Parallel Algorithms with Optimal Speedup for Bounded Treewidth*

Hans L. Bodlaender† and Torben Hagerup‡

Abstract

We describe the first parallel algorithm with optimal speedup for constructing minimum-width tree decompositions of graphs of bounded treewidth. On n-vertex input graphs, the algorithm works in $O((\log n)^2)$ time using $O(n)$ operations on the EREW PRAM. We also give faster parallel algorithms with optimal speedup for the problem of deciding whether the treewidth of an input graph is bounded by a given constant and for a variety of problems on graphs of bounded treewidth, including all decision problems expressible in monadic second-order logic. On n-vertex input graphs, the algorithms use $O(n)$ operations together with $O(\log n \log^* n)$ time on the EREW PRAM, or $O(\log n)$ time on the CRCW PRAM.

1 Introduction

The concept of treewidth has proved to be a useful tool in the design of graph algorithms: Many important classes of graphs have bounded treewidth, and many important graph problems that are otherwise quite hard can be solved efficiently on graphs of bounded treewidth. A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(T, \mathcal{U})$, where $T = (X, F)$ is a tree and $\mathcal{U} = \{U_x \mid x \in X\}$ is a family of subsets of V called *bags*, one for each node in T, such that

- $\bigcup_{x \in X} U_x = V$ (every vertex in G occurs in some bag);
- for all $(v, w) \in E$, there exists an $x \in X$ such that $\{v, w\} \subseteq U_x$ (every edge in G is "internal" to some bag);

*Partially supported by the ESPRIT Basic Research Actions Program of the EU under contract No. 7141 (project ALCOM II). A preliminary version of this paper was presented at the 22nd International Colloquium on Automata, Languages and Programming (ICALP) in July 1995.

†Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. Email: hansb@cs.ruu.nl.

‡Max-Planck-Institut für Informatik, Im Stadtwald, D–66123 Saarbrücken, Germany. Email: torben@mpi-sb.mpg.de.

– for all $x, y, z \in X$, if y is on the path from x to z in T, then $U_x \cap U_z \subseteq U_y$ (every vertex in G occurs in the bags in a connected part of T, i.e., in a subtree).

The *width* of a tree decomposition $(T, \{U_x \mid x \in X\})$ is $\max_{x \in X} |U_x| - 1$. The *treewidth* of a graph G, denoted $tw(G)$, is the smallest treewidth of any tree decomposition of G. *Path decompositions* and *pathwidth* are defined analogously, with the tree T restricted to be a path.

The majority of efficient algorithms for graphs of bounded treewidth depend not only on a guarantee that the treewidth of an input graph is small, but in fact on the availability of a minimum-width tree decomposition of the input graph, so that the construction of minimum-width tree decompositions for graphs of bounded treewidth is a key problem. A quest for the fastest possible algorithm for this problem [5, 39, 20, 10, 34, 31, 38, 8] led to the linear-time algorithm of [8], which eliminated the bottleneck in a large number of algorithms for bounded treewidth.

In the setting of parallel computation, the situation is similar. Many otherwise difficult graph problems can be solved in NC (i.e., in polylogarithmic time with a polynomial amount of hardware) on graphs of bounded treewidth, and again the need for a minimum-width tree decomposition is a serious bottleneck. The best parallel algorithms for computing tree decompositions of width k of graphs of treewidth k, for fixed k, run on the CRCW PRAM using $O((\log n)^2)$ time and $O(n^{2k+5})$ processors [13, 14], or $O(\log n)$ time and $O(n^{3k+4})$ processors [7]. Although these algorithms are fast, they are extremely wasteful in terms of processors, in view of the linear-time sequential algorithm. A related result was obtained by Wanke [41], who showed that the problem of deciding whether the treewidth of an input graph is bounded by a constant k belongs to the complexity class LOGCFL; this result also does not seem to lead to parallel algorithms that are efficient from the point of view of processor utilization. If we relax the requirements by allowing tree decompositions of width $O(k)$, rather than exactly k, more algorithms come into play: Lagergren [31] finds a decomposition of width $\leq 6k + 5$ in $O((\log n)^3)$ time using n processors, and we believe that Reed's sequential $O(n \log n)$-time algorithm [38] for obtaining a decomposition of width $\leq 4k + 3$ can be parallelized (using an algorithm of Khuller and Schieber [29] to solve a central path-finding problem) to yield an algorithm that works in $O((\log n)^2)$ time using $O(n\alpha(n)/\log n)$ processors, where α is a very slowly growing "inverse Ackermann" function. The parallel version of Reed's algorithm uses $O(n\alpha(n) \log n)$ *operations*, i.e., has a time-processor product of $O(n\alpha(n) \log n)$, and is the most efficient of the parallel algorithms discussed above. Still, since the problem can be solved in linear sequential time, it does not have optimal speedup, which requires a time-processor product of $O(n)$.

We describe an EREW PRAM algorithm for constructing minimum-width tree decompositions for graphs of bounded treewidth in $O((\log n)^2)$ time using $O(n)$ operations. The algorithm achieves optimal speedup and is the first parallel algorithm to do so. Moreover, the new algorithm is second in speed only to Bodlaender's algorithm [7], but uses a weaker model of computation (the EREW PRAM versus the CRCW PRAM), on which Bodlaender's algorithm can be simulated only in the same time of $O((\log n)^2)$. The new result immediately implies that a large number of problems on graphs of bounded treewidth can now be solved by parallel algorithms with optimal speedup.

A subroutine used in the construction algorithm but of independent interest is a parallel version of an algorithm due to Bodlaender and Kloks [11]. The algorithm takes as input a tree decomposition of bounded width of a graph G and outputs a minimum-width tree decomposition of G, thus blurring the distinction between the "exact" and the "approximate" construction algorithms discussed above. The new algorithm runs in $O(\log n)$ time using $O(n)$ operations on the EREW PRAM.

While we cannot compute tree decompositions faster than in $O((\log n)^2)$ time, it turns out that we can give faster algorithms for the related problem of deciding whether the treewidth of an input graph is bounded by a given constant k. The algorithms have optimal speedup (i.e., use $O(n)$ operations) and run in $O(\log n \log^* n)$ time on the EREW PRAM, or in $O(\log n)$ time on the CRCW PRAM. We achieve the same resource bounds for a number of problems on graphs of bounded treewidth, including all problems expressible in monadic second-order logic. These algorithms operate without an explicit tree decomposition and so bypass the (time) bottleneck of our construction algorithm. Furthermore, we achieve the same results for path decompositions and pathwidth as for tree decompositions and treewidth.

The paper combines several different techniques of wide applicability. The *graph-reduction* technique consists in repeatedly replacing parts of the graph at hand by simpler parts until a trivial graph results. The problem of interest is then solved for the trivial graph, and the solution is "carried along" while the changes to the graph are undone in reverse order. This technique pervades the paper and is used in the construction algorithm as well as in the decision algorithms (and also in the width-minimizing algorithm, provided that tree contraction is viewed as a special case of graph reduction).

Another technique used in the derivation of the CRCW PRAM decision algorithm from the corresponding EREW PRAM algorithm is that of *derandomization*. The basic idea of derandomization is that instead of letting random coin tosses select one algorithm to be executed from a collection of deterministic algorithms, we execute all deterministic algorithms in the collection and pick the best output. Because of the need to simulate several possible executions, derandomization is often associated with a price in the form of increased resource

requirements. Here we use derandomization to derive a parallel algorithm with optimal speedup, i.e., we pay no price.

A third technique of less general applicability but nonetheless independent interest is that of *bounded adjacency-list search*, which tries to circumvent the difficulties caused by high-degree vertices in parallel algorithms for sparse graphs by letting each neighbor of a high-degree vertex v inspect only a piece of constant size of the adjacency list of v near its own entry, rather than the whole adjacency list. The bounded adjacency-list search technique was used previously (although not named) in [24]; there, as here, the technique serves to eliminate the need both for concurrent reading and writing and for superlinear space.

Unlike certain related results, most of our algorithms are explicit and do not rely on nonconstructive arguments. Only the results of Theorems 5.1 and 6.1 are nonconstructive, but can be made constructive in many concrete cases, as discussed near the end of Section 5. On the other hand, it should be noted that large constant factors prevent our algorithms from being practical.

All graphs in this paper are undirected, loopless and without multiple edges. We assume that all graphs, not excluding trees, are represented according to an *adjacency-list* representation: Each vertex v in an n-vertex graph G is represented by an integer name of size $O(n)$ and has a pointer to a doubly-linked adjacency list with an entry for each neighbor of v in G. For each neighbor w of v, the entry of w in v's adjacency list contains the name of w as well as a *cross pointer* to the entry of v in w's adjacency list.

2 Minimizing decomposition width

In this section we show how to obtain a minimum-width tree decomposition of a graph G from any tree decomposition of G of bounded width. We begin with an observation that allows us to assume that tree decompositions are rooted, binary and of logarithmic depth whenever this is convenient. In representation terms, every nonroot node in a rooted tree knows its parent, and a rooted tree is *binary* if no node has more than two children.

