Parallel Algorithms for Partial Two-Trees and for Blocks of Partial k-Trees*

Babette de Fluiter and Hans L. Bodlaender

Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: {babette,hansb}@cs.ruu.nl

Abstract. In this paper we present a parallel algorithm which, given a graph G of bounded treewidth (i.e. G is a partial k-tree for some fixed $k \geq 1$), splits G in its blocks (biconnected components). Furthermore, we give a parallel algorithm that decides whether a graph G has treewidth at most two, and if so, constructs a tree decomposition of minimum width of G. Both algorithms use $O(n)$ operations and $O(\log n \log^* n)$ time on an EREW PRAM, and $O(\log n)$ time on a CRCW PRAM. The best previously known parallel algorithm for finding a tree decomposition of width two uses $O(n)$ operations and $O(\log^2 n)$ time on an EREW PRAM. The best previously know parallel algorithm for finding the biconnected components of a graph with bounded treewidth uses, on a CRCW PRAM, the same amount of resources as our algorithm, but it only returns a labeling of the edges of the graph.

1 Introduction

In this paper, we consider two problems, namely the problem of finding the biconnected components (or blocks) of a graph with bounded treewidth, and the problem of finding a tree decomposition of width at most two of a graph, if one exists. The problem of finding the blocks of a graph is well studied. In this paper, we focus on parallel algorithms for this problem. Different parallel algorithms for the problem can be distinguished by the way in which the blocks are represented. Many algorithms give as output a labeling of the edges of the graphs, such that two edges belong to the same block if and only if they have the same label (we call such a labeling a *block labeling*). However, when one wants to run a successive parallel algorithm on the separate blocks of the graph, such a block labeling may not be sufficient. The algorithm, described in this paper returns a list of subgraphs of the input graph (assumed to have bounded treewidth), which form the blocks of the graph. Thus, this algorithm enables us to run other parallel algorithms on the blocks separately; we do not know a parallel algorithm with the same resource bounds, that computes the blocks of a graph, given a block labeling of the edges.

For arbitrary graphs, the best known parallel algorithm is from Cole and Vishkin [9]. It uses $O(n \log n + m)$ operations and $O(\log n)$ time on a CRCW PRAM, where n denotes the number of vertices, and m the number of edges of the input graph (the number

* This research was partially supported by the Foundation for Computer Science (S.I.O.N) of the Netherlands Organization for Scientific Research (N.W.O.).

of operations that an algorithm uses is the product of the number of processors and the time it uses). This algorithm returns a block labeling of the edges. For sufficiently dense graphs, this algorithm is optimal, i.e. the number of operations equals the number of operations in the best known sequential algorithm.

Hagerup [13] has given an optimal parallel algorithm for finding the blocks of a graph from certain classes of sparse graphs, namely the classes of *linearly contractible graphs*. This algorithm uses $O(n)$ operations, and $O(\log n)$ time on a CRCW PRAM. It returns a block labeling of the edges.

For each fixed k, the class of graphs with treewidth at most k is a class of linearly contractible graphs. In this paper, we give, for each fixed $k \geq 1$, an algorithm for finding the blocks of a graph G, if it has treewidth at most k. Our algorithm returns a list of subgraphs of the input graph, which form the blocks of the graph. It uses $O(n)$ operations, $O(\log n)$ time on a CRCW PRAM, and $O(\log n \log^* n)$ time on an EREW PRAM.

In [14], Hagerup gives a parallel algorithm with the same resource bounds on a CRCW PRAM, which returns a graph of which the connected components are exactly the blocks of the input graph. This algorithm works for input graphs from the class of planar graphs, but it needs a planar embedding of the graph as input as well. Graphs with treewidth at most two are planar graphs, but there is no parallel algorithm known which finds a planar embedding for a graph with treewidth at most two, and which uses $O(n)$ operations and $O(\log n)$ time on a CRCW PRAM or $O(\log n \log^* n)$ time on an EREW PRAM.

Many important graph classes have bounded treewidth. Given a tree decomposition of bounded width of a graph, many (even NP-hard) problems can be solved sequentially in linear time, and in parallel with $O(n)$ operations and $O(\log n)$ time on an EREW PRAM (see e.g. [2, 4, 7]). Therefore, the problem of finding a tree decomposition of bounded width of a graph is well studied. Bodlaender [3] has given a linear time algorithm that, for each fixed $k \geq 1$, recognizes graphs of treewidth at most k and builds a tree decomposition of minimum width for the graph, if it has treewidth at most k.

The best known parallel algorithm for recognizing graphs with treewidth at most k is from Bodlaender and Hagerup [7]. It uses $O(n)$ operations, $O(\log n)$ time on a CRCW PRAM or $O(\log n \log^* n)$ on an EREW PRAM. They also give a parallel algorithm for building a tree decomposition of width at most k, which uses $O(n)$ operations, and $O(\log^2 n)$ time on an EREW PRAM. Related, earlier results can be found e.g. in [12, 15].

In this paper, we give an algorithm that constructs a tree decomposition of minimum width of a graph, if the graph has treewidth at most two. The algorithm uses $O(n)$ operations, $O(\log n)$ time on a CRCW PRAM and $O(\log n \log^* n)$ time on an EREW PRAM. Therefore, this algorithm is faster than the algorithm of [7] for treewidth two.

