \setminus X$ is also the neighborhood outside X of every vertex in $L_2 \cap X$. Q.E.D.

We continue with some remarks on the structure of positively and negatively degenerated modules containing u.

First we consider the case that X is a negatively degenerated overlap free module. Let Y be the child module of X containing u. Then all child modules $Y' \neq Y$ of X are 2-components. The weak neighborhood of any such $Y' \neq Y$ is also its strong neighborhood, and all these child modules $Y' \neq Y$ of X have the same 1-components as weak and strong neighbors. Since X is a negatively degenerated module, there is no edge joining any vertex in a child module $Y' \neq Y$ with some vertex in $X \setminus Y'$. Therefore these 1-components are not in X and therefore not in Y.

Note that Y is not a negatively degenerated module and therefore connected. Therefore each 2-component in Y has at least one 1-component in Y as a neighbor and the number of weak neighbors of any 2-component in Y is larger than the number of weak neighbors of any child module $Y' \neq Y$ of X.

Now let D' be any 2-component not in X. Note that every weak neighbor of D' is a strong neighbor of any 2-component D in X. This is particularly true for the child modules $Y' \neq Y$ of X. Therefore D' has at most as many weak neighbors as any child module $Y' \neq Y$ of X. We assume that the number of weak neighbors of D and of Y' is equal. Then Y' and D have the same weak neighbors, because the weak neighborhood of D' is a subset of the strong neighborhood of Y'. The strong neighborhood of D' must be a proper subset of the strong neighborhood of Y' (which is also the weak neighborhood of Y'). Otherwise $D' \cup X \setminus Y$ and X are overlapping modules. This is a contradiction.

Negatively degenerated overlap free modules X with Y as child module containing u have therefore the following characteristic structure.

1. All 2-components in $X \setminus Y$ have the same 1-components as weak neighbors.
2. the 2-components in $X \setminus Y$ are the child modules of X that are not identical to Y,
3. the weak neighbors of any 2-component in $X \setminus Y$ are also strong neighbors,
4. the weak (and therefore strong) neighbors of any 2-component in $X \setminus Y$ are all not in X
5. the weak (and therefore strong) neighborhood of any 2-component in $X \setminus Y$ is a proper subset of the weak neighborhood of any 2-component in Y
6. for any 2-component D' not in X and any 2-component D in $X \setminus Y$, the weak neighborhood of D' is a subset of the weak (and therefore strong) neighborhood of D and the strong neighborhood of D' is a proper subset of the strong neighborhood of D.

Now let X be a positively degenerated overlap free module and Y be the child module of X containing u.

Note that all child modules $Y' \neq Y$ of X are 1-components. All these child modules $Y' \neq Y$ are strong neighbors of each 2-component in X and have no

2-components not in X as weak neighbors. Moreover, each 1-component C in Y is a non strong neighbor (not even necessarily a weak neighbor) of some 2-component in X. Otherwise $(X \setminus Y) \cup C$ and X would be overlapping module (contradiction). Observe that each 1-component C with no 2-component outside X as weak neighbor and all 2-components in X as strong neighbors is a child module of X. Otherwise X and $(X \cup C) \setminus Y$ would be overlapping modules (contradiction).

Therefore positively degenerated overlap free modules can be characterized as follows.

1. Y and X have the same 2-components
2. each 1-component in Y is a non strong neighbor of some 2-component in Y,
3. the 1-components in $X \setminus Y$ are the child modules of X not identical to Y,
4. all 1-components $X \setminus Y$ are strong neighbors of any 2-component in X and not weak neighbors of any 2-component not in X,
5. each 1-component not in X is a weak neighbor of some 2-component not in X.

We assume that the modules of G restricted to L_1 and G restricted to L_2 are known. Then we also know the 1-components (as positively degenerated modules of L_1 that are maximal submodules of L_1) and the 2-components (as negatively degenerated modules of L_2 that are maximal submodules of L_2).

