On Optimal Algorithms for List Ranking in the Parallel External Memory Model with Applications to Treewidth and other Elementary Graph Problems

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by

Tobias Lieber
M.Sc., Technische Universität München
born 14.05.1985
citizen of Germany

accepted on the recommendation of

Prof. Dr. Peter Widmayer, examiner
Riko Jacob, PhD, co-examiner
Prof. Dr. Norbert Zeh, co-examiner

2014
Abstract

The performance of many algorithms on large input instances substantially depends on the number of triggered cache misses instead of the number of executed operations. This behavior is captured by the external memory model in a natural way. It models a computer by a fast cache of bounded size and a conceptually infinite (external) memory. In contrast to the classical RAM model, the complexity measure is the number of cache lines transferred between the cache and the memory. Computations on elements in the cache are not counted. Recent trends in processor design and advances in big data computing require massively parallel algorithms. The parallel external memory (PEM) model extends the external memory model so that it also captures parallelism. It consists of multiple processors which each have a private cache and share the (external) memory.

This thesis considers three computational problems in the context of (parallel) external memory algorithms. For the fundamental problem of list ranking, previously, an algorithm was known that has sorting complexity for many settings of the PEM model. In the first part of this thesis, this algorithm is complemented by matching lower bounds for most practical settings. Interestingly, a stronger lower bound for parameter ranges which previously have not been considered is shown. By modeling how list ranking algorithms retrieve information on the structure of the list in the memory, we give a lower bound that is quadratic in sorting complexity for certain parameter settings. It is noteworthy that this result implies the first non-trivial lower bounds for list ranking for the bulk synchronous parallel and the MapReduce model. These lower bounds are complemented by a list ranking algorithm which is, in contrast to previous algorithms, analyzed for all parameter settings of the PEM model.

In the second part, an efficient algorithm for the PEM model to compute a tree decomposition of bounded width for a graph is presented. The main challenge is to implement a load balancing strategy such that the running
time of the algorithm scales linearly with the number of processors as long as possible. A fundamental building block to achieve this with an asymptotically small number of cache misses is the above-mentioned list ranking algorithm. Therefore, the complexity of this algorithm is similar to sorting for many parameter settings. On the other hand, it also inherits the runtime behavior of list ranking which leads to a complexity that is worse than sorting for certain parameter settings. Subsequently, an easy application of the treewidth algorithm is presented in the field of kernelization.

In the last part, the complexity of modifying a graph so that it has bounded pathwidth is considered. Here, the modification operation is the splitting of vertices and the partitioning of their edges. Previous research exhibited that the number of vertex splits is tightly connected to the number of cache misses of a program for sparse matrix multiplication. This line of research is continued and the NP-completeness of this problem is shown. Furthermore, the subproblem of splitting vertices of a graph into a graph of pathwidth 1 is investigated. It is shown to admit a 2-approximation.
Zusammenfassung


Acknowledgement

This thesis would not have been possible without the support of many people. I am especially grateful to my supervisor, Riko Jacob, PhD, who enabled this thesis in many ways. Most importantly, he introduced me to the field of external memory algorithms. He shared his knowledge in many enlightening discussions and supported my research by guidance, new ideas, critical questions, and by creating a great working environment.

I am also very grateful to Prof. Peter Widmayer for examining this thesis, his advice, and the possibility to work in his group. Furthermore, my gratitude goes to Prof. Norbert Zeh for examining this thesis.

Besides the scientific support of Riko Jacob this thesis emerged from two collaborations. I would like to thank Nodari Sitchinava for the constructive visit in Zurich and the following video conferences to the other side of the globe resulting in a paper. Furthermore I would like to thank Matthias Mnich for his insights on kernelization.

I am very grateful to my colleague Phillipp Hupp for sharing an office, many discussions, and experiments regarding geometry. Of course I would like to thank my colleagues at the chair of Peter Widmayer for integrating us in their group. I would like to thank Chandan Dubey, Dennis Komm, and Marcel Schöngens for reading parts of this thesis and their helpful feedback.

Furthermore I would like to thank my friends for many interesting discussions, sharing holidays together, climbing the (swiss) mountains and just being there. Last but not least, I would like to thank my parents, my siblings, and my whole family for supporting me ever since I can think.
# Contents

1 Introduction 1
   1.1 Structure and contribution of this thesis 5
   1.2 Research contribution 6
   1.3 Definitions 7
      1.3.1 Graphs 7
      1.3.2 Directed graphs 10
      1.3.3 Kernelization 10
      1.3.4 Definitions for kernelizations on sparse graphs 11

2 External memory models 13
   2.1 The external memory model 13
   2.2 The parallel external memory model 14
      2.2.1 The parallel random access machine 15
      2.2.2 Concurrent memory access in the PEM model 16
      2.2.3 Graph representation in the PEM model 16
   2.3 Models for distributed computing 17
      2.3.1 The bulk synchronous parallel model 17
      2.3.2 The MapReduce programming model 18
   2.4 Fundamental algorithms for the PEM model 19
      2.4.1 Sorting 19
      2.4.2 Permuting 20
      2.4.3 Prefix sums 22
      2.4.4 Load balancing for simple tasks 22
   2.5 A fundamental lower bound for the PEM model 24
      2.5.1 Modeling 25
      2.5.2 A permuting lower bound in the PEM model 25

3 List Ranking 29
   3.1 Related work: algorithms for list ranking 30
      3.1.1 PRAM algorithms 31
      3.1.2 Algorithms for external memory models 33
   3.2 An improved algorithm for list ranking 35
      3.2.1 Ranking a small list 35
      3.2.2 Making a list small 36
# Contents

3.2.3 Independent set construction ........................................... 37
3.3 Lower bound via simple functions ....................................... 39
3.4 Counting lower bounds for the list ranking problem ............... 40
  3.4.1 Semigroup evaluation .................................................... 40
  3.4.2 Atomic edge contraction ................................................ 46
  3.4.3 Randomization and relation to list ranking ....................... 47
3.5 The guided interval fusion problem ..................................... 49
  3.5.1 Definitions ................................................................. 50
  3.5.2 Construction of the proof ............................................. 52
  3.5.3 Significance of the lower bound .................................... 57
  3.5.4 On usability of information ......................................... 59
  3.5.5 Improving the lower bound for GIF ................................. 67