Our algorithm for finding blocks is an application of a result from [5]. This result states that, if a graph problem satisfies certain conditions, then, for each graph with small treewidth, we can find a solution for this graph problem with a parallel algorithm that uses $O(n)$ operations, using $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ time on an EREW PRAM. A central technique for this result is the technique of *graph reduction*, which is introduced in [1], and adapted for parallel applications in [7].

Our algorithm for finding a tree decomposition of width two of a given graph makes use of the relation between the blocks of graphs of treewidth at most two, and series parallel graphs. Bodlaender and de Fluiter [6] give a parallel algorithm for recognizing series parallel graphs with $O(n)$ operations and $O(\log n)$ time on a CRCW PRAM or $O(\log n \log^* n)$ on an EREW PRAM. We use this algorithm for finding a tree decomposition of width two of the blocks of the input graph. Therefore, we use our block algorithm to find the blocks of the input graph.

This paper is organized as follows. In Sect. 2, we give a number of definitions and well known results. In Sect. 3 we describe our algorithm for finding the blocks of a graph with bounded treewidth, and in Sect. 4 we give our algorithm for finding a tree decomposition of minimum width of a graph, if it has treewidth at most two.

2 Definitions

The graphs we consider are undirected, and do not contain self-loops or multiple edges, unless stated otherwise. We say a property is *effectively decidable* if an algorithm is *known* that decides on the property. Similarly, we say a function is *effectively computable* if an algorithm is known that computes the function value for a given element of the domain.

A *tree decomposition* of a graph $G = (V, E)$ is a pair $(\mathcal{X}, T)$ with $T = (I, F)$ a tree, and $\mathcal{X} = \{X_i \mid i \in I\}$ a family of subsets of V, one for each node of T, such that

- $\bigcup_{i \in I} X_i = V$,
- for all edges $\{v, w\} \in E$, there exists an $i \in I$ with $v \in X_i$ and $w \in X_i$, and
- for all $i, j, k \in I$: if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *treewidth* or *width* of a tree decomposition $(\{X_i \mid i \in I\}, (I, F))$ is defined as $\max_{i \in I} |X_i| - 1$. The treewidth $\mathrm{tw}(G)$ of a graph G is the minimum treewidth over all tree decompositions of G [16].

Let P be a graph property, i.e. for each graph G, $P(G) \in \{\mathrm{true}, \mathrm{false}\}$. For each integer $k \geq 1$, the property P_k is defined as $P_k(G) = P(G) \wedge (\mathrm{tw}(G) \leq k)$.

A *series parallel graph* is a triple (G, s, t), with $G = (V, E)$ a graph, possibly with multiple edges, and $s, t \in V$, such that one of the following cases holds.

- G has two vertices, s and t, and one edge between s and t.
- There are two series parallel graphs (G_1, s_1, t_1) and (G_2, s_2, t_2), such that (G, s, t) is obtained by a *parallel composition* of these graphs, i.e. G is obtained by first taking the disjoint union of G_1 and G_2, and then identifying vertices s_1 and s_2 to s, and vertices t_1 and t_2 to t.
- There are two series parallel graphs (G_1, s_1, t_1) and (G_2, s_2, t_2), such that (G, s, t) is obtained by a *series composition* of these graphs, i.e. G is obtained by first taking the disjoint union of G_1 and G_2, then identifying vertex t_1 with vertex s_2, and letting $s = s_1$ and $t = t_2$.

If (G, s, t) is a series parallel graph, we also say that G is a series parallel graph. Vertex s is called the source of G, and t the sink of G.

An *sp-tree* of a series parallel graph G is a rooted tree $T = (N, F)$ for which the following holds. Every node of T corresponds to a series parallel graph (G', s', t'), and is labeled with the pair (s', t'), the root corresponds to the graph (G, s, t). Leaf nodes correspond to series parallel graphs consisting of a single edge: there is a one-to-one correspondence between leaves of T_G and edges $e \in E$. Leaf nodes are also called edge nodes (e-nodes). Internal nodes are of two types: series nodes (s-nodes), and parallel nodes (or p-nodes), each having two children. The children of a series node are ordered, while the children of a parallel node are not ordered. The series parallel graph associated to an s-node is the graph, obtained by a series composition of the series parallel graphs, associated to the children of the node, where the order of the children gives the order in which the series composition is done. The series parallel graph associated to a p-node is the graph, obtained by a parallel composition of the series parallel graphs, associated to the children of G.

A *terminal graph* G is a triple (V, E, X) with (V, E) a graph, and $X \subseteq V$ an ordered subset of the vertices ($|X| \geq 0$), denoted by $\langle x_1, \ldots, x_l \rangle$, and called the set of *terminals*. For each i, x_i is called the ith terminal of G. Vertices in $V - X$ are called *inner vertices*. A terminal graph (V, E, X) is called an l-terminal graph if $|X| = l$.

The operation $\oplus$ maps two terminal graphs G and H with the same number l of terminals to a graph $G \oplus H$, by taking the disjoint union of G and H, and then for $i = 1, \ldots, l$, identifying the ith terminal of G with the ith terminal of H, and removing multiple edges.

Let G be a (terminal) graph. G is *biconnected* if $|V(G)| \geq 2$ and for each $v \in V(G)$, the graph $G[V - \{v\}]$ is connected. A *biconnected component* or *block* of a graph G is a maximal biconnected subgraph of G. The blocks of G are induced subgraphs of G, and they partition the set of edges $E(G)$. Two distinct blocks of a graph have at most one vertex in common. The vertices of G which are in more than one block are called the *cut-vertices* of G.