We appeal twice to the *tree-contraction* technique introduced by Miller and Reif [35], which we therefore describe in generic terms. Applied to an n-node input tree $T = (V, E)$, a tree-contraction algorithm produces a sequence $T = T_0 = (V_0, E_0), T_1 = (V_1, E_1), \ldots, T_r = (V_r, E_r)$ of $O(\log n)$ binary trees, ending with a one-node tree T_r, such that each tree T_i, for $i = 1, \ldots, r$, is obtained from its predecessor T_{i-1} in the sequence by contracting a set of edges $F_{i-1} \subseteq E_{i-1}$ with the following properties:

- F_{i-1} spans a matching (i.e., no node in V_{i-1} is incident to more than one edge in F_{i-1});
- Each edge in F_{i-1} has at least one endpoint of degree 1 or 2 in T_{i-1}.

A sequence $T_0, \ldots, T_r$ with these properties, called a *contraction sequence* for T, can be computed in $O(\log n)$ time using $O(n)$ operations and $O(n)$ space [1, 16, 21, 22, 30] (the connection to our generic description of tree contraction is easiest to establish in the case of the simple and elegant algorithm of [1, 30]).

Let $X = \bigcup_{i=0}^{r} V_i$. We can define a rooted, binary tree T' on the node set X, called the *contraction tree* corresponding to the sequence $T_0, \ldots, T_r$, in the following way: The nodes in V, which will be called *base nodes*, are the leaves of T', and whenever a node $x \in X$ results from the contraction of an edge (y, z), we make x the parent of y and z in T'. For all $x, y \in X$, we will say that x *contains* y if x is an ancestor of y in T'. The base nodes contained in any node in X span a connected subgraph of the input tree T. For $i = 1, \ldots, r$, we root T_i at the node in V_i containing the root of T.

For $i = 0, \ldots, r$, call two base nodes v and w *i-neighbors* if $(v, w) \in E$ and v and w are not contained in the same node in V_i. For each $x \in V_i$, the i-neighbors of base nodes contained in x are contained in distinct neighbors of x in T_i. For all $x \in X$, we define the *border* of x as the set of base nodes contained in x and adjacent in T to one or more base nodes not contained in x. For $i = 0, \ldots, r$, if $x \in V_i$, then a base node contained in x belongs to the border of x precisely if it has at least one i-neighbor; in particular, the border of x contains at most 3 nodes.

Lemmas 2.1 and 2.2 below slightly improve a result of [7] by employing a more efficient subroutine; the same improvement was observed in [27].

Lemma 2.1 *The following problem can be solved on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space: Given an n-node rooted, binary tree T, compute a rooted, binary tree decomposition of T of depth $O(\log n)$ and width at most 2.*

Proof: Begin by using tree contraction to compute a contraction sequence $T = T_0 = (V_0, E_0), T_1 = (V_1, E_1), \ldots, T_r = (V_r, E_r)$ for the input tree $T = (V, E)$ and construct the corresponding contraction tree $T' = (X, F)$.

Observe that if e and e' are the edges incident on a node v of degree 2 in some tree H, then the tree obtained from H by contracting e is the same as the tree obtained from H by contracting e' (the contraction can be "flipped" to the other side of v). Because of this, we can assume without loss of generality that, when an edge between a node v of degree 2 and a node w of degree 3 is contracted in the transition from T_{i-1} to T_i, for some i with $1 \leq i \leq r$, then v is the parent of w in T_{i-1}. To see this, first note that all edge contractions that violate the assumption — we will call these $(3,2)$-*contractions* — can be carried out separately (i.e., we replace the one-step transition from T_{i-1} to T_i by a two-step process, thereby doubling r). Now each $(3,2)$-contraction can be "flipped", as described above, without changing the resulting tree; note that the edge set of the "flipped" contractions still spans a matching.

For all $x \in X$, denote by $B(x)$ the border of x. The sets $B(x)$, where $x \in X$, can be computed as follows: Successively, for $i = 0, \ldots, r$, we compute $B(x)$, along with the set of i-neighbors of all nodes in $B(x)$, for all $x \in V_i$. This is trivial for $i = 0$, and if, for some i with $1 \leq i \leq r$, a node $x \in V_i$ is obtained by contracting an edge (y, z), where $y, z \in V_{i-1}$, then $B(x) \subseteq B(y) \cup B(z)$, and an $(i-1)$-neighbor of a base node $v \in B(y) \cup B(z)$ is also an i-neighbor of v exactly if it does not belong to $B(y) \cup B(z)$, so that the information pertaining to x can easily be derived from the information pertaining to y and z.

We now associate a set $U_x \subseteq V$ with each $x \in X$ as follows: If x is a base node, i.e., a leaf in T', we take $U_x = \{x\}$. Otherwise, if x has the children y and z in T', we take $U_x = B(y) \cup B(z)$. We will show that $(T', \{U_x \mid x \in X\})$ is a tree decomposition of T. First, because of the convention regarding leaves of T', the condition $\bigcup_{x \in X} U_x = V$ is trivially satisfied. Second, for every edge (v, w) in E, it is easy to see that $\{v, w\} \subseteq U_x$, where x is the least common ancestor of v and w in T'. And third, the set of nodes whose bags contain a base node v form an initial part of the path in T' from v to the root of T', and hence span a connected subgraph of T'.

The width of the tree decomposition defined above is bounded by one less than twice the maximum size of a border. We will now show that $|B(x)| \leq 2$ for all $x \in X$.

Suppose that for some i with $1 \leq i \leq r$, some base node $v \in V$ has two i-neighbors and belongs to $B(x)$ for some $x \in V_i$ with $|B(x)| \geq 2$. Then neither of the two i-neighbors of v is its parent in T. To see why this is true, let j be minimal such that v belongs to $B(y)$ for some $y \in V_j$ with $|B(y)| \geq 2$. It can be seen that then y must, in fact, result from the contraction of an edge (v, z), where $z \in V_{j-1}$ is of degree 2 in T_{j-1}. But then, by the absence of $(3,2)$-contractions, z must be the parent of v in T_{j-1}, which implies that the parent of v in T is not an i-neighbor of v for any $i \geq j$.

Since the claim $|B(x)| \leq 2$ is trivially true for all base nodes x, assume by induction that it is true for all $x \in V_{i-1}$, for some i with $1 \leq i \leq r$, and consider a node $x \in V_i$ resulting from the contraction of an edge (y, z), where $y, z \in V_{i-1}$. Since $|B(x)|$ is bounded by the degree of x in T_i, which is two less than the sum of the degrees of y and z in T_{i-1}, we can assume that y is of degree 3 and that z is of degree 2 in T_{i-1}. We will show that only one node in each of $B(y)$ and $B(z)$ also belongs to $B(x)$, from which $|B(x)| \leq 2$ follows immediately. In the case of $B(z)$, this is easy to see: The nodes in $B(z)$ have a total of two $(i-1)$-neighbors, and one of these is not an i-neighbor. Similarly, the nodes in $B(y)$ lose one of their three $(i-1)$-neighbors. We must show that the two remaining $(i-1)$-neighbors, which are also i-neighbors, are adjacent to the same node in $B(y)$. But if this is not the case, then $|B(y)| = 2$ and $B(y)$ contains a node v with two $(i-1)$-neighbors, one of which belongs to $B(z)$. By what was shown above, neither of the $(i-1)$-neighbors of v is its parent in T. But this contradicts the

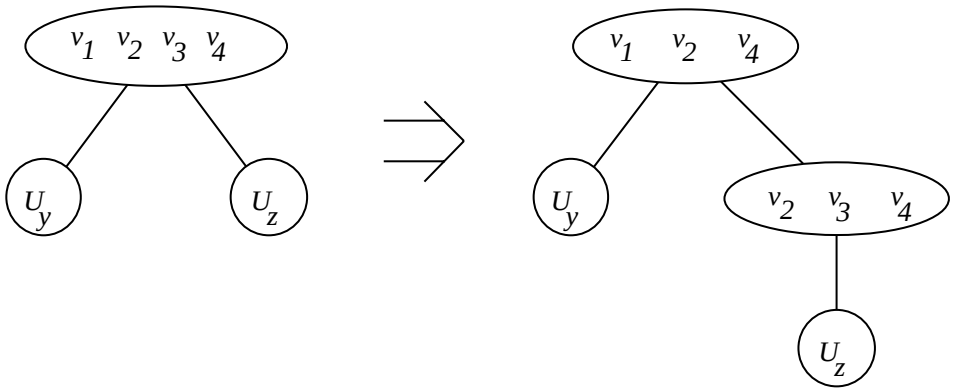

Figure 1: Transforming from width 3 to width 2.

fact that z must be the parent of y in T_{i-1}, by the absence of $(3,2)$-contractions.

The tree decomposition described so far is of width at most 3. We now describe how to reduce the width to at most 2. Suppose that $x \in X$ is a node in T' whose associated bag U_x is of size 4, let y and z be the children of x in T' and take $B(y) = \{v_1, v_2\}$ and $B(z) = \{v_3, v_4\}$, so that $U_x = \{v_1, v_2, v_3, v_4\}$. As follows easily from arguments used to bound the sizes of all bags by 2, $B(x)$ "inherits" exactly one element of each of $B(y)$ and $B(z)$, so that we can assume that $B(x) = \{v_1, v_4\}$. Then v_2 and v_3 do not occur in the bag of the parent of x in T', if any. Moreover, $(v_2, v_3) \in E$ and hence $(v_1, v_3) \notin E$. It is now easy to see that the transformation illustrated in Fig. 1 preserves the defining properties of a tree decomposition. Applied at all nodes with bags of size 4, it produces a new tree decomposition of T of width at most 2. The depth increases by a factor of at most 2 and hence remains $O(\log n)$, as desired.