We get the overlap free modules containing u in several steps. We first sort the 2-components with respect to the number of weak and strong neighbors and we determine the final segments of the sorted sequence "representing" modules, and finally we determine the modular tree.

In detail we proceed as follows.

1. We sort all 2-components with respect to the number of weak neighbors to a sequence $(D_1, \ldots, D_l)$. If the number of weak neighbors of two 2-components is equal then the 2-component with the larger number of strong neighbors is considered as the larger 2-component. This can be done in $O(\log n)$ time with $O(n+m)$ processors (we compute, for each 2-component, the number of its weak and strong neighbors in $O(\log n)$ time with $O(n+m)/\log n)$ processors and sort in $O(\log n)$ time with $O(n)$ processors [1]).
2. For each 1-component C, we determine the minimum index $i = Max(C)$ such that for all $j > i$, D_j is a strong neighbor of C, (i.e. the maximum index of a 2-component that is not a strong neighbor of C) and the minimum $j = Min(C)$ such that D_j is a weak neighbor of C. This can be done, for all C simultaneously, in $O(\log n)$ time with $O(n+m)/\log n$ processors (standard minimum computation).
3. Let $I_i := \bigcup_{j \geq i} D_j$. We determine all I_i with the property that there is a module M_i with $I_i = M_i \cap L_2$. In that case, we say that i *represents a module.* Note that i represents a module iff there is no interval $[Min(C), Max(C)]$ with $Min(C) < i$ and $Max(C) \geq i$. We check whether i represents a module as follows. For each j, we determine the maximum $j' = m(j)$, such that there

is a C with $Min(C) = j$ and $Max(C) = j'$. Let $ma(i)$ be the maximum $Max(C)$ with $Min(C) < i$. Then i represents a module iff $ma(i) < i$. Note that $ma(i) = \max_{j<i} m(j)$. $m(j)$ can be computed in $O(\log n)$ time with $O(n+m)/\log n$ processors (standard minimum computation). $ma(i)$ can be computed in $O(\log n)$ time with $O(n/\log n)$ processors using paràllel prefix computation.

4. We determine the negatively degenerated modules as follows. We compute the sets of maximal intervals I such that
   - each $i \in I$ represents a module and
   - all D_i with $i \in I$ have only strong neighbors as weak neighbors and the same strong neighbors. It is sufficient to check that the number of strong and weak neighbors coincide and that numbers of strong neighbors are equal.
   - $k_I = max\, I + 1$ represents a module (this has has to be tested only in case that I has only one element).

   If these conditions are satisfied then we say that I represents a negatively degenerate module.
   The negatively degenerated module module X_I represented by I consists of all D_i, $i \in I$ and
   $$\bigcup_{j \geq k_I} D_j \cup \{u\} \cup$$
   $$\{x \in L_1 | x \text{ is in the neighborhood of some } D_j,\ j \geq k_I$$
   $$\text{and not in the neighborhood of some } D_j,\ j < k_I\}$$
   as submodules.
   To find out such intervals I, we first find out (in $O(\log n)$ time with $O(n+m)/\log n$ processors) those i representing a module such that D_i has only strong neighbors as weak neighbors. Then we find out those i, such that i and $i+1$ represent modules, D_i and D_{i+1} have only strong neighbors as weak neighbors, and D_i and D_{i+1} have the same number of strong neighbors. This is done in constant time with $O(n)$ processors. This enables us to find the left and right borders of such intervals I in constant time with $O(n)$ processors. To get the intervals, we compute, for each i the maximum left border of such an interval, using parallel prefix computation ($O(\log n)$ time and $O(n/\log n)$ processors). Especially we have, for each right border, its corresponding left border and therefore all the intervals I representing a negatively degenerated module.
5. We compute the positively degenerated modules containing u as follows. For each i representing a module, we determine the set $P_i := \{C : C$ strong neighbor of all D_j, $j \geq i$ and C not neighbor of some D_j, $j < i\} = \{C | Min(C) = i \, and \, Max(C) = i - 1\}$. This can be done in $O(\log n)$ time with $O(n/\log n)$ processors. If $P_i \neq \emptyset$ then there is a positively degenerated module PD_i consisting of all $C \in P_i$ and a module U_i consisting of u, all vertices in some D_j, $j \geq i$, and all vertices in some C with $Max(C) > i$ as child modules. If P_i is not empty then we say that i *represents a positively degenerate module.*