Let $\mathcal{B}$ be the set of blocks of G, and C the set of cut-vertices of G. The graph T_G is defined as follows. $V(T_G) = \mathcal{B} \cup C$ and $E(T_G) = \{\{c, B\} \mid c \in C \wedge B \in \mathcal{B} \wedge c \in V(B)\}$. It can be seen that T_G is a forest of which the connected components correspond to the connected components of G. T_G is called the *block-forest* of a graph G.

Two terminal graphs $(V_1, E_1, \langle x_1, \cdots, x_k \rangle)$ and $(V_2, E_2, \langle y_1, \cdots, y_l \rangle)$ are said to be *isomorphic*, if $k = l$ and there exists a bijective function $f : V_1 \to V_2$ with for all $v, w \in V_1$, $\{v, w\} \in E_1 \Leftrightarrow \{f(v), f(w)\} \in E_2$ and for all i, $1 \leq i \leq k$, $f(x_i) = y_i$.

A *reduction rule* r is an ordered pair (H_1, H_2), with H_1 and H_2 l-terminal graphs for some $l \geq 0$. An *application* of reduction rule $r = (H_1, H_2)$ is the operation that replaces a graph G of the form $G_1 \oplus G_3$, with G_1 isomorphic to H_1, by a graph $G_2 \oplus G_3$, with G_2 isomorphic to H_2. We write $G \xrightarrow{r} G_2 \oplus G_3$. For two graphs G and G', and a set of reduction rules $\mathcal{R}$, we write $G \xrightarrow{\mathcal{R}} G'$, if there exists an $r \in \mathcal{R}$ with $G \xrightarrow{r} G'$.

Let P be a graph property and $\mathcal{R}$ a set of reduction rules. $\mathcal{R}$ is *safe* for P if, whenever $G \xrightarrow{\mathcal{R}} G'$, then $P(G) \Leftrightarrow P(G')$. $\mathcal{R}$ is *complete* for P if the set of graphs $F_{\mathcal{R}} = \{G \mid P(G) \wedge \neg\exists G' : G \xrightarrow{\mathcal{R}} G'\}$ is finite. $\mathcal{R}$ is *decreasing* if, whenever $G \xrightarrow{\mathcal{R}} G'$, then G' contains fewer vertices than G.

A set of reduction rules $\mathcal{R}$ that is finite, safe, complete, and decreasing for a property P corresponds to an algorithm that decides whether property P holds on a given graph G: repeat applying rules from $\mathcal{R}$ starting with G, until no rule from $\mathcal{R}$ can be applied anymore. $P(G)$ holds if and only if the resulting graph is in $F_{\mathcal{R}}$. In [1] it has been shown how this algorithm can be implemented such that it takes linear time and polynomial space.

A set of applications of reduction rules is said to be *concurrent* if there is no inner vertex of any subgraph to be rewritten which also occurs in another subgraph to be rewritten. The idea behind concurrent applications of rules is that in a parallel algorithms, all reduction steps from a concurrent set can be applied in parallel.

More background information about (parallel) graph reduction and graphs of bounded treewidth can be found in [4, 5, 7, 10].

3 Finding the Blocks

In this section, we give algorithms that compute the blocks of a given graph if the graph has treewidth at most k ($k \geq 1$ fixed). The algorithms return the following.

1. A list $G_1, \ldots, G_s$ of graphs which are the blocks of the input graph.
2. For each vertex $v \in V(G)$ a list of blocks in which it occurs.

Our block algorithm is an application of a theorem in [5]. We first briefly resume this result and necessary definitions.

Let P denote a graph property of the form $P(G) = \exists_{S \in D(G)} Q(G, S)$, where $D(G)$ is a *solution domain* (or shortly domain), which is some set depending on G, and Q is a property of G and S, i.e. $Q(G, S) \in \{\text{true}, \text{false}\}$ for all graphs G and all $S \in D(G)$. An $S \in D(G)$ for which $Q(G, S)$ holds is called a *solution* for G. For example, for the perfect matching problem on a graph G, $D(G)$ can be the power set of E, and for $S \in D(G)$, $Q(G, S)$ holds if and only if every vertex in G is end point of exactly one edge in S. Hence S is a solution for G if S is a perfect matching of G. For the block problem, we can define $D(G)$ to be the singleton set which contains only the set of blocks of G, and $Q(G, S)$ holds for all G and $S \in D(G)$.

Let D be some solution domain, i.e. for each graph G, $D(G)$ denotes a 'set of all possible solutions' in G. Let G and H be l-terminal graphs and let $S \in D(G \oplus H)$. $S[G]$ denotes the restriction of S to G. $S[G]$ is *properly defined* if it contains no vertices or edges which are not in G. For example, if S is a set of edges from $E(G \oplus H)$, then a proper definition of $S[G]$ would be $S \cap E(G)$.

Let a proper definition of $[\,]$ be given. For each $l \geq 0$, and each l-terminal graph G, define $D_{[\,]}(G) = \{S[G] \mid S \in D(G \oplus H) \text{ for some } l\text{-terminal graph } H\}$.$D_{[\,]}(G)$ is called the domain of *partial solutions* of G. Note that $D(G) \subseteq D_{[\,]}(G)$. Domain D is *inducible* for $[\,]$ if for each two l-terminal graphs G and H, and $S_G \in D_{[\,]}(G)$, $S_H \in D_{[\,]}(H)$, there is at most one $S \in D(G \oplus H)$ such that $S[G] = S_G$ and $S[H] = S_H$. If there is exactly one $S \in D(G \oplus H)$ such that $S[G] = S_G$ and $S[H] = S_H$, then (G, S_G) and (H, S_H) are called $\oplus$-*compatible*, and we write $S_G \oplus S_H = S$.