Starting from the sequence $T_0, \ldots, T_r$, the algorithm constructs the tree T', then computes the sets $B(x)$ and U_x for all $x \in X$, and finally carries out the transformation of T' described above. Each of these steps can easily be done in $O(\log n)$ time using $O(n)$ operations and $O(n)$ space. □

In the lemma below as well as in several later results, the input parameter k is qualified as being a constant, the meaning of which is that k can be any positive integer, but that the $O(\cdots)$ of the result may (and will) hide factors that depend on k.

Lemma 2.2 *For all constants $k \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space: Given a tree decomposition with n nodes and of width k of a graph G, compute a rooted, binary tree decomposition of G of depth $O(\log n)$ and width at most $3k + 2$.*

Proof: We begin by replacing each node of degree $m \geq 4$ and with associated bag U in the given tree decomposition by a path of $m - 2$ nodes, each of degree

3 and with associated bag U, which obviously preserves the defining properties of a tree decomposition. Then we construct an Euler tour of the modified tree (see [40]) and root the tree by breaking the Euler tour at an arbitrary node of degree at most 2, declared to be the root, computing the distance from each node to the root along the Euler tour by means of list ranking [15, 4], and determining the parent of each nonroot node as the neighbor with a smaller distance to the root. After these preliminary steps, which can easily be carried out within the stated resource bounds, we can assume that the input is a tree decomposition $(T,\mathcal{U})$, where $T = (V, E)$ is rooted and binary.

Now use the algorithm of Lemma 2.1 to obtain a rooted, binary tree decomposition $(T', \mathcal{Q})$ of T of width at most 2 and depth $O(\log n)$. Replacing each node in a bag in $\mathcal{Q}$ by the vertices of G in its own associated bag, we obtain the desired tree decomposition of G. More precisely, write $\mathcal{U} = \{U_v \mid v \in V\}$, $T' = (X, F)$ and $\mathcal{Q} = \{Q_x \mid x \in X\}$ and take $R_x = \bigcup_{v \in Q_x} U_v$, for all $x \in X$. Then $(T', \{R_x \mid x \in X\})$ is a tree decomposition of G. For if a vertex u of G occurs in U_v, with $v \in V$, and $v \in Q_x$, with $x \in X$, then $u \in R_x$. Similarly, both endpoints of each edge in G occur in some bag U_v, with $v \in V$, and therefore also in some bag R_x, with $x \in X$. Finally, each vertex u of G occurs in the bags U_v in a subtree of T, and two adjacent nodes in this subtree occur in the bags Q_x in overlapping subtrees of T', so that, altogether, u occurs in the bags R_x in a connected subgraph of T'. The width of the tree decomposition $(T', \{R_x \mid x \in X\})$ is at most $(2 + 1)(k + 1) - 1 = 3k + 2$. □

We will use the phrase "balancing a tree decomposition" to describe an application of the algorithm implicit in the preceding lemma. The remaining goal in the present section is to prove the result below.

Theorem 2.3 *For all constants $k \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space: Given a tree decomposition with n nodes and of width k of a graph G, construct a minimum-width tree decomposition of G.*

The corresponding decision problem ("Is $tw(G) \leq l$?", for some given l) can be solved by a straightforward parallelization of the decision algorithm of [11]. The latter algorithm consists of a pass from the leaves to the root of a tree decomposition of the input graph, which, in light of Lemma 2.2, can be taken to be binary and of logarithmic depth. The processing of each node takes constant time, and all nodes on the same level in the tree can be processed in parallel. If the nodes in the tree decomposition are first sorted by their levels, which can be done in $O(\log n)$ time using $O(n)$ operations [37, Lemma 3.1], it is easy to process the whole tree in $O(\log n)$ time using $O(n)$ operations. The sequential construction algorithm processes the tree decomposition in three passes. In one of these, the processing of a node no longer necessarily takes constant time, so

that an amortization argument is used in [11] to bound the total running time by $O(n)$. Since this appears to stand in the way of a direct parallelization, we choose a somewhat different approach.

Suppose that the input graph is $G = (V, E)$. Close inspection of the algorithm of [11] (we omit the details, some of which were hinted at above) reveals that $O(\log n)$ time and $O(n)$ operations suffice to compute a certain *implicit representation* of the desired tree decomposition $(T = (X, F), \{U_x \mid x \in X\})$ consisting of the binary tree T (without the bags U_x) together with, for each $v \in V$, a collection $\mathcal{P}_v$ of disjoint simple paths in T whose union contains a node $x \in X$ if and only if $v \in U_x$. Rather than directly specifying the set of vertices contained in each bag, the implicit representation thus presents the set of bags containing each vertex in the form of a collection of disjoint paths. For each $v \in V$, a path in $\mathcal{P}_v$ with end nodes x and y is represented by marking both x and y with the triple (x, y, v); a node may be marked with several triples but, of course, with at most $k + 1$.

By the preceding discussion (in particular, note that $|X| = O(n)$), proving Theorem 2.3 boils down to showing the following.

Lemma 2.4 *For all constants $k \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space: Given a rooted, binary n-node tree T and a collection $\mathcal{P}$ of simple paths in T, each of which is labeled by an integer and represented, at each of its endpoints, by a triple specifying its endpoints and label, such that no node in T belongs to more than $k + 1$ paths in $\mathcal{P}$, mark each node x in T with the set of all labels of paths in $\mathcal{P}$ containing x.*

Proof: Since duplicates are easily eliminated, we can assume that no two paths in $\mathcal{P}$ have both the same endpoints and the same label, so that we can identify each path with the triple marking its endpoints. We will also assume that the endpoints of each path are distinct, since paths consisting of a single node are trivial to handle.

We begin by using tree contraction to obtain a contraction sequence $T = T_0 = (V_0, E_0), \ldots, T_r = (V_r, E_r)$ for the input tree $T = (V, E)$. We will process the sequence twice, first in the order of increasing indices (the *up phase*), and then in the order of decreasing indices (the *down phase*).

Let $X = \bigcup_{i=0}^{r} V_i$. During the up phase, we associate a set $S(x)$ with each node $x \in X$. For $x \in V$, $S(x)$ is the set of paths in $\mathcal{P}$ with x as an endpoint. For $i = 1, \ldots, r$, if a node $x \in V_i$ results from the contraction of an edge (v, w), where $v, w \in V_{i-1}$, we compute $S(x)$ as $(S(v) \cup S(w)) \backslash (S(v) \cap S(w))$; it is easy to see by induction that for all $x \in X$, $S(x)$ will be the set of paths in $\mathcal{P}$ with exactly one endpoint contained in x. During the down phase, for $i = r, \ldots, 1$, we modify $S(v)$ for all $v \in V_{i-1} \backslash V_i$ as follows: Suppose that v and another node $w \in V_{i-1}$ are both contained in $x \in V_i$. Then, for each pair (y, z) of neighbors of x in T_i such

that some node in V_{i-1} contained in y is separated, in T_{i-1}, from some node in V_{i-1} contained in z by the removal of v (i.e., v is "between" y and z), add to $S(v)$ all paths in $S(y) \cap S(z)$. Again, it is not difficult to see by backwards induction on i that the final value of $S(x)$, for all $x \in X$, will be the set of paths in $\mathcal{P}$ comprising at least one node contained in x and at least one node not contained in x. In particular, the values $S(v)$, where $v \in V$, precisely constitute the desired output. Since the number of paths containing a given node in V is bounded by a constant, and since each path in $S(x)$ must contain at least one of the at most 3 border nodes of x, for all $x \in X$, each set $S(x)$ is of constant size, and the whole computation can be carried out in $O(\log n)$ time using $O(n)$ operations and $O(n)$ space. □

With a similar (but, in fact, easier) argument one can also show the result below.

Theorem 2.5 *For all constants $k, l \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space: Given a tree decomposition with n nodes and of width k of a graph G, decide whether the pathwidth of G is at most l and, if so, construct a minimum-width path decomposition of G.*

3 A structural lemma

In this section we provide the basis for showing that any sufficiently large connected graph of bounded treewidth admits a large number of reductions of certain types. Moreover, given any adjacency-list representation of the graph, a large fraction of these reductions can be identified efficiently.

A well-known fact that we shall use below is that every n-vertex graph of treewidth $\leq k$ contains at most kn edges, for all positive integers n and k. We provide a brief proof. Since removing a vertex from a graph of treewidth $\leq k$ with at least two vertices leaves a graph of treewidth $\leq k$, it suffices to show that every graph G of treewidth $\leq k$ contains a vertex of degree $\leq k$. To this end consider a tree decomposition $(T = (X, F), \{U_x \mid x \in X\})$ of G of width $\leq k$ with a minimal number of nodes (i.e., $|X|$ is minimal, over all such tree decompositions) and pick a node $x \in X$ of degree ≤ 1 in T. U_x contains at least one vertex v that does not occur in any other bag (otherwise x would be superfluous), and v has at most k neighbors, as desired (they all belong to U_x).