(Note that it can be checked in constant CRCW time with a linear number of processors whether P_i is empty or not). If there are 1-components that are even not weak neighbors of any 2-components then they form together with u a positively degenerate module. In that case, $k+1$ represents a positively degenerate module, where k is the largest index of a 2-component.

6. The prime modules are determined as follows. A prime module is necessarily a module U_i as defined in the last step and is a prime module if it does not appear in some interval I as defined in the step that determines the negatively degenerated modules.
   We say that i *represents a prime module* if i represents a module and does not appear in an interval representing a negatively degenerated module. We can check whether i appears in an interval representing a prime module in logarithmic time, and for all i simultaneously, in the same time bound with a linear number of processors.

We have to complete the modular tree.

We get the following tree nodes that are created by the indices representing modules.

1. The tree nodes s_i, such that i represents a prime module containing u,
2. the tree nodes t_i, such that i represents a positively degenerate module containing u, and
3. the tree nodes u_I, such that I represents a negatively degenerated module containing u.

We determine the parent function *Parent* of the modular tree as follows.

First we consider $Parent(u)$. If $k+1$ represents a positively degenerated module then $Parent(u)$ is set to be t_{k+1}. Otherwise let k' be the largest index that represents a module. We can determine k' by list ranking in $O(\log n)$ time with $O(n)$ processors [2]. If k' represents a prime module then $Parent(u)$ is $s_{k'}$. Otherwise $Parent(u)$ is the u_I such that I contains k'.

Next we consider the parents of s_i, t_i, and u_I. For each i representing a module, we determine the largest $i' < i$, say $P(i)$, that represents a module, by list ranking [2] in $O(\log n)$ time with $O(n/\log n)$ processors.

Suppose i represents a prime module or is the smallest index of an interval I representing a negatively degenerated module. If i represents also a positively degenerated module then $Parent(s_i) := t_i$ or $Parent(u_I) := t_i$ respectively. If i does not represent a positively degenerate module then there are two possible cases.

- $P(i)$ (the next smaller i' representing a module) represents a prime module. Then $Parent(s_i) := s_{P(i)}$ or $Parent(u_I) := s_{P(i)}$.
- $P(i) = i-1$ and i is the largest index of an interval I' representing a negatively degenerate module. Then $Parent(s_i)$ and $Parent(u_I)$ are set to be $u_{I'}$ respectively.

If i represents a positively degenerated module then there are the same two possible cases as mentioned above and $Parent(t_i) = s_{P(i)}$ if $P(i)$ represents a prime module and $Parent(t_i) := u_{I'}$ if $P(i)$ appears in the interval I' representing a negatively degenerate module.

It remains to determine the parents of tree nodes representing modules that are subsets of L_1 or L_2.

We have to determine the parents of homogeneous nodes with an inhomogeneous parent that have not a new node as new parent and the parents of new nodes.

Let t such a node. We first determine (by list ranking in $O(\log n)$ time with $O(n/\log n)$ processors) a leaf descendent v_t of t. Note that v_t is a vertex of L_1 or L_2. We determine the 1- or 2-component C_t or D_t v_t belongs to (in constant time with $O(n)$ processors).

Suppose t represents a module in L_2 and let $D_t = D_i$. Then i_t is the largest index $\leq i$ representing a module . Note that $i_t = P(i+1)$ and that i_t therefore can be determined in constant time. If i_t represents a prime module then $Parent(t)$ is set to be the node s_i representing this prime module. If i_t is in an interval I representing a negatively degenerate module (this I can be determined in constant time) then $Parent(t)$ is set to be u_I if t does not represent a negatively degenerate module. Otherwise t and u_I are contracted to one node. Note that in that case, $i = i_t$.