Let P be a graph property, and suppose $P(G)$ can be written as $\exists_{S \in D(G)} Q(G, S)$, such that domain D is inducible for $[\,]$. For each $l \geq 0$, $\sim_{Q,l}$ is an equivalence relation on pairs of l-terminal graphs G and partial solutions $S \in D_{[\,]}(G)$, which is defined as follows. Let G_1, G_2 be l-terminal graphs, and $S_1 \in D_{[\,]}(G_1)$ and $S_2 \in D_{[\,]}(G_2)$. $(G_1, S_1) \sim_{Q,l} (G_2, S_2)$ if and only if

- for all l-terminal graphs H and all $S \in D_{[\,]}(H)$, (G_1, S_1) and (H, S) are $\oplus$-compatible if and only if (G_2, S_2) and (H, S) are $\oplus$-compatible, and
- for all l-terminal graphs H and all $S \in D_{[\,]}(H)$, if (G_1, S_1) and (H, S) are $\oplus$-compatible then $Q(G_1 \oplus H, S_1 \oplus S) \Leftrightarrow Q(G_2 \oplus H, S_2 \oplus S)$.

By $\sim_{rQ,l}$, we usually denote an equivalence relation which is a *refinement* of $\sim_{Q,l}$, i.e. each equivalence class of $\sim_{rQ,l}$ is a subset of an equivalence class of $\sim_{Q,l}$. By $ec_{rQ,l}(G,S)$ we denote the equivalence class of $\sim_{rQ,l}$ which contains (G,S). The following theorem is from [5].

Theorem 1 [5]. *Suppose P is a graph property which can be written in the form*
$P(G) = \exists_{S \in D(G)} Q(G,S)$,
and there is a proper definition of $[\,]$ *such that the following conditions hold.*

1. *Domain D is inducible for $[\,]$, Q is effectively decidable, a refinement $\sim_{rQ,l}$ of $\sim_{Q,l}$ is effectively decidable and has finitly many equivalence classes, for fixed $l \geq 0$.*
2. *There is a representation of (partial) solutions for which the following holds.*
   (a) *For each terminal graph G, each $S \in D_{[\,]}(G)$, the amount of bits needed to represent S is at most some finite known number.*
   (b) *For each two fixed l-terminal graphs H and H', the following holds. For each l-terminal graph G, if $S \in D(G \oplus H)$, then $S[H]$ can be computed from S and H in constant time, and for each $S' \in D_{[\,]}(H')$, if $(H, S[H]) \sim_{rQ,l} (H', S')$, then $S' \oplus S[G]$ can be computed in constant time from S, S', H and H'.*

Then for each $k \geq 1$, we have a parallel algorithm which computes for a given graph G an $S \in D(G)$ such that $Q(G,S)$ holds, if $P_k(G)$ holds. The algorithm uses $O(n)$ operations and space, and $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ on an EREW PRAM.

For each $k \geq 1$, the algorithm is based on a finite, safe, complete and decreasing set $\mathcal{R}$ of reduction rules for P_k. If a graph G has treewidth at most k, then there is a set of $\Omega(n)$ concurrent applications of rules from $\mathcal{R}$ on G. Furthermore, for each rule $r = (H, H') \in \mathcal{R}$, $\{ec_l(H,S) \mid S \in D_{[\,]}(H)\} = \{ec_l(H',S') \mid S' \in D_{[\,]}(H')\}$.

In the remainder of this section, we show that the conditions of Theorem 1 hold for the problem of finding the blocks of a graph. Thus, for each $k \geq 1$, we can construct a finite, safe, complete and decreasing set $\mathcal{R}$ of reduction rules for which the above conditions hold. The algorithm based on $\mathcal{R}$ then roughly looks as follows, for input graph G (for more details, see [5, 7]).

1. First, do a number of reduction rounds. In each round, $\Omega(n)$ concurrent applications of reduction rules from $\mathcal{R}$ are done on the graph G, if possible. If no $\Omega(n)$ concurrent applications are possible but G is not 'small', then G does not have treewidth at most k, and hence the input graph has treewidth more than k. (n denotes the number of vertices of the current version of G.)
2. For the reduced graph G, construct the set S of blocks of G (the initial solution).
3. Undo the reduction rounds in reversed order: in each undo-action of a reduction round, do the following.
   (a) Undo each application of a reduction rule from this round, in parallel.
   (b) Transform the set S of blocks of the old graph into the set of blocks of the new graph: for each subgraph G_2 of the old graph which is replaced by a subgraph G_3 in undoing a reduction rule, replace, in the blocks S of the old graph, $S[G_2]$ by an $S_3 \in D_{[\,]}(G_3)$ for which $ec_{rQ,l}(G_2, S[G_2]) = ec_{rQ,l}(G_3, S_3)$.

Condition 1 of Theorem 1 assures the existence of the set of reduction rules described above, and that this set can be constructed effectively. Furthermore, it assures that in step 3(b) of the algorithm, $ec_{rQ,l}(G_2, S[G_2])$ can be effectively computed from G_2 and $S[G_2]$.

Condition 2 assures that the initial solution can be found quickly, and that, in step 3(b) of the algorithm, $S[G_2]$ can be computed in constant time, and can be replaced by S_3 in constant time.