4 An algorithmic toolbox for the PEM model ............................ 71
  4.1 Simple tasks for the PEM model ........................................ 72
    4.1.1 Duplicate removal ..................................................... 72
    4.1.2 Selecting elements with some property ............................ 73
    4.1.3 Compute a spanning tree of a DAG ................................. 74
  4.2 Simulation of PRAM algorithms ........................................ 74
    4.2.1 Computing acquainted twins ....................................... 77
    4.2.2 Computing an FIS for a bounded degree graph ................. 78
  4.3 Other fundamental algorithms for the PEM model .................. 79
    4.3.1 The Euler tour technique ............................................ 79
    4.3.2 Making a tree decomposition binary ............................... 80
    4.3.3 Computing a contraction tree for a tree ....................... 81
    4.3.4 Processing an annotated contraction tree ....................... 84
    4.3.5 Converting an implicit, into an explicit tree decomposition 85

5 Computing tree decompositions in the PEM model ..................... 91
  5.1 Overview on the algorithm ............................................. 92
    5.1.1 Decreasing the size of a graph by a constant fraction ....... 93
    5.1.2 Balancing a tree decomposition ................................... 94
    5.1.3 Computing a tree decomposition of width $k$ .................. 94
  5.2 A PEM-efficient load balancing framework .......................... 94
    5.2.1 Decreasing the size of a graph by a constant fraction ....... 95
    5.2.2 Making a tree decomposition efficiently evaluable ............ 97
    5.2.3 Making a tree decomposition nice ................................ 102
  5.3 Dynamic programming for refining tree decompositions .......... 103
    5.3.1 Replacing flippable DAGs .......................................... 103
    5.3.2 Refining a tree decomposition ................................... 107
  5.4 Analysis ................................................................. 110
    5.4.1 Independent sets by delayed pointer processing ............... 110
    5.4.2 Analysis for the full parameter range ........................... 112
  5.5 A permuting lower bound .............................................. 113
### Contents

5.6 Conclusion ............................................. 115

6  Linear-sized kernelizations for the PEM model 117  
   6.1 Finding and replacing protrusions ............... 119  
      6.1.1 PEM-efficient protrusion finder .......... 120  
      6.1.2 PEM-efficient protrusion replacer ....... 122  
   6.2 Applying the protrusion replacer ............... 125  
   6.3 A permuting lower bound ....................... 126  
   6.4 Conclusion ........................................ 126  

7  On evaluating a bilinear form in the EM model 129  
   7.1 Vertex splitting ................................... 132  
      7.1.1 Structural properties of vertex splitting .. 133  
      7.1.2 NP-hardness of splitting a graph to pathwidth 1 134  
      7.1.3 NP-hardness of splitting a graph to pathwidth $k$ for $k \geq 3$ 136  
   7.2 Bilinear form evaluation with vertex splits .... 144  
   7.3 Algorithms for split to caterpillar ............. 148  
      7.3.1 Splitting a tree to caterpillars .......... 148  
      7.3.2 Approximation for dense graphs .......... 153  
      7.3.3 The general case ......................... 154  
   7.4 Conclusion ......................................... 157  

8  Conclusion ........................................... 159  
   8.1 Results ............................................ 160  
   8.2 Implications and open problems ................ 161  
   8.3 Summary .......................................... 163  

Bibliography ............................................. 165
CHAPTER 1

Introduction

In the last 25 years, hardware manufacturers were confronted with two main obstacles in designing hardware for increasingly faster computers. The first challenge is that the frequency of processors increased much faster than the speed of the main memory. The second challenge rose in the early 2000s. Mostly due to high processor frequencies, the amount of heat to be dissipated from the CPU increased and became hardly manageable.

Over the years, the discrepancy between the memory access time and the frequency of processors increased. For hiding large memory access times, a hierarchy of fast caches has been introduced. While older processors have only one cache, current processors often have three cache levels of increasing sizes and different access times.

Computers designed after the von Neumann model admit fast algorithms that can be written without detailed knowledge of the underlying hardware. For maintaining this programming model, the cache hierarchy was implemented such that it appears to be invisible to the programmer.

To this end, caches load and evict cache lines by (mostly) hidden heuristics/rules. These rules are carefully chosen such that the amortized memory access times are small for many memory access patterns of common programs. The rules often exploit temporal locality and spatial locality in memory access patterns. Temporal locality means that a memory address is accessed multiple times in a short time span, while spatial locality means that several memory addresses are accessed which are in each other’s vicinity in the linearly addressed memory. Temporal locality is considered mostly by rules for evicting cache lines. On the other hand, spatial locality is taken into account, among others, by prefetching heuristics: they arrange that an address is loaded into the cache before an algorithm actually uses the value of the address.
The increasing heat dissipation prevented manufacturers from creating processors with higher frequencies. Several techniques were applied to increase the computing power of the chips that contain CPUs further. In particular, the decreasing feature size for integrated circuits enabled putting more than one CPU on a die. Ever since, the number of processors on a die increased to currently up to 18 CPUs on one die. It is projected that the number of CPUs per die will increase further. Thus, algorithms have to use many CPUs in parallel for using the computational power of current and future CPUs to its fullest. While there are several approaches for parallelizing algorithms automatically, some algorithmic approaches are inherently sequential and thus hard to parallelize.

Already in the late 1970s and 1980s similar insights led to fundamental research on parallel algorithms. Back then, a simple yet expressive theoretical model for analyzing parallel algorithms was established, called the parallel random access machine (PRAM) model. It is a parallel version of the random access machine (RAM) model, the standard model for analyzing sequential algorithms.

Despite the attempts to simulate a von Neumann or RAM-like computation model on current computers, the running time of an algorithm executed on an actual computer does not necessarily correlate with the running time predicted by the RAM or PRAM model. One frequent reason for these deviations are large input sets. More precisely, they occur in situations in which the input size is larger than the largest cache and if neither temporal nor spatial locality is exploited by the algorithm. For capturing these effects in the sequential case, the external memory (EM) model was introduced [AV88]. Analogously to the PRAM model, the parallel external memory (PEM) model was introduced as a parallel extension of the EM model [Arg+08]. It models $P$ processors that each have a cache for $M$ data items and can load cache lines, called blocks, containing $B$ records from a shared (external) memory. As opposed to the classical (P)RAM model, in the (P)EM model the complexity measure captures the number of cache misses instead of the computing operations.

