Treewidth Computation and Kernelization in the Parallel External Memory Model

Riko Jacob[1], Tobias Lieber[1], and Matthias Mnich[2]

[1] Institute for Theoretical Computer Science, ETH Zürich, Switzerland
{rjacob,lieberto}@inf.ethz.ch
[2] Cluster of Excellence MMCI, Saarbrücken, Germany
m.mnich@mmci.uni-saarland.de

Abstract. We present a randomized algorithm which computes, for any fixed k, a tree decomposition of width at most k of any input graph. We analyze it in the parallel external memory (PEM) model that measures efficiency by counting the number of cache misses on a multi-CPU private cache shared memory machine. Our algorithm has sorting complexity, which we prove to be optimal for a large parameter range.

We use this algorithm as part of a PEM-efficient kernelization algorithm. Kernelization is a technique for preprocessing instances of size n of NP-hard problems with a structural parameter κ by compressing them efficiently to a kernel, an equivalent instance of size at most $g(\kappa)$. An optimal solution to the original instance can then be recovered efficiently from an optimal solution to the kernel. Our main results here is an adaption of the linear-time randomized protrusion replacement algorithm by Fomin et al. (FOCS 2012). In particular, we obtain efficient randomized parallel algorithms to compute linear kernels in the PEM model for all separable contraction-bidimensional problems with finite integer index (FII) on apex minor-free graphs, and for all treewidth-bounding graph problems with FII on topological minor-free graphs.

1 Introduction

Many practically relevant computational problems are NP-hard. Many decision problems on graphs become efficiently solvable in the RAM model if the input graph has bounded treewidth. By now, there is the linear time algorithm to compute a tree decomposition [1], working in the classical RAM model. This algorithm has been adapted to the PRAM model [2] and to serial models that take the caches of the memory hierarchy into account [3]. Here, we present an adaption of the algorithm that is simultaneously parallel and cache efficient, as modeled by the PEM model, which is defined in Section 1.1.

Another common technique for solving NP-hard problems is pruning easy parts of an instance in a preprocessing step. In the field of parameterized complexity this is formalized as *kernelization*. It is known that every decidable fixed-parameter tractable problem Π for a parameter κ admits a *kernelization*, which is an algorithm that in polynomial time reduces any instance of Π of size n to

J. Diaz et al.(Eds.): TCS 2014, LNCS 8705, pp. 78–89, 2014.

an equivalent instance (the *kernel*) of size $g(\kappa)$ for some computable function g. Here, equivalent means that the original instance is a "yes"-instance if and only if the kernel is. For example for the VERTEX COVER problem, Nemhauser and Trotter have shown that any graph G on n vertices can be kernelized efficiently into a graph G' on at most 2κ vertices, where κ denotes the size of a minimum vertex cover of G [4]. Recent meta-results show that large classes of combinatorial optimization problems admit kernels of linear size on planar graphs [5], and, more generally, classes of graphs excluding a fixed minor [6,7]. On the other hand, not every fixed-parameter tractable problem admits a polynomial sized kernel, unless the polynomial hierarchy collapses to the third level [5,8].

The classical view on kernelization in the RAM model is to solve an instance I of a hard problem Π in two phases: the first phase, kernelization, transforms in polynomial time the instance I of size n into a kernel I' whose size $g(\kappa)$ depends solely on the structural parameter κ. The second phase solves the problem Π on the kernel I' in time $f(g(\kappa))$ for some function f, which is often at least exponential due to a brute force algorithm. Thus, this leads in total to a running time of $\mathcal{O}(p(n) + f(g(\kappa)))$ to decide I in the RAM model.

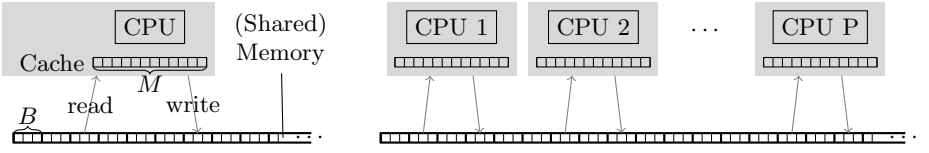
Given the abundance of practically relevant NP-hard problems where input instances are large, it would be nice to have this approach efficiently implemented. Recently, efficient kernelization algorithms have been proposed, for example by Hagerup [9] and van Bevern et al. [10], where linear time RAM algorithms are presented. Further, meta-results by Fomin et al. provide randomized linear-time kernelizations for many combinatorial optimization problems on graph classes excluding a fixed minor [6]. There are some reservations against the practicability of the algorithms implied by these meta-results, namely very large constant factors hidden in the $\mathcal{O}$-notation, and some difficulties in actually creating certain constant sized structures.