Suppose t represents a module in L_1. Note that all vertices of L_2 that are leaf descendents of t have the same neighbors in L_2.

First case: $Min(C_t) = Max(C_t) + 1$ and $i = Min(C_t)$ represents a module. Then i also represents a positively degenerated module and C_t belongs to the positively degenerated module t_i. If t itself represents a positively degenerate module then t and t_i are contracted to one node. If t does not represent a positively degenerate (this is only possible if t represents exactly the module C_t) then $Parent(t)$ is set to be t_i. This part can be done in constant time with a linear processor number)

Second case: $Min(C_t) \leq Max(C_t)$ or $Min(C_t)$ does not represent a module. Let i_t be the largest $j \leq Min(C_t)$ representing a module (in constant time by knowledge of $P(i_t + 1)$). $Parent(t)$ is just the prime module s_{i_t}. Note that i_t cannot appear in an interval representing by a negatively degenerate module.

Therefore:

Lemma 2. *If the conditions of the first case are satisfied then one recursion step can be done in logarithmic time with a linear processor bound on a CRCW-PRAM.*

3.2 Second Case: All Vertices are Low Degree Vertices

Here we proceed in a similar way as in the low degree refinement of the perfect elimination ordering algorithm of Klein [8]. We compute a spanning forest F of G (in $O(\log n)$ time with $O(n+m)$ processors [13]). The connected components of G, i.e. the trees of F are modules. If G is not connected then the whole vertex set V forms a negatively degenerated module. For each tree T of F with a root u, we compute a preorder $\{u_1, \dots, u_n\}$ of T. We set $L_1 := \{u\}$ and

$$L_{i+1} := \{x | u_i \text{ is the neighbor of } x$$
$$\text{with minimum index }\}.$$

By a similar argument as in the first case, each module not containing u is a subset of some L_i. To compute the modules in L_i, we compute, as in the first case, the modules of G restricted to L_i and compute, in the same way as in the first case, the modules of G in L_i by comparison of the neighborhood outside L_i.

The modules containing u can be found out in the same way as the modules containing u in the first case.

The processor and time bounds of one recursion step are therefore as in the first case.

3.3 Third Case: All Vertices are of Low or High Degree

We distinguish between *small* modules that contain only $1/4|V|$ vertices and *large* modules that contain more than $1/4|V|$ vertices.

Lemma 3. *If X is a small module then all vertices of X are of low degree or all vertices of X are of high degree.*

Proof. Suppose $|X| < 1/4|V|$, $x, y \in X$, $degree(x) < 1/4|V|$, and $degree(y) > 3/4|V|$. Then x has at most $1/4|V|$ neighbors outside X and y has at least $1/2|V|$ neighbors outside X. Q.E.D.

Lemma 4. *If X is a large module then all neighbors of X that are not in X are of high degree.*

Therefore

Lemma 5. *If X is a large module containing low degree and high degree vertices then the set X' of low degree vertices of X is a union of connected components of G restricted to the set of low degree vertices.*

Corollary 6. *If X is a large module containing low degree and high degree vertices then the set X'' of high degree vertices of X is a union of connected components of the complement of G restricted to the set of high degree vertices.*

First, we compute all modules of the set L of low degree vertices and of the set H of high degree vertices. The modules of L are computed as in the second case. To compute the modules of H, we first compute the complement of G restricted to H, and afterwards, we proceed as in the second case. Note that the number of edges in G restricted to H is smaller than the number of edges adjacent to some vertex in H and therefore the number of edges of the complement of G restricted to H is $O(m)$. By comparison of neighbors outside L or H, we get all modules in L and H that are also modules of G. By previous lemma and corollary, modules having high degree and low degree vertices satisfy the same conditions as modules containing the set u in the first case. In so far, we can determine them in the same way as in the first case.

Again, as in the second and first case, one recursion step can be done in $O(\log n)$ time with $O(n+m)$ processors.