Besides the well-understood RAM algorithms, there are optimal algorithms for several fundamental problems in the EM model. Some of the tasks are sorting and permuting [AV88], list ranking [Chi+95], sparse matrix dense vector (SpMxV) multiplication [Ben+10], and treewidth computation for a graph [MZ09]. In the EM model, all these problems have essentially the complexity of sorting. This might be surprising since all these problems, except
sorting, have linear complexity in the RAM model. The fundamental paper of Arge et al. that introduced the PEM model shows that a variation of the merge sort algorithm can be used to sort optimally in the PEM model [Arg+08]. A deeper analysis by Greiner proves that this algorithm is indeed optimal for the full parameter range of \( P, M, \) and \( B \) [Gre12]. Furthermore, the algorithm basically settled the worst-case complexity of SpMxV multiplication in the PEM model.

As most fundamental operations in linear algebra, sparse matrix dense vector multiplication is a truly important building block in the field of scientific computing. A consequence of the lower bounds for SpMxV in the PEM model [Gre12], and even in the EM model [Ben+10] is that nearly every sparse matrix is hard in the sense that it actually needs the worst-case running time. On the other hand, it is not clear what a matrix looks like that needs asymptotically more running time than the trivial scanning bound. Previous research showed a tight connection between the pathwidth of a certain graph \( G_A \) of a matrix \( A \) and the number of necessary cache misses required to compute the SpMxV product. Considered from the point of view of scientific computing, it is good to know in which cases a matrix can be multiplied very efficiently, meaning below worst-case running time. On the other hand, actually exploiting the pathwidth of \( G_A \) in the context of a sparse matrix multiplication in a (P)EM algorithm means that the graph \( G_A \) is large. Thus, a RAM algorithm for computing the pathwidth of \( G_A \) would be horribly slow, not only due to huge constants in the asymptotic notation of current pathwidth algorithms. This raises the question whether there is a PEM-efficient algorithm to compute a path decomposition in order to extend the feasible graph sizes.

**Treewidth** Algorithms for computing the treewidth or the pathwidth for a graph are built on the same fundamental theory [RS04], and thus are tightly connected [Bod96; BK96; BH98; MZ09]. If a graph family has a tree decomposition of small (tree-)width, this has strong algorithmic consequences. Many \( NP \)-hard problems, such as computing a maximum independent set, become polynomial time solvable [AP89; Arn85; Bod88; Bod93b; Cou90].

Recently, classes of efficiently solvable problems on graph classes that have bounded treewidth or fulfill similar properties were described that yield by meta-theorems linear time algorithms in the RAM model [Kim+13; Fom+12b].
This theory lies in the area of kernelization algorithms, which construct for an input instance an equivalent instance of small size. This (very) small instance can be solved by a brute force computation, which is in general easily parallelizable and cache friendly. Therefore, thinking of “small size” in a magnitude such that the required space of a brute force algorithm does not exceed the cache size naturally demands PEM-efficient kernelization algorithms. The algorithms implied by the meta-theorems rely fundamentally on an efficient algorithm for computing tree decompositions. Thus, PEM-efficient algorithms for computing tree decompositions are important.

While the algorithm of Maheshwari and Zeh computes a tree decomposition of bounded width efficiently in the EM model [MZ09], it cannot be generalized easily to the PEM model. In crucial parts, it relies on a technique called time forward processing [Chi+95; Arg03], for which an efficient parallel implementation is unclear [Lad75; GHR95]. Thus, one major task in developing an efficient treewidth algorithm in the PEM model is to develop a PEM-efficient version of these components of the algorithm of Maheshwari and Zeh. Replacing these parts reveals the fundamental role of list ranking for graph algorithms in the PEM model.

**List ranking** List ranking is known to be a building block for several important graph problems [Chi+95]. Arge, Goodrich, and Sitchinava presented an algorithm for list ranking which exhibits sorting complexity for an important part of the parameter range of the PEM model [AGS10]. However, in contrast to the complexity of sorting, there is no simple argument known for list ranking that easily generalizes the lower bound for list ranking from the EM model [Chi+95] to the PEM model. On the other hand, designing a list ranking algorithm reveals that it is hard to obtain an algorithm that requires only a number of cache misses which is asymptotically equal to sorting for all parameter settings in the PEM model.

This suggests that list ranking might be a fundamental task that exhibits different behaviour in the PEM model and in the PRAM model, similar to the difference between permuting in the EM and the RAM model. While permuting elements in the RAM model is trivially solvable in linear time, in the EM model it has the asymptotic complexity of sorting for many parameter settings. Since permuting is often an inherent part of computational problems, they have
permuting complexity in the EM model as well. Likewise, the complexity of list ranking in the PRAM model is asymptotically smaller than the complexity of sorting. However, this does not seem to be true for the complete parameter range in the PEM model.

1.1 Structure and contribution of this thesis

In Chapter 2, the external memory model and the parallel external memory model are described in detail. Furthermore, a permuting lower bound for the Proximate Neighbors problem [Chi+95] in the PEM model is presented. The Proximate Neighbors problem is important since it is often used as an intermediate problem to prove permuting lower bounds in the (P)EM model. For completeness, this chapter also restates the complexities for the most fundamental tasks in the PEM model.