To obtain efficient algorithms on modern computers, parallelism and hierarchical memories have to be exploited. Such a parallel and memory efficient implementation is frequently possible for the second phase of kernelization. On the other hand, due to the exponential work that is done, this only moderately extends the size of I', which can be handled. We show that the randomized kernelization algorithms of Fomin et al. can be efficiently implemented in the PEM model, i.e., in parallel and taking the memory hierarchy into account. While this algorithm does not improve the large constant factors in the runtime, it shows that there are no fundamental obstacles when implementing this kind of algorithm in the PEM model. We expected that concrete algorithms for the problems covered by the meta-results can also be implemented efficiently in the PEM model.

1.1 Model of Computation: The Parallel External Memory Model

For using modern computers optimally, two architectural features require special consideration: parallelism and the memory hierarchy. To capture these two fundamental features, the *parallel external memory* (PEM) model was introduced [11]. It is based on the *external memory* (EM) model which captures

cache efficiency of algorithms [12]. The EM model extends the RAM model by a *cache*, located between CPU and memory (see Figure 1a). The cache can store up to M elements, on which the CPU can perform any computations for free. The conceptually infinite (external) memory is partitioned in blocks, each holding up to B elements. A block can be transferred as a unit by an *input/output operation* (I/O) between the cache and the memory. The memory initially holds the input of size n in the first $\lceil n/B \rceil$ blocks.



(a) The EM Model (b) The PEM Model

Fig. 1. The Sequential and the Parallel EM Model for $M = 12$ and $B = 3$

The PEM model (see Figure 1b) is a parallel extension of the EM model. It has P CPUs which all have their own (private) cache of size M. In one *parallel I/O* each processor can perform one I/O operation for moving data between its cache and the shared memory. Similar to the PRAM model, this requires a policy for simultaneous access to the same block. Here, we work with the CREW policy that allows concurrent read but disallows concurrent write. The *complexity* of an algorithm in the PEM model is the number of parallel I/Os it performs. An algorithm is called *PEM-efficient* if its complexity is matched by a lower bound for the problem in the PEM model.

The easiest non-trivial task in the (P)EM model is that of permuting: given a permutation π, create from the input $(x_1, \ldots, x_n)$ the output $(x_{\pi(1)}, \ldots, x_{\pi(n)})$. While in the (P)RAM model the complexity of permuting is the same as for scanning the input, $\Theta(n/P)$, in the (P)EM model it is not. For many settings of the parameters, the optimal solution in the PEM model for permuting is to sort in $\mathsf{sort}_P(n) = \Theta(\frac{n}{PB} \overline{\log}_{\frac{M}{B}} \frac{n}{B})$ parallel I/Os, where $\overline{\log} x$ equals $\max\{1, \log x\}$.

In the PRAM model one important goal is to obtain work optimal algorithms for a maximal number of processors. For the PEM model, one strives similarly to use as many processors as possible, while still using the optimal number of parallel I/Os. We define a *PEM-efficient kernelization* for a parameterized problem Π as an algorithm $\mathcal{A}$ that computes for any instance x of size n with parameter κ a kernel (x', κ'), and $\mathcal{A}$ is PEM-efficient. In our case this means $\mathcal{A}$ executes with $\mathcal{O}(\mathsf{sort}_P(n))$ parallel I/Os.

1.2 Related Work

A *tree decomposition* of a graph G is a tree T and for each node i of T there is a *bag* B_i of vertices of G such that (i) for each vertex v of G there is at least one

bag containing v; (ii) for each edge e of G there is at least one bag containing both its endpoints; and (iii) for each v of G, the set $\{i \mid v \in \mathcal{B}_i\}$ induces a subtree of T. The *width* of a tree decomposition is defined as the maximum bag size minus one. The *treewidth* of G, denoted by $\mathsf{tw}(G)$, is the minimum width over all tree decompositions of G.

There are several fixed-parameter algorithms to compute the treewidth of a graph in different computational models. The most important to this paper are the following: Bodlaender and Kloks introduced in 1993 the first linear time algorithm to efficiently compute a tree decomposition of fixed width k of a graph G in the RAM model [13]. Later Bodlaender and Hagerup introduced a work optimal algorithm for the PRAM model for up to $p \leq \frac{n}{\log^2 n}$ processors [2]. An I/O efficient algorithm for computing the treewidth in the EM model has been presented by Maheshwari and Zeh [3].