4 Complexity Analysis

In all three cases, the problem of modular decomposition is reduced to components of size at 3/4 of the number of vertices. Therefore the recursion depth is $O(\log n)$. In all three cases, one recursion can be done in $O(\log n)$ time by a CRCW-PRAM with $O(n+m)$ processors. Therefore the overall complexity on a CRCW-PRAM is of $O(n+m)$ processors and $O(\log^2 n)$ time.

Theorem 7. *Modular decomposition can be done by a CRCW-PRAM in $O(\log^2 n)$ time with $O(n+m)$ processors.*

5 Conclusions

The linear time modular decomposition algorithms of [9, 4] is quite complicated. It should not be too difficult to transform the parallel modular decomposition algorithm as presented here to a simpler linear time algorithm. It is almost obvious how to turn the algorithm as presented here into an $O(n+m)\log n$ algorithm. One might also ask whether the techniques can be applied to other structures like directed graphs or graphs with colored edges. We believe that this is possible with not too much effort.

Very related to the problem of modular decomposition is also the problem of transitive orientation of an undirected graph [12]. Further research should be done to extend the algorithm presented in this paper to an efficient parallel algorithm for transitive orientation and permutation graph recognition. Note that permutation graphs are exactly those graphs which are transitively orientable and where the complement is transitive orientable.

The parallel modular decomposition algorithm induces also a parallel cograph recognition algorithm and therefore generalizes a result of X. He [7]. An improved cograph algorithm working in $O(\log^2 n)$ time on a CREW-PRAM with a linear processor number is developed in [5].

References

1. R. Cole, Parallel Merge Sort, *27. IEEE-FOCS* (1986), pp. 511-516.
2. R. Cole, U. Vishkin, Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking, *Information and Control* 70 (1986), pp. 32-53.

3. D. Corneil, H. Lerchs, L. Burlingham, Complement Reducible Graphs, *Discrete Applied Mathematics* 3 (1981), pp. 163-174.
4. A. Cournier, M. Habib, A New Linear Time Algorithm for Modular Decomposition, *Trees in Algebra and Programming*, LNCS 787 (1994), pp. 68-84.
5. E. Dahlhaus, Efficient Parallel Recognition Algorithms of Cographs and Distance Hereditary Graphs, *Discrete Applied Mathematics* 57 (1995), pp. 29-44.
6. A. Ehrenfeucht, H. Gabow, R. McConnell, S. Sullyvan, An $O(n^2)$ Divide-and-Conquer Algorithms for the Prime Tree Decomposition of Two-Structures and Modular Decomposition of Graphs, *Journal of Algorithms*, 16 (1994), pp. 283-294.
7. X. He, Parallel Algorithms for Cograph Recognition with Applications, *Journal of Algorithms* 15 (1993), pp. 284-313.
8. P. Klein, Efficient Parallel Algorithms for Chordal Graphs, 29th IEEE-FOCS (1988), pp. 150-161.
9. R. McConnell, J. Spinrad, Linear-Time Modular Decomposition and Efficient Transitive Orientation of Comparability Graphs, *Fifth Annual ACM-SIAM Symposium of Discrete Algorithms* (1994), pp. 536-545.
10. J.H. Muller, J. Spinrad, Incremental Modular Decomposition, *Journal of the ACM*, 36 (1989), pp. 1-19.
11. B. Schieber, U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM Journal on Computing* 17 (1988), pp. 1253-1262.
12. J. Spinrad, On Comparability and Permutation Graphs, *SIAM-Journal on Computing*, 14 (1985), pp. 658-670.
13. Y. Shiloach, U. Vishkin, An O(log n) Parallel Connectivity Algorithm, *Journal of Algorithms*, 3 (1982), pp. 57-67.
14. J. Spinrad, P4-Trees and Substitution Decomposition, *Discrete Applied Mathematics*, 39 (1992), pp. 263-291.
15. R. Tarjan, U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time, *SIAM Journal on Computing* 14 (1985), pp. 862-874.