The number of reduction rounds is $O(\log n)$, because in each reduction round, the number of vertices of the graph is reduced by a constant factor. Each reduction round can be done with $O(n)$ operations, in $O(1)$ time on a CRCW PRAM and $O(\log^* n)$ time on an EREW PRAM: sufficiently many processors can find an applicable reduction rule in $O(1)$ time, if there are enough rules, but finding a set of $\Omega(n)$ concurrent reduction rules out of this total set takes $O(\log^* n)$ time on an EREW PRAM, and $O(1)$ time on a CRCW PRAM (see [7] for details). Each processor applies at most one application of a reduction rule in $O(1)$ time.

In step 3, each application of a reduction rule is undone in $O(1)$ time by the same processor that applied it in step 1. This processor also computes $ec_l(G_2, S[G_2])$ and S_3, and replaces $S[G_2]$ by S_3 in $O(1)$ time.

We now show that the conditions of Theorem 1 hold for the block problem. For each graph G, let $P(G) = \exists_{S \in D(G)}\, Q(G, S)$, where $D(G)$ is the singleton set containing the set $\{G_1, \ldots, G_s\}$ of blocks of G. For each graph G, each $S \in D(G)$, $Q(G, S) = \mathrm{true}$.

For each two terminal graphs G and H, each $S = \{G_1, \ldots, G_s\} \in D(G \oplus H)$, let

$$S[G] = \{G[E_i \cap E(G)] \mid 1 \leq i \leq s \wedge |E_i \cap E(G)| \geq 1\},$$

where $G[E_i \cap E(G)]$ denotes the subgraph of G which is induced by the end points of all edges in $E_i \cap E(G)$. This is a proper definition of $[\,]$. Note that if $|E_i \cap E(G)| \geq 1$ and $|E_i \cap E(H)| \geq 1$, then $G' = G[E_i \cap E(G)]$ and $H' = H[E_i \cap E(H)]$ have at least two terminals in common: if they have no terminals in common, then there is no edge between any vertex of G' and a vertex of H', so G_i is not connected. If they have one terminal v in common, then all edges between vertices in G' and H' are incident with v, so $G_i[V_i - \{v\}]$ is not connected, and hence G_i is not biconnected.

For a given terminal graph G and $S \in D_{[\,]}(G)$, we call the graphs in S *partial blocks*. Vertices $v \in V(G)$ for which there are two distinct partial blocks in S which contain v are called *partial cut-vertices*.

Proof of Condition 1

Domain D is inducible for $[\,]$, since $D(G)$ contains only one element. For two l-terminal graphs $G = (V, E, X)$ and $H = (V', E', X')$ and $S \in D_{[\,]}(G)$, $S' \in D_{[\,]}(H)$, if (G, S) and (H, S') are $\oplus$-compatible, then by the definition of $[\,]$,

$$\begin{aligned} S \oplus S' = {} & \{G' \in S \mid |V(G') \cap X| \leq 1\} \cup \{G' \in S' \mid |V(G') \cap X'| \leq 1\} \\ & \cup \{G' \cup H' \mid G' \in S \wedge H' \in S' \wedge |V(G') \cap V(H')| \geq 2\} \end{aligned}$$

It is clear that Q is effectively decidable.

Lemma 2. *Domain D is inducible for $[\,]$, and Q is effectively decidable.*

We now have to give an equivalence relation $\sim_{rQ,l}$ for each $l \geq 0$, which is a refinement of $\sim_{Q,l}$, which has a finite number of equivalence classes (fixed l), and which is effectively decidable.

For each l-terminal graph G and each $S \in D_{[\,]}(G)$, let $ec_{rQ,l}(G,S) = (F_1(G,S), F_2(G,S), F_3(G,S), F_4(G,S))$, where $F_1, \ldots, F_4$ are as defined below.

For each l-terminal graph $G = (V, E, X = \langle x_1, \ldots, x_l \rangle)$ and $S \in D_{[\,]}(G)$, let

$$F_1(G,S) = \{\{i,j\} \mid 1 \leq i < j \leq l \wedge \{x_i, x_j\} \in E\},$$
$$F_2(G,S) = \{I \subseteq \{1,\ldots,l\} \mid \exists_{G' \in S}\, V(G') \cap X = \{x_i \mid i \in I\} \wedge |I| \geq 2\},$$

i.e. $F_1(G,S)$ represents the set of edges of G between two terminals of G, and $F_2(G,S)$ represents each partial block in S which contains two or more terminals, by the terminals it contains. Note that these sets exactly represent the partial blocks in S which can be extended with vertices from an l-terminal graph H in $G \oplus H$.

Definition 3. Let G be an l-terminal graph, let $S = \{G_1, \ldots, G_s\} \in D_{[\,]}(G)$. Let $T_{G,S}$ be the forest defined as follows. $V(T_G) = S \cup C$, where C is the set of partial cut-vertices. $E(T_G) = \{\{G_i, c\} \mid G_i \in S \wedge c \in C \wedge c \in V(G_i)\}$. $T_{G,S}$ is called the *partial block-forest* of G and S.

For given l-terminal graph $G = (V, E, X)$ and $S \in D_{[\,]}(G)$ let $T^{\mathrm{red}}_{G,S}$ be the forest obtained from $T_{G,S}$ as follows. First, we give each node a label. A node in $T_{G,S}$ which represents a partial cut-vertex v gets label $\mathtt{nt}$ (from non-terminal) if v is an inner vertex, otherwise, it gets label i if v is the ith terminal. A node which represents a partial block $G' \in S$ gets label $\mathtt{nt}$ if it contains at most one terminal, otherwise it gets label $\{i \mid x_i \in V(G')\}$.