Recently numerous problems have been shown to admit linear kernels on generalizations of planar graphs, including graphs of bounded genus, and graphs of bounded maximum degree [6,7]. The results are briefly summarized:

Throughout the paper, let H denote a fixed graph and let $\mathcal{G}_H$ be the graph class that excludes H as a minor. For any class $\mathcal{G}_H$, where H is an apex graph (so $H \setminus \{v\}$ is planar for some vertex v), we deal with essentially all problems Π that are "contraction-bidimensional". Roughly speaking, a parameterized problem Π is contraction-bidimensional if contracting an edge cannot increase the objective value κ, and on square grids the objective value grows proportionally with the size of the vertex set of the grid.

Lemma 1 ([14]). *On any class of graphs excluding a fixed apex graph as a minor, all separable contraction-bidimensional problems with finite integer index, parameterized by solution size κ admit kernels of size $\mathcal{O}(\kappa)$.*

Graph classes excluding a fixed graph H as a minor can be generalized to classes excluding H as a topological minor. A graph problem Π, parameterized by κ is called *treewidth-bounding* if all "yes"-instances (G, κ) of Π admit an $\mathcal{O}(\kappa)$-sized set $S \subseteq V(G)$ such that the graph $G \setminus S$ has constant treewidth.

Lemma 2 ([7]). *On any class of graphs excluding a fixed graph as a topological minor, all treewidth-bounding graph problems with finite integer index, parameterized by solution size κ have a kernels of size $\mathcal{O}(\kappa)$.*

Recently, Hagerup [9] and van Bevern et al. [10] argued that not only the size $g(\kappa)$ of the produced kernel, but also lowering the running time of the kernelization algorithm is an important research direction. Another strong case for designing kernels that are as efficient as possible is made by Komusiewicz and Niedermeier [15].

Neither of the approaches considers kernelization in a context of parallel I/O algorithms, but only in the classical RAM model. Thus the approach of designing PEM-efficient algorithms for kernelization is completely new.

Most proofs are omitted due to space limitations and will be published in a full version of this paper.

1.3 Our Contributions

Our first contribution is for every fixed $k \in \mathbb{N}$ a randomized PEM-efficient algorithm which for a given graph G computes a tree decomposition of width at most k, or decides that the treewidth of G is larger than k. Our algorithm is based on the treewidth algorithm for the PRAM model [2] and the treewidth algorithm for the EM model [3]. While the first algorithm yields a framework for load balancing in treewidth computations, the latter yields an EM efficient implementation of the dynamic programming approach of Bodlaender and Kloks [1]. The combination yields a rather technical implementation of the load balancing part of Bodlaender and Hagerup, and a parallelized version of the algorithm of Maheshwari and Zeh. The fundamental building block for our algorithm, which is given in Section 2, is list ranking. Furthermore, the fundamental data structure flippable DAG for the EM model, introduced by Maheshwari and Zeh [3], is replaced by a simpler construction as it appears to be hardly parallelizable.

Theorem 3. *For every $k, t \in \mathbb{N}$, the expected number of parallel I/Os needed to compute a tree decomposition of width at most k, if such exists, for a graph G of size n in the CREW PEM model with $P \leq n/(B^2 \log B \log n \log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$ is $\mathcal{O}(\mathsf{sort}_P(n))$.*

Throughout this paper, t is a fixed constant which can be chosen arbitrary, influencing only the constant in the $\mathsf{sort}_P(n)$ term. Furthermore, it is defined $\log^{(1)} x = \log x$ and $\log^{(k)} x = \log \log^{(k-1)} x$.

Observe that the bound on the I/O complexity is tight for a wide range of the number of processors, which is shown in Section 5.

Our second contribution is a PEM-efficient implementation of the randomized fast protrusion replacer of Fomin et al. [6]. Their fast protrusion replacer works by replacing large protrusions, which are subgraphs of small treewidth and small attachment to the rest of the input graph, by smaller, constant-sized protrusions. It can be used to provide efficient randomized kernelization algorithms for the (linear) kernels mentioned in Lemma 1 and Lemma 2. Using our PEM-efficient algorithm for computing the treewidth of a graph we argue in Section 3 that this protrusion replacer can be implemented efficiently in the PEM model. Using the randomized fast protrusion replacer, in Section 4 the following theorem is shown.