The *boundary* of a subgraph H of a graph G is the set of those vertices in G that have at least one neighbor in H, but do not themselves belong to H. Let d, k, $n_{\min}$ and $n_{\max}$ be positive integers, to be characterized more closely in the following. A vertex will be called *small* if its degree is bounded by d, and *large* otherwise. Given a graph G of treewidth at most k, we are essentially looking

for connected subgraphs of G consisting of between $n_{\min}$ and $n_{\max}$ small vertices and with a boundary of size at most $2(k+1)$. It turns out that such subgraphs may not occur in G at all, for which reason we have to replace the connectedness condition by a weaker, more complicated condition described below after the introduction of additional terminology.

Two vertices are said to be *twins* if they have the same set of neighbors. By analogy, we call two subgraphs of a common graph twins if they have the same boundary. A *weakly connected component* of a subgraph H of a graph G is a connected component of the graph obtained from H by the introduction of an edge between each pair of nonneighbors in H with a common small neighbor in G; a weakly connected component of H may thus comprise several (usual) connected components of H, linked indirectly via small common neighbors in the boundary of H. A subgraph that consists of a single weakly connected component is *weakly connected.* Given an adjacency-list representation of a graph G, two disjoint subgraphs H_1 and H_2 of G are said to be *acquainted* if the intersection of their boundaries contains a vertex in whose adjacency list some entry of a vertex in H_1 is separated from some entry of a vertex in H_2 by a distance of at most d. This definition, which embodies the bounded adjacency-list search technique, reflects the fact that H_1 can "discover" H_2 by searching through a piece of length at most $2d+1$ of the adjacency list of each of its boundary vertices.

We can now define the objects of interest and state the main result of the section. A *valley* in a graph G is a weakly connected subgraph of G induced by a set of at most $n_{\max}$ small vertices and with a boundary of size at most $2(k+1)$. A *plain* (or $(d,k,n_{\min},n_{\max})$-plain, for emphasis) in G (relative to a particular adjacency-list representation of G) is a subgraph of G induced by at least $n_{\min}$ and at most $n_{\max}$ vertices, whose weakly connected components are pairwise acquainted twin valleys.

Lemma 3.1 *For all integer constants $k, n_{\min} \geq 1$, there are constants $d, n_{\max} \geq 1$ and $c > 0$ such that every connected graph with $n > n_{\max}$ vertices and treewidth at most k contains at least cn disjoint $(d,k,n_{\min},n_{\max})$-plains (relative to any adjacency-list representation).*

Proof: Take $b = 3(k+1)(n_{\min}+1)$. We will prove the lemma with $n_{\max} = 3b$ and $d = 2^{k+4}\, n_{\min}\, n_{\max}$. Let $G = (V,E)$ be a connected graph with $n > n_{\max}$ vertices and treewidth at most k and fix a particular adjacency-list representation of G and a particular maximal collection $\mathcal{P}$ of disjoint $(d,k,n_{\min},n_{\max})$-plains in G. We will show that $|\mathcal{P}| \geq cn$ for a suitably chosen constant $c > 0$.

Let $(T,\mathcal{U})$ be a tree decomposition of G of width at most k and write $T = (X,F)$ and $\mathcal{U} = \{U_x \mid x \in X\}$. We view T as rooted at an arbitrary node. Using the same standard transformation as in the beginning of the proof of Lemma 2.2, we can assume without loss of generality that T is binary. On two occasions in the proof we will use the fact that if $v \in V$, then the subgraph T_v of T induced

by the node set $\{x \in X \mid v \in U_x\}$ is a tree, whose root can therefore be reached from any node in T_v by going from a child node to a parent node zero or more times; we will refer to this as the *root-seeking principle.*

We begin by showing that the set X of tree nodes can be partitioned into disjoint *clusters* $C_1, \ldots, C_s$ such that for $i = 1, \ldots, s$,

(1) C_i induces a subtree of T;

(2) $|\bigcup_{x \in C_i} U_x| \leq n_{\max}$;

(3) If C_i does not contain the root of T, then $|\bigcup_{x \in C_i} U_x| \geq b$.

The partition $C_1, \ldots, C_s$ can be constructed by a simple procedure that processes T in inverse topological order, i.e., every node is processed after all of its children. The processing of a node y computes the set C consisting of y itself and all descendants of y that have not yet been assigned to clusters. If $|\bigcup_{x \in C} U_x| \geq b$ or y is the root of T, then C is made into a new cluster; otherwise the processing continues to the next node.

It is easy to see that the set of nodes assigned to a cluster always induces a connected subgraph of T, i.e., condition (1) above is satisfied. Condition (3) is satisfied by construction. As for condition (2), observe that if a cluster C is formed during the processing at a node y, then C receives a contribution of at most $b - 1$ vertices from each of the at most two children of y and of at most $k + 1$ vertices from y itself, a total of at most $2b + k - 1 \leq n_{\max}$ vertices. This establishes properties (1)–(3) of the cluster partition.

For $i = 1, \ldots, s$, let C_i' be the set of those nodes in C_i that are adjacent in T to a node not in C_i, take $Z_i = \bigcup_{x \in C_i'} U_x$ and let H_i be the graph induced by the vertices in $(\bigcup_{x \in C_i} U_x) \backslash Z_i$. By the properties of tree decompositions, the graphs $H_1, \ldots, H_s$ are disjoint, and the boundary of H_i is contained in Z_i, for $i = 1, \ldots, s$.

For $i = 1, \ldots, s$, we will call H_i a *kernel* if $|C_i'| \leq 2$ and C_i is not the cluster containing the root of T (we exclude the latter cluster because of its special status). We next establish a lower bound on the number of kernels. By property (2) of the cluster partition and the fact that $\bigcup_{i=1}^{s} \bigcup_{x \in C_i} U_x = V$, the number s of clusters is at least $n/n_{\max}$. The clusters form a *cluster tree* in a natural fashion: Two clusters are adjacent if one contains a node adjacent in T to a node in the other cluster, and the degree of a cluster C_i is at least $|C_i'|$. In general, a tree with m nodes contains $m - 1$ edges. Hence if h denotes the number of nodes of degree ≥ 3 in an m-node tree and $m \geq 2$, we have $3h + (m - h) \leq 2(m - 1)$ or $h \leq m/2 - 1$. Applying this to the cluster tree, with at least $n/n_{\max} > 1$ nodes, only one of which contains the root of T, shows that the number of clusters of degree ≤ 2 is at least $1 + \frac{1}{2} n/n_{\max}$, and hence that the number of kernels is at least $\frac{1}{2} n/n_{\max}$.

The boundary of a kernel H contains at most $2(k+1)$ vertices and, by property (2) of the cluster partition, H contains no more than $n_{\max}$ vertices. In particular, since $d \geq n_{\max} + 2(k+1)$, all vertices in H are small. It follows that every weakly connected component of H is a valley. We will call a valley of this kind *good* if it is part of a plain in $\mathcal{P}$, and *bad* otherwise. Note that a bad valley contains fewer than $n_{\min}$ vertices (otherwise it would be a plain, contradicting the maximality of $\mathcal{P}$). A bad valley with boundary B will be called a B-valley.

We classify the bad valleys into three types depending on their boundaries. Consider a bad valley L with boundary B. If B contains one or more small vertices, L is of type (a). If B contains only large vertices and $B \not\subseteq U_x$ for all $x \in X$, L is of type (b). If B contains only large vertices and $B \subseteq U_x$ for some $x \in X$, finally, L is of type (c). We next bound the number of valleys of type (a) per kernel, the number of kernels containing valleys of type (b), and the total number of valleys of type (c).

Type (a) valleys (B contains a small vertex).

A kernel can contain at most $2(k+1)$ valleys of type (a), since each such valley "uses up" one or more of the at most $2(k+1)$ boundary vertices. We here use the fact that valleys need only be weakly connected: Two vertices in the same kernel and with a common small neighbor belong to the same valley.

We now consider the B-valleys for which B contains only large vertices. Given such a B-valley, choose $v \in B$ such that the root r_v of the subtree T_v of T induced by the node set $\{x \in X \mid v \in U_x\}$ is of maximal depth and assign the B-valley to v. We here use the fact that G is connected, which ensures that $B \neq \emptyset$.

Type (b) valleys ($B \not\subseteq U_x$ for all $x \in X$).

In this case we can conclude from the root-seeking principle that r_v lies within the cluster C containing the B-valley under consideration; otherwise B would be contained in U_x, where x is the node in C of minimal depth in T. Since v is large, the number of clusters containing valleys of type (b) is therefore bounded by the number of large vertices. Because G has at most kn edges, the latter number in turn is bounded by $(2k/d) \cdot n$.

Type (c) valleys ($B \subseteq U_x$ for some $x \in X$).