Chapter 3 considers the list ranking problem. First, by a non-trivial combination of existing algorithms for list ranking, a new algorithm is obtained. In contrast to the results of Arge, Goodrich, and Sitchinava [AGS10], this algorithm works efficiently for the full parameter range of the PEM model. For a wide range of the parameter settings of the PEM model it has sorting complexity. On the other hand, for certain parameter settings, it has asymptotically a complexity of $O(\log^2 N)$ for a list of length $N$. Note that the complexity for sorting in this parameter setting is $O(\log N)$. The remainder of Chapter 3 is devoted to proving lower bounds for the list ranking problem. First, a permuting lower bound is established, based on the result for the Proximate Neighbors problem. This also yields a detailed proof for the permuting lower bound for list ranking in the EM model claimed by Chiang et al. [Chi+95]. Considering the parameter setting that yields list ranking algorithms that are asymptotically worse than sorting, it appears that not only permuting the elements of a list ranking instance is a challenge in solving it, but also how extracted information can be used by processors. The novel Guided Interval Fusion problem is an attempt to model how information is used in list ranking algorithms. For certain parameter settings, it is used to prove a lower bound of $\Omega(\log^2 N)$ parallel I/Os for classes of list ranking algorithms in the PEM model.

In order to obtain an algorithm for computing a tree decomposition for a graph, in Chapter 4, several computational problems are considered. For each of
the presented problems there is an algorithm, often based on a corresponding PRAM algorithm, that achieves at most list ranking complexity. Subsequently, in Chapter 5, an algorithm for computing a tree decomposition of bounded width is given. It is fundamentally based on the PRAM algorithm of Bodlaender and Hagerup and the algorithm of Maheshwari and Zeh for computing tree decompositions in the EM model [BH98; MZ09]. Since the algorithm is a combination of two individually complex implementations, one focus is to provide a crisp description of the algorithm. Therefore, clear interfaces for the procedures of the algorithm are defined in order to obtain a presentation without repeating proofs from previous algorithms that remain valid. Chapter 6 presents an application of the PEM-efficient tree decomposition algorithm in the area of kernelization algorithms, showing that a certain class of fixed parameter tractable problems has asymptotically PEM-efficient linear kernels.

Finally, the mentioned work on the connection between the pathwidth of a certain graph of a matrix and EM-efficient algorithms for SpMxV multiplication is continued. For a specific setting of $M$ and $B$ in the EM model, Chapter 7 provides a proof of the NP-completeness for determining the number of non-compulsory cache misses needed to evaluate a bilinear form, induced by a sparse matrix. These results are further augmented by providing approximation algorithms for the very specific case of $M = 2$ and $B = 1$, yielding constant-factor approximations.

### 1.2 Research contribution

This thesis is based on joint work with Riko Jacob, Nodari Sitchinava, and Matthias Mnich. The central results on list ranking, presented in Chapter 3, are published [JLS14b; JLS14a]. The central results considering tree decompositions of bounded width, presented in Chapters 4, 5, and 6, are published [JLM14]. Therefore, parts of this thesis are taken identically from these publications.
1.3 Definitions

In this thesis, the following notation is used. Euler’s number is denoted by \( e = 2.7182818 \ldots \). The set of natural numbers is denoted by \( \mathbb{N} = \{1, 2, 3, 4, 5 \ldots \} \), the positive integers are denoted by \( \mathbb{Z}^+ = \mathbb{N}_0 = \{0, 1, 2, 3, 4, 5 \ldots \} \) and the empty set is denoted by \( \emptyset \). An interval \([a, a+1, \ldots, b-1, b]\) of the natural numbers is denoted by \([a, b]\), and \([N]\) is a shorthand for \([1, N]\). The subset relation is denoted by \( \subseteq \), and therefore \([5, 10]\) \( \subseteq [10] \subseteq [1, \ldots, 10] \) holds. The proper subset relation is denoted by \( \subset \). So \([5, 10]\) \( \subset [10] \subseteq [1, \ldots, 10] \) holds. The disjoint union of two sets \( A \) and \( B \) is denoted by \( A \sqcup B \). The Cartesian product \( \{ (a, b) \mid a \in A, b \in B \} \) of two sets \( A \) and \( B \) is denoted by \( A \times B \). For a set \( A \), the expression \( A^k \) is defined recursively by \( A^k = A \times A^{k-1} \) and \( A^1 = A \).

The factorial \( n! \) of a number \( n \in \mathbb{N}_0 \) is defined as \( \prod_{i=1}^{n} i \). For \( n \geq k \in \mathbb{N}_0 \), the binomial coefficient \( \binom{n}{k} \) is defined as \( \frac{n!}{(n-k)!k!} \). For a set \( A \), the expression \( \binom{A}{k} \) denotes all subsets of \( A \) of size \( k \).

Furthermore, the floor function \( \lfloor r \rfloor \) for some real number \( r \geq 0 \) is the biggest integer that is not larger than \( r \). Conversely, the ceiling function \( \lceil r \rceil \) denotes the smallest integer that is not smaller than \( r \). Thus, \( \lfloor e \rfloor = 2 = \lceil e^2 \rceil = \lfloor 2^2 \rfloor = \lceil 2 \rceil = \lfloor 2 \rfloor \).

An unspecified base of a logarithm is always assumed to be 2: \( \log n = \log_2 n \). The natural logarithm \( \log_e n \) is denoted by \( \ln n \). The abbreviation \( \log^2 n \) denotes the square of a logarithm, \( (\log n) \cdot (\log n) \). The \( k \)-times iterated logarithm \( \log^{(k)} n \) is defined recursively as \( \log n \) if \( k = 1 \) and \( \log \log^{(k-1)} n \) otherwise. The term \( \log^* n \) is defined recursively: if \( n < 1 \) it evaluates to 0 and otherwise it evaluates to \( 1 + \log^* (\log n) \). In this thesis, the term \( \log \) is used for the expression \( \max\{1, \log n\} \).

A semigroup is a set \( S \) together with a binary operation “\( \cdot \)” that is an associative function \( : S \times S \mapsto S \). This means for all \( a, b, c \in S \), it is true that \( (a \cdot b) \cdot c = a \cdot (b \cdot c) \).

1.3.1 Graphs

All graphs considered in this thesis will be finite and simple, that is, without loops or multiple edges, unless stated explicitly otherwise. An undirected graph \( G \) consists of a finite set of vertices \( V(G) \) and a set of edges \( E(G) \subseteq \binom{V}{2} \).
If the context is clear, their short versions $V$ and $E$ are used. The size $|G|$ of a graph $G$ is $|V| + |E|$.