Theorem 4. *The expected number of parallel I/Os to compute a linear kernel for each of the problems of Lemma 1 and Lemma 2 is $\mathcal{O}(\mathsf{sort}_P(n))$ in the CREW PEM model with $P \leq n/(B^2 \log B \log^2 n \log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$.*

By Lemma 1, PEM-efficient kernelizations to linear kernels exist, among others, for Dominating Set, Connected Dominating Set, and Induced Matching on all classes of apex-minor free graphs. Furthermore, by Lemma 2 there exist PEM-efficient parallel randomized kernelizations on H-topological-minor free graphs for, among others, Chordal Vertex Deletion, Interval Vertex Deletion, Treewidth-w Vertex Deletion, and Edge Dominating Set. Moreover, the Width-b Vertex Deletion problem admits a linear kernel in the PEM model on H-topological minor-free graphs, where the width measure is either treewidth, branchwidth, rankwidth, or cliquewidth.

2 Computing Tree Decompositions in the PEM Model

In this section we present for every k a randomized PEM-efficient algorithm to decide the Treewidth-k problem. More precisely, for every k the algorithm computes a tree decomposition of a graph G of width k if there exists one.

We briefly state the I/O-complexities of the most important algorithmic problems in the PEM model. Sorting n records can be done by a multi-way merge sort with $\mathsf{sort}_P(n) = \Theta(\frac{n}{PB} \log_d \frac{n}{B})$ parallel I/Os for $d = \max\{2, \min\{\frac{n}{PB}, \frac{M}{B}\}\}$, and $P \leq \frac{n}{B}$ [16]. The problem of ranking a list of size n is fundamental in the (P)EM model and implies an efficient algorithm for the prefix sums problem [17,18,19]. Both can be solved with $\mathsf{listRank}_P(n) = \mathsf{sort}_P(n)$ parallel I/Os if $P \leq \frac{n}{B^2 \log B \log^{(t)} n}$, and $M = B^{\mathcal{O}(1)}$ [19].

Note that we assume in this paper that all graphs are given by *edge lists* and each undirected edge $\{u, v\}$ is represented by two directed edges (u, v) and (v, u).

2.1 A Framework for Parallel Treewidth Computation

Most tree decomposition algorithms follow a recursive approach [1,2,3]: by recursive application of a reduction rule, the input graph G is reduced to constant size for which a tree decomposition of width k can be given easily. By revoking a round of the application of the reduction rules, a tree decomposition is obtained which is only slightly too large. By using dynamic programming the width of this tree decomposition can be reduced to k, again. Our algorithm follows the approach presented by Bodlaender and Hagerup [2] for dealing with the load balancing in the PRAM model. Its pseudo code, which also indicates a load balancing step, is presented in Algorithm 1.

In the following, for the three methods, `reduce`, `treeDecompostionOfAtMost`, and `balance`, the most important properties and implementation details are presented. They are presented in detail in the full version of this paper. Note that the correctness for all these algorithms follows directly from the original works [2,3].

Algorithm 1. Computing a Tree Decomposition in a parallel Model

```
1  G_0 ← G
2  r ← d · log n
3  for 1 ≤ i ≤ r do
4      G_i ← reduce(G_{i−1})
5  T_r ← treeDecompostionOfAtMost(G_r, k)
6  for r ≥ j ≥ 1 do
7      T_{j−1} ← T_j ∪ (G_{j−1} \ G_j)        // revoking changes of reduce round j
8      T_{j−1} ← balance(T_{j−1})
9      T_{j−1} ← treeDecompostionOfAtMost(T_{j−1}, k)
10 return T_0
```

The `reduce` *Method:* It was first presented by Bodlaender and Hagerup [2]. One call (in round i) to the reduce method decreases the size of the input graph by a constant fraction $\frac{1}{d} \geq \frac{1}{8^k}$ by identifying disjoint pairs of vertices, so called acquainted twins. Therefore, after $d \cdot \log n$ rounds, the resulting graph has constant size and thus a tree decomposition of constant width can be computed by brute force. Furthermore, revoking the vertex identifications of round i increases the width of the tree decomposition T_i for G_{i+1} by at most $k+1$.

The implementation is mostly a straight forward, but technical and careful, implementation of the PRAM algorithm in the PEM model. It uses rounds of local operations, and information exchange, implemented by scanning and sorting. For finding vertices which can be identified in parallel a conflict graph of bounded degree, which represents if acquainted twins can be reduced in parallel, is computed. A fractional independent set (FIS) in the conflict graph then yields the vertices which are reduced in parallel. For computing a FIS of the conflict graph we observe that the PRAM algorithm of Dadoun and Kirkpatrick [20] can be implemented efficiently in the PEM model.