Next, delete all nodes from $T_{G,S}$ which have degree at most one and have label $\mathtt{nt}$, and also delete their incident edges, until there are no nodes of degree one with label $\mathtt{nt}$ left. Now for each pair n, m of adjacent nodes which both have label $\mathtt{nt}$, replace n, m and the edge $\{n, m\}$ by a new node p with label $\mathtt{nt}$ which is adjacent to all neighbors of n and m. Repeat these deletions until no pairs of adjacent nodes with label $\mathtt{nt}$ remain.

Let $F_3(G,S) = T^{\mathrm{red}}_{G,S}$.

The property that a graph is biconnected can be expressed in monadic second order logic (see e.g. [8, 11]). This means that, for each $l \geq 0$, there is an equivalence relation $\sim_l$ on l-terminal graphs for which the following holds.

1. For each two l-terminal graphs G and G', if $G \sim_l G'$, then for each l-terminal graph H, $G \oplus H$ is biconnected if and only if $G' \oplus H$ is biconnected.
2. The number of equivalence classes depends only on l.
3. For each l-terminal graph G, $ec_l(G)$ (the equivalence class containing G) can be effectively computed.

For each l-terminal graph G, each $S \in D_{[\,]}(G)$, let

$$F_4(G,S) = \{\{I, c\} \mid \exists_{G' \in S}\, (V(G') \cap X) = \{x_i \mid i \in I\} \wedge |I| \geq 2 \wedge c = ec_l(G')\}$$

Lemma 4. *$\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$, is effectively decidable, and has a finite number of equivalence classes for each fixed $l \geq 0$.*

Proof. We only give a sketch of the proof. It is clear that $\sim_{rQ,l}$ is effectively decidable: F_i is effectively computable for each i, $1 \leq i \leq 4$, and hence $ec_{rQ,l}$ is effectively computable.

The fact that $ec_{rQ,l}$ only has a finite number of equivalence classes follows from the fact that each F_i has a finite range (for fixed l). For F_1, F_2 and F_4 this follows directly from the definition. For F_3 we have to show that the number of different trees $T^{\mathrm{red}}_{G,S}$ for different values of G and S depends only on l. This is because each node in the tree which has degree one or less does not have label $\mathtt{nt}$, there are no two adjacent nodes which both have label $\mathtt{nt}$, each node which does not have label $\mathtt{nt}$ has a unique label, and there are at most $2^l + 1$ different possible labels.

Finally, we have to prove that $\sim_{rQ,l}$ is a refinement of $\sim_{Q,l}$. Therefore, we have to prove for all pairs of l-terminal graphs $G = (V, E, X)$ and $G' = (V', E', X')$, if $(G, S) \sim_{rQ,l} (G', S')$, then for all l-terminal graphs H and all $S_H \in D_{[\,]}(H)$, (G, S) and (H, S_H) are $\oplus$-compatible if and only if (G', S') and (H, S) are $\oplus$-compatible. Suppose $(G, S) \sim_{rQ,l} (G', S')$. Let $H = (V_H, E_H, Y_H)$ and $S_H \in D_{[\,]}(H)$, and suppose there is an $S_{G,H} \in D(G \oplus H)$ such that $S_{G,H}[G] = S$ and $S_{G,H}[H] = S_H$. It suffices to show that there is an $S_{G',H} \in D(G' \oplus H)$ such that $S_{G',H}[G'] = S'$ and $S_{G',H}[H] = S_H$. Let $G_1, \ldots, G_s$ be subgraphs of $G' \oplus H$, such that for each i, $1 \leq i \leq s$, either

- there is a $G'' \in S'$ such that $G_i = G''$ and $|V(G'') \cap X'| \leq 1$, or
- there is an $H' \in S_H$ such that $G_i = H'$ and $|V(H') \cap Y| \leq 1$, or
- there are $G'' \in S'$ and $H' \in S_H$ such that G'' and H' have at least two terminals in common, and $G_i = G'' \cup H'$.

This is a proper definition, because $F_2(G, S) = F_2(G', S')$. It can be seen that each G_i is biconnected: each $G'' \in S'$ and each $H' \in S_H$ are biconnected, and each G_i formed by taking the union of a $G'' \in S'$ and an $H' \in S_H$ is biconnected because $F_4(G, S) = F_4(G', S')$. The fact that $F_3(G, S) = F_3(G', S')$ implies that there are no two vertices v and w in $G' \oplus H$ which are in different G_i's but which are biconnected. This means that $G_1, \ldots, G_s$ are the blocks of $G' \oplus H$, and hence $\{G_1, \ldots, G_s\} = S[G'] \oplus S[H]$. By symmetry, this ends the proof. □

Proof of Condition 2

Let G be a (terminal) graph, let $S = \{G_1, \ldots, G_s\} \in D_{[\,]}(G)$. $G, G_1, G_2, \ldots, G_s$ are each represented as an array of vertices, where each vertex v has a pointer to its adjacency list, and each vertex w in this adjacency list has a pointer to vertex v in the adjacency list of w. S is represented as a list of $G_1, \ldots, G_s$. Additionally, each vertex $v \in V(G)$ gets a pointer to the list of all G_i's which contain v. For each edge $\{v, w\} \in E$, w in the adjacency list of v in G has a pointer to w in the adjacency list of v in G_i (if $\{v, w\} \in E(G_i)$), and vice versa.