By definition, all B-valleys are twins. Thus no d consecutive entries in the adjacency list of v can contain entries of vertices in $n_{\min}$ or more different B-valleys, since then all of these (bad) B-valleys would be acquainted, and some of them (with a suitable total size) would form a plain, contradicting the maximality of $\mathcal{P}$. We may conclude that at most $\lceil \deg(v)/d \rceil \cdot n_{\min} \leq 2\deg(v) \cdot n_{\min}/d$ B-valleys are assigned to v, where $\deg(v)$ denotes the degree of v and the inequality follows from the fact that $\deg(v) > d$.

The valleys assigned to v may not all be twins. However, the choice of v and the root-seeking principle ensure that if a B-valley is assigned to v and B is contained in some (single) bag, then $B \subseteq U_{r_v}$. Thus the valleys of type (c)

assigned to v have at most 2^k different boundaries (all such boundaries are subsets of a fixed set of at most $k+1$ vertices, and all contain v). It follows that the total number of valleys of type (c) assigned to v is bounded by $2^{k+1}\deg(v) \cdot n_{\min}/d$. Again since the total number of edges is at most kn, this sums over all vertices v to at most $(2^{k+2}k \cdot n_{\min}/d) \cdot n$.

Since a bad valley contains fewer than $n_{\min}$ vertices and a kernel contains at least $b-2(k+1)$ vertices, each kernel containing only bad valleys decomposes into at least $(b-2(k+1))/n_{\min} \geq 3(k+1)$ bad valleys. At most $2(k+1)$ of these are of type (a). Hence if a kernel contains only bad valleys, then either one or more of these are of type (b), or at least $k+1$ of them are of type (c). The first condition applies to at most $(2k/d) \cdot n \leq \frac{1}{8}n/n_{\max}$ kernels, and because the total number of valleys of type (c) is bounded by $(2^{k+2}k \cdot n_{\min}/d) \cdot n$, the second condition applies to at most $(2^{k+2}k \cdot n_{\min}/(d(k+1))) \cdot n \leq \frac{1}{4}n/n_{\max}$ kernels. Since the total number of kernels is at least $\frac{1}{2}n/n_{\max}$, at least $(\frac{1}{2} - \frac{1}{4} - \frac{1}{8}) \cdot n/n_{\max} = \frac{1}{8}n/n_{\max}$ kernels contain one or more good valleys. At most $n_{\max}$ valleys can belong to the same plain, so the number of plains in $\mathcal{P}$ is at least cn if we take $c = 1/(8 \cdot n_{\max}^2)$. □

4 Constructing tree decompositions

In this section we show that minimum-width tree decompositions of n-vertex graphs of bounded treewidth can be constructed on an EREW PRAM using $O((\log n)^2)$ time and $O(n)$ operations. More precisely, given an n-vertex graph G and a constant k, our algorithm outputs either a tree decomposition of G of treewidth $tw(G)$ or an indication of the fact that $tw(G) > k$.

The algorithm is based on the graph-reduction technique: A connected input graph of treewidth $\leq k$ is successively replaced by smaller and smaller graphs in a series of *reductions* until a constant-size graph results. Starting from a minimum-width tree decomposition of the final constant-size graph, the reductions are then undone one by one in the reverse order of their application, where, in undoing a reduction that originally replaced a graph G' by a smaller graph G'', a minimum-width tree decomposition of G' is derived from one of G''. At the end of this process we obtain a minimum-width tree decomposition of the input graph.

Suppose that v and w are vertices in a graph G' that are either adjacent or twins and let G'' be the graph obtained from G' by removing v and its incident edges after first inserting an edge between w and each neighbor of v that was not previously a neighbor of w; we will call v and w *reduction partners* and say that G'' is obtained from G' by reduction *on* the pair $\{v, w\}$. A tree decomposition of G'' can be obtained from any tree decomposition of G' by replacing each occurrence of v in a bag by w if v and w are adjacent in G', and by removing all occurrences of v if v and w are twins in G'; hence $tw(G'') \leq tw(G')$. On the other hand, $tw(G') \leq tw(G'') + 1$, since a tree decomposition of G' can be obtained from

any tree decomposition of G'' by replacing each occurrence of w in a bag by occurrences of both v and w — we will say that w is *expanded*. If G' is of bounded treewidth, we can therefore undo the reduction transforming G' into G'' by applying the width-minimizing procedure of Theorem 2.3 to derive a minimum-width tree decomposition of G' from one of G''.

For a fast parallel algorithm it clearly does not suffice to remove vertices one by one. It is easy to see, however, that the scheme described in the preceding paragraph remains valid if, rather than reducing on a single pair of vertices, we reduce simultaneously on an arbitrary collection of pairs that are sufficiently far apart in the graph not to interfere with each other. The only difference is that the treewidth of G' may now be as much as twice that of G'', plus one (each vertex in a bag may need to be expanded into two vertices), which is still fine for the width-minimizing procedure.

Theorem 4.1 *For all constants $k \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O((\log n)^2)$ time, $O(n)$ operations and $O(n)$ space: Given an n-vertex graph G, construct a minimum-width tree decomposition of G or decide (correctly) that $tw(G) > k$.*

Proof: For the time being assume that G is connected and of treewidth at most k. We will apply Lemma 3.1 to G with $n_{\min} = 2$. Hence let the constants c and d be as in the lemma and define the concepts *small* and *acquainted* accordingly. The lemma implies that G contains at least $cn/2$ distinct pairs $\{v, w\}$ of small vertices such that v and w are either adjacent or acquainted twins; to see this, note that each of the cn disjoint plains whose existence is guaranteed by the lemma contains either a valley of at least two vertices, hence a small vertex with a small neighbor (either the small neighbor also belongs to the plain, or it is one of its boundary vertices), or two or more acquainted twin valleys of one vertex each. Furthermore, the set R of all such pairs can be computed in constant time using $O(n)$ operations, since it suffices to let each small vertex inspect all its neighbors and all vertices with which it is acquainted; with some care, this can be done without concurrent reading.

We cannot necessarily execute all reductions corresponding to pairs in R, since vertices in distinct pairs may coincide, be adjacent or have adjacent entries in some adjacency list, which hinders the simultaneous execution of the associated reductions. In order to deal with this complication, we construct a *conflict graph* with a vertex for each pair in R and an edge between two vertices if the corresponding reductions exclude each other for one of the reasons mentioned above. It is easy to see that the conflict graph is of bounded degree and can be constructed in constant time using $O(n)$ operations. Following [19], we define a *fractional independent set* in an m-vertex graph H as an independent vertex set in H of size at least ϵm, where ϵ is an (unspecified) positive constant. We proceed to compute a fractional independent set in the conflict graph, which can be done

in $O(\log n)$ time using $O(n)$ operations [24, Lemma 7(b)]. Finally we execute the reductions on the pairs in the independent set, which takes constant time and uses $O(n)$ operations.

The reductions described above change G into a smaller graph G'. Let us now see that we can undo the reductions in the sense of deriving a minimum-width tree decomposition of G from one of G'. We already observed that all that is involved is to expand certain vertices into the corresponding pair of reduction partners, after which we can finish using the width-minimizing procedure of Theorem 2.3. Allowing concurrent reading, the task would be trivial — processors collectively inspecting the whole tree decomposition could simply expand each such vertex after looking up its partner in a table. In order to avoid concurrent reading from the table, we begin by balancing the given tree decomposition of G' (Lemma 2.2); this may increase its width, but only by a constant factor. We then process the resulting balanced tree decomposition $(T = (X, F), \{U_x \mid x \in X\})$ in topological order, i.e., each node is processed before all of its children. The processing of a node x in T expands all vertices in U_x that need to be expanded. If x is the root of T, this is easy. If not, the identity of the reduction partner of each relevant vertex $v \in U_x$ can be passed to x from its parent y, except if v occurs in U_x for the first time (i.e., if $v \notin U_y$). For each vertex v the latter happens only at a single tree node x, however, so that in this case we can use table lookup to find the reduction partner of v without any risk of concurrent reading. The balanced tree decomposition can be processed as described in $O(\log n)$ time using $O(n)$ operations.

The graph G' derived from G is connected and of treewidth at most k, so that a new batch of reductions can be applied to G'. Since G' is smaller than G by a constant factor, as measured by the number of vertices, $O(\log n)$ successive stages of simultaneous reductions suffice to reduce the input graph to a graph of constant size. Provided that the representation of the graph at hand is compacted after each stage by means of prefix summation, the number of operations and the space needed decrease geometrically over the stages, so that the whole process uses $O((\log n)^2)$ time, $O(n)$ operations and $O(n)$ space. Undoing the reductions is no more expensive. This proves Theorem 4.1 for connected input graphs of treewidth at most k.

Suppose now that the input graph G is of treewidth at most k, but not connected. Our approach will be to apply the algorithm developed above not to G, but to an auxiliary connected graph H obtained from G by introducing a new vertex r and an edge between r and a single vertex in each connected component of G. Except in the trivial case in which G has no edges, G and H have the same treewidth, so that a minimum-width tree decomposition of G can be obtained from a minimum-width tree decomposition of H by removing the occurrences of r from all bags. In order to select a vertex from each connected component of G, we can apply the first part of the reduction algorithm to G in a preprocessing

phase: Each connected component of G, being of treewidth at most k, is reduced to constant size, at which point the selection is easy, and the component can be removed (since its size may not decrease any further, keeping it around might make subsequent stages too expensive).