Balancing a Tree Decomposition: The method `balance` obtains from an arbitrary tree decomposition $\mathcal{T} = (T, B)$ of a graph G of width k' a tree decomposition $\mathcal{T}_b = (T', B')$ of width at most $\ell = 7k'$. The most important property of $\mathcal{T}_b$ is that T' is *balanced*, meaning it is suited for processing computations on its nodes in a bottom up manner with at most $\mathcal{O}(\mathsf{listRank}_P(n))$ parallel I/Os. To this end, a contraction tree T' of T is computed. Based on the randomized algorithm for computing an independent set of Vishkin [21] an algorithm to compute a T' can be obtained with at most $\mathcal{O}(\mathsf{listRank}_P(n))$ parallel I/Os. Finally, the bags for $\mathcal{T}_b$ can be computed in a bottom-up manner on the vertices of T'.

The implementation of the `balance` method is a simplified version of the algorithm of Bodlaender and Hagerup [2], yielding $\ell = 7k'$, instead of $\ell = 3k'$. However this affects only constants hidden in the asymptotic notation.

Computing a Tree Decomposition of Width k: Bodlaender and Hagerup [2] use in their PRAM algorithm the algorithm of Bodlaender and Kloks [1] to compute from a tree decomposition of width ℓ for G, a tree decomposition of width k for a graph G. This is possible since the dynamic programming approach [1] is straight forward parallelizable.

Similar to their approach, in our algorithm, a modification of the algorithm for the EM model [3] is used to implement `treeDecompostionOfAtMost`. For efficient parallelization, the algorithm obtains a balanced tree decomposition of width ℓ for a graph of size n.

Large parts of the algorithm of Maheshwari and Zeh can be reused, since it heavily uses scanning, sorting and prefix sum computations. The most challenging part is to return an actual tree decomposition, and not a corresponding implicit (intermediate) representation. Their data structure "Flippable DAG", which represents a DAG such that its entire edge set can be flipped (implicitly) with one I/O, is used in this process fundamentally. Precisely, their procedure to extract from a flippable DAG an explicit DAG uses time forward processing

(TFP) [17]. Since it is not clear how TFP can be implemented efficiently in the PEM model we observe the following: in the case of computing a tree decomposition it is possible to store instead of one flippable DAG data structure a DAG and its "flip" explicitly. Therefore `treeDecompostionOfAtMost` does not rely on TFP anymore and can be implemented PEM-efficiently.

Analysis: The I/O-complexity of the three methods sketched in this section, `reduce`, `balance`, and `treeDecompostionOfAtMost`, are dominated asymptotically by sorting or list ranking. The methods are applied in the loops of Algorithm 1 repeatedly to different sizes of inputs, which geometrically decrease (respectively increase) from the initial input size n to constant size. Thus, the number of available processors increases relatively to the input size and the runtimes $\mathsf{sort}_P(n)$ and $\mathsf{listRank}_P(n)$ are not applicable anymore due to their processor bounds. The I/O complexity of list ranking equals the I/O complexity of sorting in a large parameter range but when applied with many processors (in relation to the input size) it dominates sorting. A detailed analysis yields the following term to capture the complexities of all loops:

$$\mathcal{O}\left(\sum_{j=0}^{\log n-(\log(PB^2\log^{(t)} n)-1)} \mathsf{sort}_P\left(\frac{n}{d^j}\right) + \sum_{\ell=0}^{\log B} B\log B\log^{(t)} n\log_{\frac{M}{B}}\frac{n}{B} + \sum_{m=0}^{\log(PB\log^{(t)} n)}\left(B\log^{(t)} n + \log B\log_{\frac{M}{B}}\frac{n}{B}\right)\right)$$

The first sum can be bound by $\mathcal{O}(\mathsf{sort}_P(n))$ since the input sizes are geometrically decreasing. The last sums can be bound by $\mathcal{O}(\mathsf{sort}_P(n))$ due to the restrictions on $P \leq n/(B^2\log B\log n\log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$, as stated by Theorem 3. This yields in total $\mathcal{O}(\mathsf{sort}_P(n))$ parallel I/Os for computing a tree decomposition. Note that improvements of the parameter range would directly imply improvements on the parameter range of this algorithm.

Furthermore, observe that the bound on P is compared to the list ranking bound decreased by another $\log n$ factor. This seems natural, since the PRAM algorithm [2] requires $P \leq \frac{n}{\log^2 n}$, opposed to $P \leq \frac{n}{\log n}$ for sorting or list ranking in the PRAM model.

3 Replacing Protrusions

Let G be a graph, and $X \subseteq V(G)$, then $\partial_G(X)$ is defined as the set of vertices in X having a neighbor in $V \setminus X$. Furthermore X is a *r-protrusion* of G if $|\partial_G(X)| \leq r$ and $\mathsf{tw}(G[X]) \leq r$.