The vertices $u$ and $v$ of an edge $e = \{u,v\}$ are said to be adjacent to each other, and $u$ and $v$ are incident to the edge $e$. The neighborhood $N_G(v)$ of a vertex $v$ in $G$ is the set of all vertices that are adjacent to $v$: $N_G(v) = \{u \in V(G) \mid \{u,v\} \in E(G)\}$. The subscript $G$ is omitted if it is clear from the context. The degree of a vertex $v$ is the size of its neighborhood $\deg(v) = |N(v)|$.

A subgraph $G' = (V',E')$ of a graph $G = (V,E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq E$. The induced subgraph $G[V']$ is defined as the edge-maximal subgraph $G' = (V',E')$ of $G$.

An independent set of a graph $G$ is a set of vertices $I$ such that $G[I]$ does not contain edges. A fractional independent set (FIS) of $G$ is an independent set which contains at least $c \cdot |V|$ vertices for some $c > 0$. A partition of the vertex set of a graph $G$ into $k$ classes $C_1, \ldots, C_k$ is called a $k$-coloring if each $C_i$ is an independent set in $G$.

A bipartite graph $G$ has a partition of its vertices $V = U \uplus W$ such that $U$ and $W$ are independent sets. This means, a bipartite graph has a 2-coloring. The complete bipartite graph between two independent sets of size $a$ and $b$ contains all $a \cdot b$ possible edges between the independent sets and is denoted by $K_{a,b}$.

A path $P$, or also $v_1v_n$-path, of length $n$ is a graph on the vertices $\{v_1, v_2, \ldots, v_n\}$ and the edge set of $P$ is defined by $E(P) = \{[v_i, v_{i+1}] \mid 1 \leq i \leq n - 1\}$. The distance between two vertices $u$ and $v$ is the number of edges of a shortest path between $u$ and $v$. A path with $v_1 = v_n$ is called a cycle. A graph $G$ is called connected if for all vertices $u, v \in V$ there is $uv$-path which is a subgraph of $G$. A maximal connected subgraph of a graph $G$ is called a connected component of $G$.

A tree is a connected graph which does not contain a cycle. The vertices of a tree that have degree 1 are called leaves; all other vertices are called internal vertices. For a tree that has a root vertex $r$, the children of a vertex $v$ are the vertices in the neighborhood of $v$ which are not on the $rv$-path. Conversely, the vertex in the neighborhood of $v$ on the $rv$-path is called the parent of $v$. A tree is called rooted at $r \in V$ if for each vertex, its parent vertex with respect to $r$ is known. A tree is a perfect binary tree if all internal vertices have two children and all leaves have the same distance to the root vertex $r$. A caterpillar is a tree
that is a path after removing all leaves. A spanning tree of a graph \( G \) with \( n \) vertices is a subgraph \( S \) of \( G \) that is a tree and has \( n \) vertices.

In a list graph, each vertex has degree at most 2. Note that such a graph can contain several cycles! A matching is a subset of the edges such that there is no vertex incident to more than one edge. A graph is \( k \)-connected if removing any vertex set of size \( k - 1 \) does not disconnect the graph. A separator \( S \) for a graph \( G = (V, E) \) is a subset of \( V \) so that \( G[V \setminus S] \) has at least one connected component more than \( G \).

An \( r \)-ruling set in a list graph \( G \) is an independent set \( I \) of \( V \) such that for each vertex \( v \in V \) there is a vertex \( u \in I \) such that there is a \( uv \)-path in \( G \) of length at most \( r - 1 \).

A tree decomposition \( D \) for a graph \( G \) is a pair \((T, B)\) consisting of a tree \( T \) and a collection of bags \( B = \{B_i \mid i \in V(T)\} \) such that

- \( \bigcup_{i \in V(T)} B_i = V(G) \),
- each edge \( \{u, v\} \in E(G) \) satisfies \( \{u, v\} \subseteq B_i \) for some \( i \), and
- for each \( v \in V(G) \) the set \( \{i \mid v \in B_i\} \) induces a subtree of \( T \).

The width of a tree decomposition \((T, B)\) is \( \max_{i \in V(T)} |B_i| - 1 \). The treewidth of a graph \( G \) is the minimum width over all tree decompositions for \( G \), and denoted by \( \tw(G) \).

A tree decomposition \((T, B)\) for a graph \( G \) is rooted if \( T \) is rooted, and a rooted tree decomposition is nice if every bag \( B \in B \) has one of the following four types: leaf bag with no children in \( T \) and \( |B| = 1 \); introduce bag with one child \( B' \) in \( T \) and \( B = B' \cup \{v\} \), for some vertex \( v \in V(G) \setminus B' \); forget bag with one child \( B' \) in \( T \) and \( B = B' \setminus \{v\} \), for some vertex \( v \in B' \); or join bag with children \( B', B'' \) in \( T \) and \( B = B' = B'' \).

A path decomposition for a graph \( G \) is a tree decomposition \((T, B)\) where \( T \) is a path. The pathwidth of a graph \( G \) is defined accordingly and denoted by \( \pw(G) \).

Given an edge \( e = \{x, y\} \) of a graph \( G \), contracting \( e \) means the vertices \( x \) and \( y \) are identified to a vertex which is adjacent to the old neighbors of \( x \) and \( y \) (except \( x \) and \( y \)). A graph \( H \) obtained by a sequence of edge contractions is said to be a contraction of \( G \). A graph \( H \) is a minor of a graph \( G \) if \( H \) is the contraction of some subgraph of \( G \). The graph \( G \) is called \( H \)-minor free if it
does not contain \( H \) as a minor. A graph class \( G_H \) is called \( H\)-minor free if all its members are \( H\)-minor free.

### 1.3.2 Directed graphs

A directed graph \( G \) consists of a finite set of vertices \( V(G) \) and a set of directed edges \( E(G) \subseteq V \times V \setminus \{(v,v) \mid v \in V\} \). A (directed) path \( P \) is a subgraph of a directed graph such that \( P \) is a vertex sequence \( (v_1,v_2,...,v_n) \) such that \( E(P) \) is defined as \( \{ (v_i,v_{i+1}) \mid 1 \leq i \leq n - 1 \} \). A cycle is a directed path where \( v_1 = v_n \).