This, and the definition of $D_{[\,]}(G)$ imply that condition 2(a) is satisfied: for some constant C, $C \cdot |V(G) + E(G)|$ bits is enough to represent any $S \in D_{[\,]}(G)$.

Consider condition 2(b). Let $l \geq 0$ and let H and H' be fixed l-terminal graphs. Let G be an l-terminal graph, let $S = \{G_1, \ldots, G_s\} \in D(G \oplus H)$. To compute $S[H]$ from S and H, do the following. Check for each edge in H in which block it occurs. Make one partial block H_j for each block G_i of S which contains an edge of H, keep pointers between H_j and G_i. Then put each vertex $v \in V(H)$ in each partial block which contains one of its incident edges, and put each edge in the partial block that contains it. Add pointers where they have to be added. This can all be done in constant time on one processor.

Now we show how $S[G] \oplus S'$ can be computed, for an $S' \in D_{[]}(H')$ with $(H, S[H]) \sim_{rQ,l} (H', S')$. Remove all inner vertices of H from all places in S where it occurs. Similarly, remove all edges of H which contain at most one terminal from all places in S where it occurs. For each partial block in S' which contains at most one terminal, add this partial block as a new block to S. For each partial block H_j in S' which contains two or more terminals, check in which block G_i of S at least two of these terminals all occur (this is at most one block), and add H_j to G_i. Checking which block G_i contains at least two terminals of H_j can be done as follows. Check which partial block in $S[H]$ contains the same set of terminals as H_j. Then check to which block in S this partial block of $S[H]$ corresponds, by following its pointer. This gives the desired block G_i. All steps can again be done in constant time on one processor.

We conclude with the main result of this section.

Theorem 5. *There is a parallel algorithm which, given a graph G, returns, the blocks of the input graph as a list $G_1, \ldots, G_s$, and for each vertex $v \in V(G)$ a list of blocks in which v occurs.*

The algorithm uses $O(n)$ operations and space, and $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ on an EREW PRAM.

4 Finding a Tree Decomposition of Width at Most Two

Our algorithm for finding a tree decomposition of width at most two of a graph uses an algorithm for finding the sp-tree of a graph of [6]. Therefore, we need two lemmas.

The following lemma has been proven in e.g. [6]. We give a constructive proof, which is of use later in this section.

Lemma 6. *If G is a series parallel graph, then the treewidth of G is at most two.*

Proof. Suppose $T = (N, F)$ is an sp-tree for G. For each p- or e-node i with label (v, w), write $X_i = \{v, w\}$; for each s-node with label (v, w) and labels of its two children (v, x) and (x, w), write $X_i = \{v, w, x\}$. Now, one can verify that $(\{X_i \mid i \in N\}, T_G)$ is a tree-decomposition of G of treewidth at most two. □

Lemma 7. *A graph G has treewidth at most two if and only if each block of G is a series parallel graph.*

Proof. Suppose G has treewidth at most two. Let G' be a block of G (G' has treewidth at most two). We show by induction that G' is series parallel.

If $|V(G')| \leq 3$, then it clearly holds. Suppose $|V(G')| > 3$. Let $TD = (\{X_i \mid i \in I\}, T = (I, F))$ be a tree decomposition of width two of G'. Let TD' be obtained from TD by repeating the following as often as possible. For each $i \in I$, if i has exactly one neighbor $j \in I$, and $X_i \subseteq X_j$, then remove X_i. Note that TD' is a tree decomposition of width two of G', and it has at least two nodes. Let $i \in I$ such that i has exactly one neighbor $j \in I$ in T. There is a $v \in X_i$ such that $v \notin X_j$.

Let $v \in X_i$ such that $v \notin X_j$. v must have degree two in G'. Let $G'' = (V(G') - \{v\}, E(G') \cup \{u, w\})$. Then G'' has treewidth two, and is biconnected. By induction hypothesis, G'' is a series parallel graph. Now take any sp-tree T of G''. We transform it into an sp-tree of G', which proves that G' is series parallel. Suppose w.l.o.g. that T has an e-node n with label (u, w). If $\{u, w\} \notin E(G')$, then replace n by an s-node with label (u, w) with as children two e-nodes with labels (u, v) and (v, w), respectively. If $\{u, w\} \in E(G')$, then replace n by a p-node with label (u, w), which has as its children an e-node with label (u, w), and an s-node with label (u, w) which in turn has two children: an e-node with label (u, v) and an e-node with label (v, w).

Now suppose each block of G is series parallel. For each block of G, make a tree decomposition of width two of G (which is possible according to Lemma 6). For each cut-vertex v of G, make a node $\{v\}$ and, for each block G' containing v, make an edge between node $\{v\}$ and an arbitrary node in the tree decomposition of G' which contains v. If G is not connected, connect the tree decompositions of the connected components. This results in a tree decomposition of G of width two. □

The algorithm for computing a tree decomposition of minimum width of an input graph G, if $\mathrm{tw}(G) \leq 2$, is now as follows, if G is connected.