In this section we present PEM-efficient implementations of the (randomized) large protrusion finder, and the protrusion replacer of Fomin et al. [6]. Both, protrusion finder and protrusion replacer, are be applied to n-vertex graphs G from classes $\mathcal{G}_H^{(\text{top})}$, for some fixed graph H. Thus, all handled graphs G

have $E(G) = \mathcal{O}(|V(G)|)$ edges, and our implementations run with $\mathcal{O}(\mathsf{sort}_P(|E(G)|)) = \mathcal{O}(\mathsf{sort}_P(n))$ parallel I/Os.

There is only one reduction rule which is applied repeatedly on the input graph G to reduce its size. This rule was introduced by Fomin et al. [14]:

If G has a τ-protrusion X, then X is replaced by a constant sized graph $Y \in \mathcal{R}_{\Pi,\tau}$, which is equivalent to X with respect to a relation $\equiv_{\Pi,\tau}$. ($\star$)

An instance (G, κ) is called *reduced* if the reduction rule cannot be applied anymore to it. By [7, Lemma 1] for any parameterized problem Π which has finite integer index (FII) in a graph class $\mathcal{G}$ there exists for every fixed $\tau \in \mathbb{N}$ such a finite set $\mathcal{R}_{\Pi,\tau}$ of representatives for the problem Π restricted to the graph class $\mathcal{G}$. The safety of the reduction rule ($\star$) is proven in [7, p. 620].

Since $\mathcal{R}_{\Pi,\tau}$ depends on τ, our kernelization algorithm (as well as the known polynomial-time kernelizations of [7,6]) is non-uniform in κ. On the other hand, in the following we may assume that $\mathcal{R}_{\Pi,\tau}$ is known explicitly to the algorithms.

By definition, $\mathcal{R}_{\Pi,\tau}$ is finite for every fixed τ and for every Π which has FII. Thus, the *protrusion limit* of Π restricted to a graph class $\mathcal{G}$ is well-defined as $\rho_{\Pi,\mathcal{G}}(\tau) = \max_{G \in \mathcal{R}_{\Pi,\tau}} |V(G)|$ [7].

The key result to be proven in this section is a PEM-efficient implementation of the reduction rule ($\star$). To this end, a randomized PEM-efficient protrusion finder yielding a set $\mathcal{P}$ of protrusions, and a PEM-efficient protrusion replacer, replacing constant sized protrusions, is presented. Since not all protrusions of $\mathcal{P}$ are of constant size, in Lemma 7, a PEM-efficient algorithm for replacing protrusions of unbound size, by constant sized protrusions, is presented.

3.1 PEM-Efficient Protrusion Finder

The protrusion finder is an algorithm that finds τ-protrusions such that every τ-protrusion X has size at least $\rho_{\Pi,\mathcal{G}}(2\tau)$. The complexity of our PEM-efficient protrusion finder is the following:

Lemma 5. *The randomized fast protrusion finder of Fomin et al. [6], for a input graph G of size n, can be implemented in the CREW PEM model with $P \leq n/(B^2 \log B \log n \log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$ taking an expected number of $\mathcal{O}(\mathsf{sort}_P(n))$ parallel I/Os. It yields a set $\mathcal{P}$ of τ-protrusions such that for every $X \in \mathcal{P}$ holds $|X| > \rho_{\Pi,\mathcal{G}}(2\tau)$.*

The proof of Lemma 5 is given in the full version of this paper. The algorithm is a straight forward implementation of the algorithm of Fomin et al. with special considerations on load balancing, using the connected components algorithm for graphs which are sparse under contraction of Arge et al. [19] and our new algorithm for tree decompositions (Section 2).

3.2 PEM-Efficient Protrusion Replacer

In this section, based on the ideas of Bodlaender et al. [22], a PEM-efficient algorithm is given for replacing all τ-protrusions in parallel.

We first deduct an algorithm to replace constant sized τ-protrusions. For this task, there is a constant time algorithm ([23, Lemma 5]) in the RAM model. A trivial simulation of this algorithm requires not more than $\mathcal{O}(1)$ I/Os, which yields the following lemma:

Corollary 6. *Let H be a graph and let Π be a parameterized graph problem with finite integer index in $\mathcal{G}_H^{(\text{top})}$. If for $\tau \in \mathbb{N}$ the set $\mathcal{R}_{\Pi,\tau}$ of representatives of $\equiv_{\Pi,\tau}$ is given, then for any τ-protrusion Y of size at most c one can decide with $\mathcal{O}(1)$ I/Os which representative $G' \in \mathcal{R}_{\Pi,\tau}$ satisfies $G' \equiv_{\Pi,\tau} G[Y]$, where the hidden constants depend only on τ and c.*

Recall that the protrusion finder of Fomin et al. finds a collection $\mathcal{P}$ of τ-protrusions X of size $|X| > \rho_{\Pi,\mathcal{G}}(2\tau)$. Since Corollary 6 can handle only protrusions of constant size, a PEM-efficient implementation of an algorithm [6] is presented, which replaces protrusions which are larger than $2\rho_{\Pi,\mathcal{G}}(2\tau)$ by smaller $(2\tau+1)$-protrusions.