A directed graph is called acyclic if it does not contain a cycle as a subgraph. A directed acyclic graph is abbreviated as DAG. The in-neighborhood \( N^-(v) \) of a vertex \( v \) of a directed graph \( G \) is the set of vertices whose target vertex is \( v \), formally defined as \( \{ u \in V \mid (u,v) \in E(G) \} \). Accordingly, the in-degree of a vertex \( v \) is \( |N^-(v)| \). Similarly, the out-neighborhood \( N^+(v) \) of \( v \) is the set \( \{ u \in V \mid (v,u) \in E(G) \} \). Accordingly, the out-degree of a vertex \( v \) is \( |N^+(v)| \). A vertex \( v \) in a directed graph is called a source if its in-neighborhood has size 0. A vertex \( v \) in a directed graph is called a sink if its out-neighborhood has size 0.

The underlying undirected graph \( G \) of a directed graph \( G' = (V,E') \) is defined as \( G = (V,E) \) with \( E = \{ (u,v) \mid (u,v) \in E' \} \). A subgraph \( S \) of a graph \( G' \) is called a spanning tree of \( G' \) if the underlying undirected graph of \( S \) is a spanning tree of the underlying undirected graph \( G \).

### 1.3.3 Kernelization

A parameterized problem \( \Pi \) is a subset of \( \Sigma^* \times \mathbb{N} \) for some finite alphabet \( \Sigma \). An instance of \( \Pi \) is a pair \( (x, \kappa) \), where \( \kappa \) is called the parameter. A kernelization for a parameterized problem \( \Pi \) is an algorithm that compresses in time \( (|x| + \kappa)^O(1) \) any instance \( (x, \kappa) \in \Pi \) into an instance \( (x', \kappa') \in \Pi \), such that \( (x, \kappa) \in \Pi \Leftrightarrow (x', \kappa') \in \Pi \), with \( |x'|, \kappa' \leq g(\kappa) \) for some computable function \( g \). The instance \( (x', \kappa') \) is called a kernel of \( \Pi \), and the function \( g \) is referred to as the size of the kernel. If \( g(\kappa) = O(\kappa) \), then \( \Pi \) is said to admit a linear kernel.

The complexity class of problems that are decidable by a nondeterministic turing machine in polynomial time are denoted by \( \text{NP} \). The complexity class of fixed parameter tractable problems is denoted by \( \text{FPT} \).
1.3 Definitions

1.3.4 Definitions for kernelizations on sparse graphs

The remaining definitions are included to define the computational problems covered in Chapter 6. However, they are not crucially important to understand the content of that chapter and are included for completeness. All definitions can be found, mostly literally, in [Fom+12b; Kim+13].

Let \( r \geq 2 \) be a natural number. The graph on the vertex set \( V_r = \{(i,j) \in [r]^2 \} \) and the edge set \( \{((i,j),(k,l)) \in \binom{V}{2} \mid (i = k \text{ and } j = l - 1) \text{ or } (i = k - 1 \text{ and } j = l) \} \) is called \((r \times r)\)-grid.

For a graph \( G \) and \( X \subseteq V(G) \), the term \( \partial G(X) \) is defined as the set of vertices in \( X \) having a neighbor in \( V \setminus X \). The set \( X \) is an \( r \)-protrusion of \( G \) if \( |\partial G(X)| \leq r \) and \( \text{tw}(G[X]) \leq r \).

For an integer \( \tau \in \mathbb{N} \), a \( \tau \)-boundaried graph is a graph \( G \) with \( \tau \) distinguished vertices that are uniquely labeled from 1 to \( \tau \). The set \( \partial(G) \) of labeled vertices is called the boundary of \( G \), and its elements are called boundary vertices. The set \( G_\tau \) denotes all \( \tau \)-boundaried graphs of a graph class \( G \).

For two \( \tau \)-boundaried graphs \( G_1 \) and \( G_2 \), the expression \( G_1 \oplus G_2 \) denotes the \( \tau \)-boundaried graph that is obtained by taking the disjoint union of \( G_1 \) and \( G_2 \), and identifying each vertex of \( \partial(G_1) \) with the vertex of \( \partial(G_2) \) with the same label; that is, \( G_1 \) and \( G_2 \) are glued together at their boundaries.

Let \( \Pi_G \) be a parameterized graph problem restricted to a class \( G \), and let \( G_1, G_2 \) be two \( \tau \)-boundaried graphs in \( G_\tau \). We say that \( G_1 \equiv_{\Pi,\tau} G_2 \) if there exists a constant \( \Delta_{\Pi,\tau}(G_1,G_2) \) (that depends on \( \Pi_G \), \( \tau \), and the ordered pair \( (G_1,G_2) \)) such that for all \( \tau \)-boundaried graphs \( G_3 \) and for all \( \kappa \):

1. \( G_1 \oplus G_3 \in G \) iff \( G_2 \oplus G_3 \in G \);
2. \( (G_1 \oplus G_3, \kappa) \in \Pi_G \) iff \( (G_2 \oplus G_3, \kappa + \Delta_{\Pi,\tau}(G_1,G_2)) \in \Pi_G \).

The problem \( \Pi_G \) has finite integer index in the class \( G \) if and only if for every integer \( \tau \), the equivalence relation \( \equiv_{\Pi,\tau} \) has a finite number of equivalence classes. If \( (G_1 \oplus G, \kappa) \notin \Pi \) or \( G_1 \oplus G \notin G \), for all \( G \in G_\tau \) the value \( \Delta_{\Pi,\tau}(G_1,G_2) \) is defined to be 0.

A parameterized graph problem \( \Pi \) is contraction-bidimensional if for any pair of graphs \( G \) and \( H \), such that \( H \) is a contraction of \( G \), and integer \( k \in \mathbb{N} \),
if \((G,k) \in \Pi\) then \((H,k) \in \Pi\). On the other hand, there is a \(\delta > 0\) such that for the \((r \times r)\)-grid \(R\) and \(k \leq \delta r^2\), \((R,k) \notin \Pi\). In words, \(\Pi\) is contraction-bidimensional if contracting an edge in a graph \(G\) cannot increase the parameter and the value of the solution on \(R\) is at least \(\delta r^2\).