1. Compute the blocks of G
   if $\mathrm{tw}(G) > 2$ **return** false
2. **for all** blocks G' **do in parallel**
3. Compute the sp-tree of G'
4. Change sp-tree into tree decomposition of minimum width
5. Connect tree decompositions of blocks together

If the input graph G is not connected, then we first add an auxiliary vertex r to G, which has one edge to each connected component of G (if $\mathrm{tw}(G) \leq 2$, then $\mathrm{tw}(G \cup \{r\}) \leq 2$). After the tree decomposition has been found, vertex r can be removed from each node in which it occurs, which gives a tree decomposition of G of minimum width. The selection of a vertex in each connected component of G to connect r to, can be done in a similar way as is given in [7]. The resource bounds of the algorithm do not change by this.

In the remainder of this section, we describe the different steps of the algorithm.

4.1 Computing the Blocks

The computation of the blocks in step 1 of the algorithm is done with the algorithm from Sect. 3 for $k = 2$. This takes $O(n)$ operations, and $O(\log n)$ time on a CRCW PRAM or $O(\log n \log^* n)$ on an EREW PRAM. If G does not have treewidth at most two, then this algorithm returns false, and we can return false.

4.2 Computing the sp-trees

In step 3, the algorithm finds an sp-tree for each of the blocks of G. We use the algorithm of [6] for finding sp-trees (we will refer to this algorithm as the sp-algorithm). The sp-algorithm is based on a finite, safe, complete and terminating set of reduction rules, of which $\Omega(n')$ applications are possible on a series parallel graph with n' vertices. The structure of the algorithm is roughly the same as the structure of our block algorithm: first, the graph is reduced in $O(\log n')$ reduction rounds. After that, an sp-tree for the reduced graph is constructed, and the reductions are undone in reversed order. During undoing a reduction, the sp-tree is locally adapted for the new graph.

On an input graph with n' vertices, the sp-algorithm uses $O(n')$ operations, and $O(\log n')$ time on a CRCW PRAM or $O(\log n' \log^* n')$ on an EREW PRAM. If we run it in parallel on all blocks of the graph G, then step 3 takes $O(n)$ operations, $O(\log n)$ time on a CRCW PRAM and $O(\log n \log^* n)$ on an EREW PRAM.

4.3 Turning the sp-Trees into Tree Decompositions

In step 4, the sp-trees that result from step 3 are transformed into tree decompositions of minimum width of each block of G. For this transformation we use the construction in the proof of Lemma 6: each node in the sp-tree is transformed into a node in the tree decomposition. This can be done with $O(n)$ operations in $O(1)$ time on an EREW PRAM: use one processor for each node in each sp-tree (the total number of nodes in the sp-trees is $O(n)$ nodes, since the number of edges in a series parallel graph is linear in the number of vertices).

4.4 Connecting the Tree Decompositions

In step 5 of the algorithm, the tree decompositions of the different blocks of G are combined into one tree decomposition of minimum width of G. For each pair (v, B), where v is a vertex of G and B is a block which contains v, we do the following, in parallel. If v is not a cut-vertex, then do nothing. If v is a cut-vertex, then we make a node $\{v\}$ in the tree decomposition, we connect this node to an arbitrary node in the tree decomposition of B which contains v, and we connect the node to the node $\{v\}$ which corresponds to the block B' that contains v and is the successor of B in the list of blocks containing v (so if B is the last block in the list, then it is not connected to another node $\{v\}$). This can all be done in constant time on an EREW PRAM. The number of operations is $O(n)$, since the number of pairs (v, B) with $v \in V(B)$ is $O(n)$.

We conclude with the main result of this section.

Theorem 8. *There is a parallel algorithm which checks if a given graph G has treewidth at most two, and if so, returns a tree decomposition of minimum width of G.*

The algorithm uses $O(n)$ operations and space, and $O(\log n)$ time on a CRCW PRAM, or $O(\log n \log^* n)$ time on an EREW PRAM.

Acknowledgement. We thank Torben Hagerup for helpful discussions and information.

References

1. S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *J. ACM*, 40:1134–1164, 1993.
2. H. L. Bodlaender. NC-algorithms for graphs with small treewidth. In J. van Leeuwen, editor, *Proceedings 14th International Workshop on Graph-Theoretic Concepts in Computer Science WG'88*, pages 1–10. Springer Verlag, Lecture Notes in Computer Science, vol. 344, 1988.
3. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, pages 226–234, New York, 1993. ACM Press.
4. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.
5. H. L. Bodlaender and B. de Fluiter. Reduction algorithms for graphs with small treewidth. Technical Report UU-CS-1995-37, Department of Computer Science, Utrecht University, Utrecht, 1995.
6. H. L. Bodlaender and B. de Fluiter. Parallel algorithms for series parallel graphs. Manuscript, 1996.
7. H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. In Z. Fülöp and F. Gécseg, editors, *Proceedings 22nd International Colloquium on Automata, Languages and Programming*, pages 268–279, Berlin, 1995. Springer-Verlag, Lecture Notes in Computer Science 944.
8. R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.
9. R. Cole and U. Vishkin. Approximate parallel scheduling II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92:1–47, 1991.
10. B. Courcelle. Graph rewriting: an algebraic and logical approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 192–242, Amsterdam, 1990. North Holland Publ. Comp.
11. B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
12. D. Granot and D. Skorin-Kapov. NC algorithms for recognizing partial 2-trees and 3-trees. *SIAM J. Disc. Meth.*, 4(3):342–354, 1991.
13. T. Hagerup. Optimal parallel algorithms on planar graphs. *Information and Computation*, 84:71–96, 1990.
14. T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM J. Comput.*, 19:678–704, 1990.
15. J. Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. *J. Algorithms*, 20:20–44, 1996.
16. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986.

This article was processed using the LaTeX macro package with LLNCS style