If the treewidth of the input graph G is larger than k, one or more of its connected components may fail to be reduced to constant size within the time bound established for graphs of treewidth at most k, or one of the intermediate graphs encountered while undoing reductions may have treewidth larger than k. In either case, the algorithm can stop and announce that $tw(G) > k$. Finally, it is easy to see from the description of the algorithm that even if $tw(G) > k$, the algorithm never performs an illegal action such as concurrent reading, and any output produced by the algorithm is a correct minimum-width tree decomposition of G. □

By applying first the algorithm of Theorem 4.1 and then that of Theorem 2.5, we obtain the result below.

Corollary 4.2 *For all constants $k \geq 1$ and all integers $n \geq 2$, the following problem can be solved on an EREW PRAM using $O((\log n)^2)$ time, $O(n)$ operations and $O(n)$ space: Given an n-vertex graph G, construct a minimum-width path decomposition of G or decide (correctly) that the pathwidth of G is larger than k.*

5 Deciding treewidth on the EREW PRAM

An important bottleneck for the running time of the algorithm in the previous section is the repeated application of the algorithm of Theorem 2.3 while undoing the reductions. When we aim for a decision algorithm only, we can follow a different approach: We will not undo reductions, but instead make sure that all reductions preserve treewidth. We actually describe a generic algorithm, whose instantiations solve various decision problems on graphs of bounded treewidth; in the more general setting, reductions must not affect membership in the class of graphs to be recognized.

Our algorithm can be viewed as a parallelization of a linear-time sequential algorithm due to Arnborg *et al.* [6]. A first parallel version of this algorithm was given in [9]. The algorithm described there is randomized, works only for graphs of bounded degree and uses $O(\log n)$ expected time and $O(n \log n)$ expected operations on n-vertex input graphs. The algorithm given in this section works for arbitrary graphs, uses $O(n)$ operations and is deterministic, but at a cost of an extra factor of $O(\log^* n)$ in the running time. The algorithm of [6] uses an amount of space bounded by a polynomial, but a polynomial whose degree is large and unspecified. We reduce this to $O(n)$ by means of the bounded adjacency-list search technique.

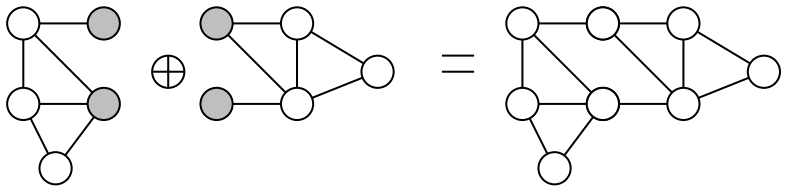

Figure 2: Combination of two terminal graphs using $\oplus$.

A *terminal graph* is a triple $G = (V, E, Z)$, where (V, E) is a graph and $Z \subseteq V$ is an ordered set of distinguished vertices in G. The vertices in Z and those in $V \backslash Z$ are called the *terminals* and the *internal vertices* of G, respectively. A terminal graph is *open* if there are no edges between terminals. For $l \geq 0$, an *l-terminal graph* is a terminal graph with exactly l terminals. Let $\mathcal{H}_l$ denote the class of l-terminal graphs, for $l \geq 0$.

Given two l-terminal graphs G_1 and G_2, for some $l \geq 0$, we define $G_1 \oplus G_2$ as the graph obtained by taking the disjoint union of G_1 and G_2 and then identifying the ith terminals in G_1 and G_2, for $i = 1, \ldots, l$. An example is shown in Fig. 2. (When there is an edge between a pair of terminals in both G_1 and G_2, we take just a single edge between these in $G_1 \oplus G_2$.)

Let $\mathcal{G}$ be a class of graphs. We define an equivalence relation $\sim_{\mathcal{G}}$ on the set of terminal graphs as follows: $G_1 \sim_{\mathcal{G}} G_2$ if and only if for some l, G_1 and G_2 both have l terminals, and for all $H \in \mathcal{H}_l$, we have $G_1 \oplus H \in \mathcal{G}$ if and only if $G_2 \oplus H \in \mathcal{G}$. Informally, G_1 and G_2 are equivalent under $\sim_{\mathcal{G}}$ if any occurrence of G_1 in a bigger graph can be replaced by an occurrence of G_2 without affecting membership of the bigger graph in $\mathcal{G}$. We say that a class $\mathcal{G}$ or its defining property P (i.e., $G \in \mathcal{G}$ if and only if $P(G)$) is of *finite index* if, for every $l \geq 0$, $\mathcal{H}_l$ is split into a finite number of equivalence classes under $\sim_{\mathcal{G}}$. (Graph properties of finite index are also known as being *regular* or of *finite state*.) Many important properties are known to be of finite index.

Theorem 5.1 *For every graph property P of finite index and for all constants $k \geq 1$ and all integers $n \geq 2$, the problem of deciding whether $P(G) \wedge (tw(G) \leq k)$ for an n-vertex input graph G can be solved on an EREW PRAM using $O(\log n \log^* n)$ time, $O(n)$ operations and $O(n)$ space.*

Proof: Assume first that $P(G)$ implies that G is connected. It was shown in [32] that the class of graphs of treewidth at most k is of finite index, and one easily observes that finite index is closed under intersection (see, e.g., [12]). Hence $\mathcal{G} = \{G \mid P(G) \wedge (tw(G) \leq k)\}$ is of finite index. Let $\mathcal{R}$ be a finite set of open terminal graphs that contains at least one element of each equivalence class of $\sim_{\mathcal{G}}$ comprising one or more open terminal graphs with at most $2(k+1)$ terminals, and take $n_{\min}$ as one more than the largest number of vertices of any graph in $\mathcal{R}$. By Lemma 3.1, we can choose $d, n_{\max} \geq 1$ and $c > 0$ such that G contains at

least cn disjoint $(d, k, n_{\min}, n_{\max})$-plains (for any adjacency-list representation of G).

The significance of $n_{\min}$ is that any open terminal graph with at least $n_{\min}$ vertices and at most $2(k+1)$ terminals has a smaller equivalent terminal graph in $\mathcal{R}$. In particular, each plain H together with its boundary B and all edges joining a vertex in H and a vertex in B, with the vertices in B considered as terminals (call this an *extended plain*), is such an open terminal graph, so that it can be replaced by a smaller terminal graph in $\mathcal{R}$. Considering isomorphic graphs as identical, there is only a finite number of different extended plains, all of which can therefore be mapped to equivalent smaller open terminal graphs by means of a finite table T. Each entry in T corresponds to a reduction in a natural way.

The algorithm proceeds in a number of phases. In each phase, each vertex determines whether it belongs to a plain and, if so, looks up a corresponding reduction in T. This can be done in constant time: It suffices to let each vertex u inspect those vertices and edges that lie on a path of length at most $2n_{\max}$ from u such that the entries of any two consecutive edges (v, w) and (w, x) on the path are separated by a distance of at most d in the adjacency list of w; this can be done without concurrent reading. The reductions found by two distinct vertices may not be simultaneously executable: The plain containing one vertex may intersect the plain containing the other vertex or its boundary, or two vertices, one from each plain, may have adjacent entries in some common boundary vertex. Because we only replace open terminal graphs by other open terminal graphs, however, these are the only ways in which two reductions can interfere with each other. As in Section 4, we construct a conflict graph of bounded degree on the vertices belonging to plains, compute a fractional independent set in the conflict graph and execute the corresponding reductions, which reduces the size of the graph by at least a constant factor. After $O(\log n)$ stages, either we are left with a graph of constant size, whose membership in $\mathcal{G}$ can be determined directly, or the input graph did not belong to $\mathcal{G}$.

The only part of a stage that takes more than constant time with a linear number of processors is the computation of a fractional independent set in the conflict graph. For this, we employ in the first $O(\log^* n)$ stages the algorithm of [24, Lemma 7(b)], which uses $O(\log m)$ time and $O(m)$ operations, where m is the number of vertices in the conflict graph. In the remaining phases, we use the algorithm of [23, Theorem 4], which needs $O(\log^* n)$ time and $O(m \log^* n)$ operations. The total time is $O(\log n \log^* n)$, and a simple simulation argument that schedules compactions of the representation conveniently (see [26, Section 4]) shows that the algorithm can be carried out using $O(n)$ operations.

Dropping the assumption that $P(G)$ implies that G is connected, we can still proceed as described above, provided that we remove and save each connected component with fewer than $n_{\min}$ vertices as soon as it arises. After $O(\log n)$ stages, either the input graph has been reduced to an equivalent collection of

connected graphs, each of which contains fewer than $n_{\min}$ vertices, or it did not belong to $\mathcal{G}$. Assume the former. In constant time, a single processor can *combine* two graphs in the collection, i.e., replace them by a single graph equivalent to their union and containing fewer than $n_{\min}$ vertices. By means of a tree-structured combination process that uses $O(\log n)$ time and $O(n)$ operations, we can therefore reduce the input graph to a single equivalent graph with fewer than $n_{\min}$ vertices, for which membership in $\mathcal{G}$ can be decided directly. □

The theorem implies, in particular, that the problem of deciding whether the treewidth of a given graph is at most k, for constant k, can be solved in $O(\log n \log^* n)$ time with $O(n)$ operations. Moreover, the same result can be shown to hold for pathwidth. Many well-known graph properties are of finite index. For instance, this is true of all problems that can be expressed in monadic second-order logic, such as Hamiltonicity and l-colorability. This was first shown by Courcelle [18]; see [12] for a possibly more accessible proof.