A contraction-bidimensional problem has the \textit{separation} property if the following holds. For a graph \(G\), a vertex cut \(S\) of \(G\) whose removal disconnects \(G\) into connected components \(C_1,\ldots,C_p\) and a subset \(I \subseteq \{1,\ldots,p\}\), let \(G_I\) be the graph obtained from \(G\) by contracting for every \(j \notin I\) the component \(C_j\) into the vertex in \(N(C_j)\) with smallest index. Let \(\text{opt}(G)\) be an optimal solution to \(G\). Then for any \(I \subseteq \{1,\ldots,p\}\), \(|\text{opt}(G) \cap G_I| \in \{|\text{opt}(G_I) - O(|S|)|,|\text{opt}(G_I) + O(|S|)|\}.

A graph problem \(\Pi\) parameterized by \(\kappa\) is called \textit{treewidth-bounding} if all “yes”-instances \((G,\kappa)\) of \(\Pi\) admit an \(O(\kappa)\)-sized set \(S \subseteq V(G)\) such that the graph \(G \setminus S\) has constant treewidth.

An \textit{apex graph} is a graph obtained from a planar graph \(G\) by adding a single vertex and making it adjacent to some of the vertices of \(G\). A graph class \(G_H\) is \textit{apex-minor free} if \(G_H\) excludes a fixed apex graph \(H\) as a minor. A graph \(H\) is a \textit{topological minor} of a graph \(G\) if \(H\) is obtained from a subgraph of \(G\) by contracting edges with at least one endpoint of degree at most two. A graph \(G\) is \textit{\(H\)-topological-minor free} if it does not contain \(H\) as a minor. A graph class \(G_H^{\top}\) is \textit{\(H\)-topological-minor free} (or, \textit{excludes \(H\) as a topological minor}) if all its members are \(H\)-topological-minor free. Since every topological minor is also a minor it follows for any graph \(H\), \(G_H^{\top} \supseteq G_H\).
CHAPTER 2

External memory models

This chapter provides an overview on the computational models that are used in this thesis. The central model for investigating the complexity of computational problems in this thesis is the parallel external memory model. First, the external memory model is introduced. The parallel external memory model is a parallel extension of the external memory model. It is described in Section 2.2. Section 2.3 shortly describes two distributed models, the bulk synchronous parallel model and the MapReduce model.

Besides the pure definitions, this chapter presents some of the most fundamental insights on the parallel external memory model. Section 2.4 introduces the complexity of some fundamental problems in the parallel external memory model. In Section 2.5, the fundamental building block for proving permuting lower bounds in the parallel external memory model is introduced.

2.1 The external memory model

Aggarwal and Vitter introduced the external memory (EM) model to investigate the number of input/output operations between a cache and an external memory for sorting-like problems [AV88]. As visualized in Figure 2.1, it consists, as the standard RAM model, of one CPU and the external memory. In contrast to the RAM model, a cache is located between the CPU and the memory.

The cache can store up to \( M \) elements, on which the CPU can perform any computational operations for free. The conceptually infinite (external) memory is partitioned into blocks, each consisting of \( B \) contiguous memory cells. A block, and thus the elements it contains, can be transferred between the cache and the memory as a unit. Such a transfer is called an input/output operation, or for
short I/O. The input of a computational problem of size $N$ is located initially in the first $\lceil \frac{N}{B} \rceil$ blocks. Note that we assume in this thesis globally $M \geq 2B$.

While in the standard RAM model the complexity measure is the number of operations to solve a computational problem, in the EM model it is the number of I/Os. Most noteworthy and characteristic for the EM model is the complexity of permuting a set of records. For almost all practical settings of $M$ and $B$, it has sorting complexity instead of linear complexity as in the RAM model. Chiang et al. announced permuting and sorting lower bounds for numerous problems in the EM model [Chi+95]. This resembles the fact that permuting is one of the most fundamental problems and is actually needed to solve numerous problems efficiently. The complexity for sorting in the EM model is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M} \frac{N}{B})$ I/Os. This corresponds to a speedup of approximately $B$ compared to a RAM heap sort algorithm that is trivially analyzed and implemented in the EM model.

### 2.2 The parallel external memory model

Arge et al. introduced the parallel external memory (PEM) model as a parallel extension of the EM model to capture the hierarchical memory organization of modern multicore processors [Arg+08]. The PEM model is depicted in Figure 2.2. It consists of $P$ uniquely identifiable CPUs which each have their own (private) cache of size $M$. It has some properties in common with the EM model: the PEM model has a conceptually infinite shared memory that is partitioned into blocks of $B$ contiguous memory cells. Input data of a computational problem initially resides in the first $\lceil \frac{N}{B} \rceil$ blocks of the shared memory. For processing any data by a processor, it must reside in its cache. Data exchange between the caches of the processors and the shared memory is done by I/O operations.
The complexity measure for computational problems in the PEM model is the parallel equivalent of the number of I/Os, the number of parallel I/Os. In one parallel I/O, every processor can perform one I/O operation to move data between its cache and the shared memory. Thus, up to \( P \) blocks can be transferred in each parallel I/O.

Processors have to exchange information and data from time to time. Some models assume that the processors are interconnected by some network infrastructure through which they can exchange information among each other. The PEM model does not assume such a network. Therefore, information and data can only be exchanged through the shared memory.

### 2.2.1 The parallel random access machine

The parallel random access machine (PRAM) is a parallel extension of the well-known RAM model. Note that a PEM model with \( B = 1 \) grants random access to the input. Therefore, it is closely related to the PRAM model. For a constant cache size \( M \) and \( B = 1 \), the PEM model can be considered a PRAM model with a different complexity measure. In the PRAM model, the complexity measure is the number of parallel (arithmetic) operations instead of the number of parallel I/Os. In one parallel operation, every processor can perform one (arithmetic) operation.