Since $\mathcal{R}_{\Pi,\tau}$ exists for a problem Π for all τ, the resulting protrusions can then be replaced by Corollary 6.

Lemma 7. *Let $\tau \in \mathbb{N}$ and let Π be a graph problem with finite integer index on a graph class $\mathcal{G}$. Given for a graph $G \in \mathcal{G}$, of size n, a set of τ-protrusions $\mathcal{P}$, the expected number of parallel I/Os to find for all $X \in \mathcal{P}$ with $|X| > \rho_{\Pi,\mathcal{G}}(2\tau)$, a $(2\tau+1)$-protrusion $Y \subseteq X$ satisfying $\rho_{\Pi,\mathcal{G}}(2\tau) < |Y| < 2\rho_{\Pi,\mathcal{G}}(2\tau)$ is $\mathsf{sort}_P(n)$ in the CREW PEM model with $P \leq n/(B^2 \log B \log n \log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$.*

The algorithm of Fomin et al. [24] used for Lemma 7 can be implemented by evaluating a tree expression evaluation on the nodes of a tree decomposition.

4 Applying the Protrusion Replacer

The randomized fast protrusion replacer of Fomin et al. [6], respectively its PEM-efficient implementation, does not yield a kernelization yet. By Fomin et al. [24, Theorem 10] one application, transforming G into G', reduces, with high probability, the size of G by at least a constant fraction $r > 0$. Thus, for obtaining a linear kernel it has to be applied $\mathcal{O}(\log n)$ times. We restate Theorem 4 and prove it.

Theorem 4. *The expected number of parallel I/Os to compute a linear kernel for each of the problems of Lemma 1 and Lemma 2 is $\mathcal{O}(\mathsf{sort}_P(n))$ in the CREW PEM model with $P \leq n/(B^2 \log B \log^2 n \log^{(t)} n)$ and $M = B^{\mathcal{O}(1)}$.*

Proof. The value τ of the randomized fast protrusion replacer is chosen depending on κ as described by Fomin et al. [6] and Kim et al. [7], respectively.

Using the I/O complexities given by Lemma 5, Lemma 7, and Corollary 6 the complexity of the algorithm is split up in a term when the number of processors is bigger than the processor bound of Lemma 5 and a term capturing the part

in which the lemma does not provide optimal bounds for the remaining graphs of size at most $x = PB^2 \log B \log n \log^{(t)} n$:

$$\sum_{i=0}^{\log \frac{n}{x}-1} \mathsf{sort}_P\left(\frac{n}{r^i}\right) + \sum_{j=\log \frac{n}{x}}^{\log n} \mathsf{sort}_P(x) \leq \mathcal{O}\left(\mathsf{sort}_P(n)\right) + \mathsf{sort}_P(x) \log n$$

The second term is in $\mathcal{O}\left(\mathsf{sort}_P(n)\right)$, since $\frac{x}{PB} \log n$ is in $\mathcal{O}\left(\frac{n}{PB}\right)$ by the processor bound $P \leq n/(B^2 \log B \log^2 n \log^{(t)} n)$. Since the sorting terms are geometrically decreasing, the first term is in $\mathcal{O}\left(\mathsf{sort}_P(n)\right)$, yielding the theorem. □

5 A Permuting Lower Bound

To show a lower bound in the PEM model, we need a standard indivisibility assumption [12]. More precisely, we assume that the node-identifiers that are used to describe the edges of the graph are atomic [25].

A well known problem to which many computational problems can be reduced to is the Proximate Neighbors problem [17]. The input of the Proximate Neighbors problem are $2n$ atoms representing $x_1, \ldots, x_n, y_{\pi(1)}, \ldots, y_{\pi(n)}$. For solving the problem, a program which may be perfectly adapted to the permutation π, has to move the pair x_i, y_i into the same main memory at some time (and compare them on some CPU). Chiang et al. [17] have shown a permuting lower bound on the number of I/Os needed to solve this task in the EM model. Despite the absence of a speed up theorem [16] the lower bound can be extended to the PEM model [25].