Theorem 5.1 is nonconstructive: An algorithm with the stated properties is merely shown to exist. To actually exhibit the algorithm, we must be able to compute the number $n_{\min}$ and to construct the table T. If we have a terminating algorithm that decides whether two given terminal graphs are equivalent under $\sim_{\mathcal{G}}$ or under some refinement (subdivision) of $\sim_{\mathcal{G}}$ that still has a finite number of equivalence classes, this can be done by a method described in [6] (in a general algebraic setting). For the case in which $\mathcal{G}$ is the class of all graphs of treewidth at most k, such an explicit decision algorithm was exhibited in [32]. If $\mathcal{G}$ is the set of those graphs of treewidth at most k that satisfy a property P expressed in monadic second-order logic, then an algorithm that decides a subdivision of $\sim_{\mathcal{G}}$ with a finite number of equivalence classes can be obtained by combining results implicit in [12, 18, 32].

It is also possible to apply the parallel reduction techniques to problems that are of *finite integer index*, in the sense of [9]. This allows deciding on the size of a maximum independent set, minimum vertex cover, minimum dominating set and others on graphs of bounded treewidth in $O(\log n \log^* n)$ time using $O(n)$ operations on an EREW PRAM. Using the technique of [9, Section 6.1], it is also possible to construct corresponding solutions for some of these problems.

6 Deciding treewidth on the CRCW PRAM

In this section we show how the running time of $O(\log n \log^* n)$ of the algorithm in the previous section can be reduced to $O(\log n)$ if we move to the stronger CRCW PRAM. Among the many variants of the CRCW PRAM, we employ one that allows m processors to compute the OR of m bits in constant time using $O(m)$ space, for all integers $m \geq 1$; this requirement excludes none of the CRCW PRAM variants commonly considered. We assume an instruction set that includes unit-

time binary left and right shifts of words of $O(\log n)$ bits by amounts specified in a second word.

As concerns its running time, the EREW PRAM algorithm has two bottlenecks: First, as dictated by efficiency considerations, the representation of the graph at hand must be compacted $\Theta(\log^* n)$ times, with each compaction taking logarithmic time. Second, in each of $\Theta(\log n)$ stages a fractional independent set is found in a conflict graph of bounded degree, for which we spend $\Theta(\log^* n)$ time per stage. Moving to the CRCW PRAM, we can easily eliminate the first bottleneck, since in this model compaction can be done in $\Theta(\log n/\log\log n)$ time [17], rather than the $\Theta(\log n)$ time for the EREW PRAM. Before attacking the second bottleneck, let us observe that we can execute $\Theta(\log n/\log^* n)$ stages of the EREW PRAM algorithm without exceeding a time bound of $O(\log n)$. After compacting once, we can then associate $2^{\Omega(\log n/\log^* n)}$ processors with each remaining vertex in the graph, We will express this by saying that we have a *processor advantage* of $2^{\Omega(\log n/\log^* n)}$, which is much more than what we need in the following.

The remaining problem is to finish the computation in $O(\log n)$ time making use of the large processor advantage mentioned above, which we will do by means of derandomization. The task is, for a positive integer $m \leq n$, to compute a fractional independent set I in an m-vertex graph of bounded degree in constant time. Observe that there is a very simple randomized algorithm for obtaining I: Each vertex picks a random bit uniformly from $\{0,1\}$ and independently of other vertices and then steps into I exactly if it picked a 1, while each of its neighbors picked a 0. Although this formulation assumes that the vertices make independent choices, it is easy to see that much less will also do. If each vertex v has at least a constant probability of stepping into I, then the expected size of I is $\Omega(m)$, so that, obviously, at least one possible execution of the randomized algorithm will result in $|I| = \Omega(m)$. Whether v steps into I, however, is a function only of the random bits picked by v and by its neighbors, i.e., it suffices to guarantee d-wise independence, where d is one more than the maximum degree of the graph. In the case of perfect d-wise independence, the probability that v steps into I is at least 2^{-d}. Since we can allow any positive constant here instead of 2^{-d}, however, we can relax the requirements even more. For $\epsilon > 0$, random bits $X_1, \ldots, X_m$ are said to be (ϵ, d)-*independent* [3] if for all positive integers $l \leq d$, all distinct integers $i_1, \ldots, i_l$ with $1 \leq i_1, \ldots, i_l \leq m$ and all $b_1, \ldots, b_l \in \{0,1\}$, the probability of the event $X_{i_1} = b_1, X_{i_2} = b_2, \ldots, X_{i_l} = b_l$ deviates from 2^{-l} by at most ϵ (d-wise independence is the special case $\epsilon = 0$). It is easy to see that (ϵ, d)-independent random bits, where $\epsilon = 2^{-d-1}$, suffice for our purpose. We now appeal to Theorem 2 of [3], which promises that m $(2^{-d-1}, d)$-independent random bits can be drawn from a sample space of size $(\log m)^{O(1)}$ (where the exponent depends on d); we argue separately in Lemma 6.2 below that the computation of the m bits can be carried out in constant time with m processors.

Since our processor advantage is much bigger than polylogarithmic in n,

we can use the limited-randomness algorithm developed above and simulate all $(\log m)^{O(1)} = (\log n)^{O(1)}$ possible executions of it in parallel. We know that at least one execution will be good, in the sense that it will lead to an independent set I of size $\Omega(m)$. We would like simply to pick a good execution, but this is not entirely trivial, since we have only constant time per stage, which is not sufficient for computing the size of I. Using the deterministic approximate-summation algorithm of [25, Theorem 3], we can compute the size of I, up to a constant factor (which is sufficiently accurate), in $O((\log\log n)^3)$ time. While this is fast, it is not fast enough. We overcome this using a technique of [25], namely to simulate all possible executions of the randomized algorithm not just for one stage at a time, but for $\Theta((\log\log n)^3)$ consecutive stages, after which we can spend $O((\log\log n)^3)$ time determining a good execution without violating our time bound (as much time is then spent on graph reduction as on counting). Doing this increases the size of the sample space to $(\log n)^{O((\log\log n)^3)} = 2^{O((\log\log n)^4)}$, which is still sufficiently small, in view of our larger processor advantage.

Theorem 6.1 *For every graph property P of finite index and for all constants $k \geq 1$ and all integers $n \geq 2$, the problem of deciding whether $P(G) \wedge (tw(G) \leq k)$ for an n-vertex input graph G can be solved on a CRCW PRAM using $O(\log n)$ time, $O(n)$ operations and $O(n)$ space.*

As in the case of Theorem 5.1, Theorem 6.1 is nonconstructive; see the discussion near the end of Section 5. In order to complete the proof of Theorem 6.1, we still have to show the following.

Lemma 6.2 *For all given integers $m, K \geq 2$ with $K = (\log m)^{O(1)}$ and all constant integers $d \geq 2$, m $(1/K, d)$-independent random bits can be computed in constant time on a CRCW PRAM using m processors, $O(m)$ space and a single random integer drawn from the uniform distribution over a range of size $(\log m)^{O(1)}$.*

Proof: Our construction, described below, is an elaboration of one given in [3, Theorem 2].

Without loss of generality assume that d is odd, say $d = 2t + 1$. Let r be the smallest number of the form $2 \cdot 3^i$ no smaller than $\log(m+1)$, where i is an integer, and take p as the smallest prime no smaller than $(2K(1+rt))^2$.

Let F be the set of all bit vectors of length r and denote by $\phi : \{0, \ldots, 2^r - 1\} \rightarrow F$ the function that maps each integer to its standard r-bit binary representation. Assume that F is organized into a field by means of suitable addition and multiplication operations.

Now choose a random integer h from the uniform distribution over $\{0, \ldots, p-1\}$ and compute the bit vector y of length $1+rt$ whose $(i+1)$st bit, for $i = 0, \ldots, rt$, is 0 exactly if $i + h \equiv s^2 \pmod p$ for some $s \in \{1, \ldots, p-1\}$ (i.e., if $h + i$ is a quadratic residue modulo p). Finally, for $i = 1, \ldots, m$, compute the ith output bit as the inner product modulo 2 of y with a bit vector x_i of length $1 + rt$

constructed as follows: The first bit of x_i is 1, the next r bits are those of $\phi(i)$, the next r bits are those of $(\phi(i))^3$, where the powering is done according to the multiplication in F, the next r bits are those of $(\phi(i))^5$, etc., until the last r bits, which are those of $(\phi(i))^{2t-1}$.