Hence, for reasonably big B, this justifies sorting for solving the proximate neighbors problem. Based on this result the following lower bound can be obtained easily:

Theorem 5. *Every randomized algorithm that can decide if a connected planar input graph is a tree must use* $\Omega\left(\min\{n/P, \mathsf{sort}_P(n)\}\right)$ *parallel I/Os.*

A connected graph G is a tree if and only if G has a tree-decomposition of width 1. This yields a lower bound of $\Omega\left(\min\{n/P, \mathsf{sort}_P(n)\}\right)$ parallel I/Os for tree decomposition algorithms. Similarly, the problem Treewidth-1 Vertex Deletion on a planar graph, for the special case that there is no vertex deletion necessary has the same bound. This problem is covered by Lemma 1 and Lemma 2. Thus, we have matching upper and lower bounds for a large parameter range of Theorem 3 and Theorem 4.

References

1. Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. J. Algorithms 21(2), 358–402 (1996)
2. Bodlaender, H.L., Hagerup, T.: Parallel algorithms with optimal speedup for bounded treewidth. SIAM J. Comput. 27(6), 1725–1746 (1998)
3. Maheshwari, A., Zeh, N.: I/O-efficient algorithms for graphs and bounded treewidth. Algorithmica 54(3), 413–469 (2009)

4. Flum, J., Grohe, M.: Parameterized complexity theory. Springer (2006)
5. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. J. Comput. System Sci. 75(8), 423–434 (2009)
6. Fomin, F.V., Lokshtanov, D., Misra, N., Saurabh, S.: Planar $\mathcal{F}$-deletion: Approximation, kernelization and optimal FPT algorithms. In: Proc. FOCS 2012, pp. 470–479 (2012)
7. Kim, E.J., Langer, A., Paul, C., Reidl, F., Rossmanith, P., Sau, I., Sikdar, S.: Linear kernels and single-exponential algorithms via protrusion decompositions. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 613–624. Springer, Heidelberg (2013)
8. Drucker, A.: New limits to classical and quantum instance compression. In: Proc. FOCS 2012, pp. 609–618 (2012)
9. Hagerup, T.: Simpler linear-time kernelization for planar dominating set. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 181–193. Springer, Heidelberg (2012)
10. van Bevern, R., Hartung, S., Kammer, F., Niedermeier, R., Weller, M.: Linear-time computation of a linear problem kernel for dominating set on planar graphs. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 194–206. Springer, Heidelberg (2012)
11. Arge, L., Goodrich, M.T., Nelson, M., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: Proc. SPAA 2008, pp. 197–206 (2008)
12. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
13. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
14. Fomin, F.V., Lokshtanov, D., Saurabh, S., Thilikos, D.M.: Bidimensionality and kernels. In: Proc. SODA 2010, pp. 503–510 (2010)
15. Komusiewicz, C., Niedermeier, R.: New races in parameterized algorithmics. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 19–30. Springer, Heidelberg (2012)
16. Greiner, G.: Sparse Matrix Computations and their I/O Complexity. Dissertation, Technische Universität München (2012)
17. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. SODA, pp. 139–149 (1995)
18. Katriel, I., Meyer, U.: Elementary graph algorithms in external memory. In: Meyer, U., Sanders, P., Sibeyn, J.F. (eds.) Algorithms for Memory Hierarchies. LNCS, vol. 2625, pp. 62–84. Springer, Heidelberg (2003)
19. Arge, L., Goodrich, M.T., Sitchinava, N.: Parallel external memory graph algorithms. In: IPDPS, pp. 1–11 (2010)
20. Dadoun, N., Kirkpatrick, D.G.: Parallel construction of subdivision hierarchies. J. Comput. Syst. Sci. 39(2), 153–165 (1989)
21. Vishkin, U.: Randomized speed-ups in parallel computation. In: Proc. STOC, pp. 230–239 (1984)
22. Bodlaender, H.L., Fomin, F.V., Lokshtanov, D., Penninkx, E., Saurabh, S., Thilikos, D.M. (Meta) kernelization. In: Proc. FOCS, pp. 629–638 (2009)
23. Kim, E.J., Langer, A., Paul, C., Reidl, F., Rossmanith, P., Sau, I., Sikdar, S.: Linear kernels and single-exponential algorithms via protrusion decompositions. CoRR abs/1207.0835 (2012)
24. Fomin, F.V., Lokshtanov, D., Misra, N., Saurabh, S.: Planar $\mathcal{F}$-deletion: Approximation and optimal FPT algorithms. CoRR abs/1204.4230 (2012)
25. Jacob, R., Lieber, T., Sitchinava, N.: On the complexity of list ranking in the parallel external memory model. In: Proc. MFCS (to appear, 2014)