It is proved in [3, Proposition 2] that the inner product modulo 2 of y with any fixed bit vector of length $1+rt$ is ϵ-*biased* [3], where $\epsilon = rt/\sqrt{p}+(1+rt)/p \leq 2(1+rt)/\sqrt{p} \leq 1/K$, i.e., it takes on the values 0 and 1 with probabilities differing by at most ϵ. It then follows from [3, Lemma 2] and [2, Proposition 6.5] that the m bits output by the algorithm are indeed $(1/K, d)$-independent. What remains is to bound the resources needed by the computation.

Obviously, $r = O(\log m)$ and, by Bertrand's postulate (see, e.g., [28, Thm. 418]), which asserts the existence of a prime in the range $\{s, \ldots, 2s\}$ for every positive integer s, we obtain that $p = (\log m)^{O(1)}$, so that the random integer h is indeed chosen from a range as small as claimed in the lemma. Because of the small size of r and p, it is easy to compute these quantities, as well as the vector y, in constant time by brute force that amounts to trying out all possibilities.

In order to construct the vectors $x_1, \ldots, x_m$, we need to implement the multiplication operation in the field F. We define the product of the vectors $(a_{r-1}, a_{r-2}, \ldots, a_0)$ and $(b_{r-1}, b_{r-2}, \ldots, b_0)$ as $(c_{r-1}, c_{r-2}, \ldots, c_0)$, where $\sum_{i=0}^{r-1} c_i x^i$ is the remainder polynomial obtained by dividing the product $(\sum_{i=0}^{r-1} a_i x^i)(\sum_{i=0}^{r-1} b_i x^i)$ by the fixed polynomial $f(x) = x^r + x^{r/2} + 1$ over the 2-element field $\mathbb{Z}_2$. Since $f(x)$ is irreducible over $\mathbb{Z}_2$ [33, Exercise 3.96], it is well-known that this multiplication operation (together with componentwise addition over $\mathbb{Z}_2$) indeed turns F into a field.

Compute q as a positive integer with $q \leq (\log m)/5$, but $q = \Omega(\log m)$. We can implement addition and multiplication over $\mathbb{Z}_2$ of polynomials of degree less than q, represented by bit vectors in the obvious way, by means of table lookup. In each case, we need a $(2^q - 1) \times (2^q - 1)$ table with entries in the range $\{0, \ldots, 2^{2q-1} - 1\}$. In constructing the tables, we can therefore, for each table entry and each integer in the range $\{0, \ldots, 2^{2q-1} - 1\}$, dedicate a team of $\Omega(m^{1/5})$ processors to testing whether the integer is the correct value of the entry under consideration, in which case the team will fill in that table entry. In the case of addition, the testing is trivial to do in constant time with just q processors, each of which takes care of one bit position. For multiplication, the problem reduces to computing the parities of $2q - 1$ bit sequences, each of length at most q. Since the parity of q bits can be computed in constant time with $O(m^\delta)$ processors, for arbitrary constant $\delta > 0$ (see, e.g., [36, lemma on p. 375]), we have enough processors in this case as well.

Because $r = O(q)$, addition and multiplication over $\mathbb{Z}_2$ of polynomials of degree less than r reduces to a constant number of additions and multiplications over $\mathbb{Z}_2$ of polynomials of degree less than q, so that both operations can be carried out in constant time by one processor using the tables constructed above.

In order to complete the implementation of multiplication over F, we need to describe how to compute the remainder over $\mathbb{Z}_2$ of a polynomial $a(x) = \sum_{i=0}^{l} a_i x^i$ modulo $f(x)$, where $r \leq l \leq 2r-2$. But since none of the powers $x^{r-1}, \ldots, x^{(r/2)+1}$ occur in $f(x)$, the polynomial $a(x) - (\sum_{i=r}^{l} a_i x^{i-r}) f(x)$, which obviously has the same remainder modulo $f(x)$ as $a(x)$, is of degree at most $\max\{l - r/2, r - 1\}$, so that constant time suffices to reduce the degree of the input polynomial by at least $r/2$ or below r. Doing this twice completes the computation.

The final operation that must be supported is forming the inner product modulo 2 of two bit vectors, each of length $O(r)$. This operation can easily be carried out in constant time by one processor using a table that maps each bit sequence of q bits to its parity. Before the table can be used, it is necessary to convert y from a representation with one bit per word to one with q bits per word. Again, this can be done in constant time by trying out all possibilities in parallel. □

Acknowledgment. We thank Jordan Gergov and Rajeev Raman for helpful discussions related to the proof of Lemma 6.2.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms* **10** (1989) 287–302.

[2] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms* **7** (1986) 567–583.

[3] N. Alon, O. Goldreich, J. Håstad, and R. Peralta. Simple constructions of almost k-wise independent random variables. *Random Structures and Algorithms* **3** (1992) 289–304.

[4] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In *Proc. 3rd Aegean Workshop on Computing (AWOC 1988), Springer-Verlag, Lecture Notes in Computer Science, Vol. 319*, pages 81–90.

[5] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.* **8** (1987) 277–284.

[6] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *J. ACM* **40** (1993) 1134–1164.

[7] H. L. Bodlaender. NC-algorithms for graphs with small treewidth. In *Proc. 14th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1988), Springer-Verlag, Lecture Notes in Computer Science, Vol. 344*, pages 1–10.

[8] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proc. 25th Annual Symposium on Theory of Computing (STOC 1993)*, pages 226–234. To appear in *SIAM J. Comput.*

[9] H. L. Bodlaender. On reduction algorithms for graphs with small treewidth. In *Proc. 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1993), Springer-Verlag, Lecture Notes in Computer Science, Vol. 790*, pages 45–56.

[10] H. L. Bodlaender. Improved self-reduction algorithms for graphs with bounded treewidth. *Disc. Appl. Math.* **54** (1994) 101–115.

[11] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. Technical Report RUU-CS-93-27, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, 1993. A preliminary version appeared in *Proc. 18th International Colloquium on Automata, Languages and Programming (ICALP 1991), Springer-Verlag, Lecture Notes in Computer Science, Vol. 510*, pages 544–555. To appear in *J. Algorithms.*

[12] 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** (1992) 555–581.

[13] N. Chandrasekharan. *Fast Parallel Algorithms and Enumeration Techniques for Partial k-Trees.* Ph.D. thesis, Clemson University, 1989.

[14] N. Chandrasekharan and S. T. Hedetniemi. Fast parallel algorithms for tree decomposing and parsing partial k-trees. In *Proc. 26th Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Illinois, 1988.

[15] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.* **17** (1988) 128–142.

[16] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica* **3** (1988) 329–346.

[17] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Inform. and Comput.* **81** (1989) 334–352.

[18] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inform. and Comput.* **85** (1990) 12–75.

[19] N. Dadoun and D. G. Kirkpatrick. Parallel construction of subdivision hierarchies. *J. Comput. System Sci.* **39** (1989) 153–165.

[20] M. R. Fellows and M. A. Langston. On search, decision, and the efficiency of polynomial-time algorithms. *J. Comput. System Sci.* **49** (1994) 769–779.

[21] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture, and Technology*, pages 139–156. Plenum Press, 1988.

[22] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Inform. and Comput.* **81** (1989) 32–45.

[23] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Disc. Math.* **1** (1988) 434–446.

[24] T. Hagerup. Optimal parallel algorithms on planar graphs. *Inform. and Comput.* **84** (1990) 71–96.

[25] T. Hagerup. Fast deterministic processor allocation. *J. Algorithms* **18** (1995) 629–649.

[26] T. Hagerup, M. Chrobak, and K. Diks. Optimal parallel 5-colouring of planar graphs. *SIAM J. Comput.* **18** (1989) 288–300.

[27] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizations of k-terminal flow networks and computing network flows in partial k-trees. In *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1995)*, pages 641–649.

[28] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers* (5th ed.). Oxford University Press, Oxford, 1979.

[29] S. Khuller and B. Schieber. Efficient parallel algorithms for testing connectivity and finding disjoint s-t paths in graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*, pages 288–293.

[30] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In *Proc. 3rd Aegean Workshop on Computing (AWOC 1988), Springer-Verlag, Lecture Notes in Computer Science, Vol. 319*, pages 101–110.

[31] J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proc. 31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, pages 173–182.

[32] J. Lagergren and S. Arnborg. Finding minimal forbidden minors using a finite congruence. In *Proc. 18th International Colloquium on Automata, Languages and Programming (ICALP 1991), Springer-Verlag, Lecture Notes in Computer Science, Vol. 510*, pages 532–543.

[33] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and their Applications.* Cambridge University Press, Cambridge, 1986.

[34] J. Matoušek and R. Thomas. Algorithms finding tree-decompositions of graphs. *J. Algorithms* **12** (1991) 1–22.

[35] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*, pages 478–489.

[36] P. Ragde. The parallel simplicity of compaction and chaining. *J. Algorithms* **14** (1993) 371–380.

[37] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.* **18** (1989) 594–607.

[38] B. A. Reed. Finding approximate separators and computing tree width quickly. In *Proc. 24th Annual Symposium on Theory of Computing, (STOC 1992)*, pages 221–228.

[39] N. Robertson and P. D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory Ser. B.* **63** (1995) 65–110.

[40] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* **14** (1985) 862–874.

[41] E. Wanke. Bounded tree-width and LOGCFL. *J. Algorithms* **16** (1994) 470–491.