Besides the number of parallel operations, an important measure for the optimality of an algorithm in the PRAM model is the work it performs. Let \( \mathcal{A} \) be an algorithm which uses \( P \) processors and finishes after \( t \) time steps. The work that \( \mathcal{A} \) performs is defined to be \( w = P \cdot t \). A PRAM algorithm \( \mathcal{A} \) is called work-optimal if \( w \) is asymptotically the same as the number of time steps needed to solve the same problem in the RAM model. In other words, a work-optimal
algorithm is not allowed to use a huge number of processors without obtaining a proportional speedup over a sequential algorithm.

### 2.2.2 Concurrent memory access in the PEM model

Due to the parallelism of the PEM model, concurrent access to blocks of the shared memory may occur. Depending on the handling of such concurrent accesses, the power of the PEM model might differ. As in the classical PRAM model, there are three general policies to cope with concurrent access [KR90; KR88; BH82]:

1. The exclusive-read, exclusive-write (EREW) policy prohibits any concurrent access and is thus the most strict policy [LPV81].

2. A more liberal policy, is the concurrent-read, exclusive-write (CREW) policy, which allows concurrent access only if a block is read [FW78].

3. The concurrent-read, concurrent-write (CRCW) policy is the least restrictive policy, and it is subdivided into several rules. They determine in which case a concurrent write operation succeeds or which result is produced by such an operation [Gol78; SV81].

Despite the fact that most algorithms for the CREW policy can be modified for the EREW policy with minor changes [Gre12; BH98], in this thesis computational problems are investigated only in the CREW PEM model.

Note that the definition of a parallel I/O implies that the processors are synchronized in a certain way. This is a restriction compared to the first definition of the model by Sitchinava, which allows the processors to work asynchronously [Sit09]. However, this does not seem to be a drawback, as the lower bounds presented in this thesis hold for an asynchronous definition as well, even though this is not stated explicitly.

### 2.2.3 Graph representation in the PEM model

In this thesis several computational problems on graphs are considered. Therefore, their representation is stated here.
Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. We assume that every vertex of $V$ is represented by a unique natural number bounded by $O(|G|)$. Every directed edge $e = (u, v)$ is represented by a pair of vertices $(u, v)$ and thus by two natural numbers of $V$. An undirected edge $e = \{u, v\}$ is represented by the two directed edges $(u, v)$ and $(v, u)$. An edge list representation of $G$ consists of a set of natural numbers, which each represent a vertex, and the edge set as just defined.

If every vertex is incident to some edge, the vertex set does not need to be represented explicitly. Therefore, in this thesis, the representation of the vertex set is omitted in many cases. Since the vertices are natural numbers, they are totally ordered. This is used on several occasions for sorting edges by the natural order of their source or target vertices.

### 2.3 Models for distributed computing

In this section two models for distributed computing are introduced: the bulk synchronous parallel model and the MapReduce model. In this thesis, both models for distributed computing are considered only for one special case of their parameter range in order to give a simple generalization of a lower bound in the PEM model. Therefore, they are not described in their full generality.

#### 2.3.1 The bulk synchronous parallel model

The bulk synchronous parallel (BSP) model was introduced by Valiant for modeling parallel among processors that are connected by a network [Val90]. In contrast to the PEM model, the $P$ processors have the possibility to exchange data through a router, but there is no shared memory. Thus, for storing the input, each of the $P$ processors has a cache which is capable of storing at least $M = \Omega\left(\frac{N}{P}\right)$ records.

A computation in the BSP model is divided into rounds, that are called supersteps. After a superstep, all processors are synchronized and may send to up to $h$ processors a message that contains up to $M$ records. In the BSP model the communication costs are modeled in detail. Their exact description is omitted.
as they are irrelevant for this thesis. Despite a detailed modeling of the communication costs, the central complexity measure for an algorithm is basically the number of supersteps it needs to finish a task.

Note that this complexity measure implies that, from a theoretical point of view, the memory of each processor has to be bounded from above. Otherwise, a smaller number of processors can be used and therefore as smaller amount of information exchange may be needed. Therefore, it is often assumed $M = \Theta(N/P)$.

### 2.3.2 The MapReduce programming model

The MapReduce programming model was first introduced by Google for processing data at a large scale on a large pool of computers [DG04; DG08]. Subsequently, it became a success, among others due to the Hadoop framework [Whi09]. Similar to the BSP model, algorithms are organized in rounds and the number of rounds is the complexity measure, too. Each round consists of two sequential functions specified by the programmer: a map and a reduce function. The map function computes for each record an intermediate value and a corresponding intermediate key, which is not necessarily unique. A shuffle step, implemented by the MapReduce framework, permutes the intermediate values to produce for each key $k$ a list of values that have the key value $k$. Finally, the reduce step is applied to these lists of values. In general, it computes a smaller set of new keys and values for the next round, respectively for the output.

The map and the reduce function obtain only one record at a time which consists of a key and a value. Thus, the application of the map and the reduce function can be parallelized in a straight forward fashion. In practical scenarios, the input records are partitioned into small chunks of typically 64 Mega Byte [DG04]. Each chunk is processed by one computer in the pool by applying the map function [DG04].

As in the BSP model, from a theoretical point of view, it is important that the memory size of a computer is bounded. This can be achieved in several ways, which are omitted here [KSV10; GSZ11]. They lead to assumptions such as $M = O(N^{1-\epsilon})$ for $0 < \epsilon < 1$. 

18
2.4 Fundamental algorithms for the PEM model

The most fundamental task in the PEM model is to apply a function $f$ to each of $N$ input records. If $f$ can be applied in arbitrary order to each input record independently and yields for each record in constant computation time the correct result, this task is called *scanning*.

Scanning can be implemented easily by assigning to each processor a chunk of $\frac{N}{P}$ records which are stored in contiguous memory. Each chunk is then processed in a sequential fashion by a single processor using $O(\lceil \frac{N}{PB} \rceil)$ I/Os. Thus, the number of parallel I/Os needed to scan all $N$ input records is $\text{scan}_P(N) := O(\lceil \frac{N}{PB} \rceil)$ in the CREW PEM model.

A basic task that can be performed by scanning is computing the underlying undirected graph from a DAG. By scanning the edges of the DAG, the edge set of the underlying undirected graph can be computed easily by adding for each edge $(u,v)$ the edge $(v,u)$ to the edge set. In contrast to the EM model, prefix sums computation (Section 2.4.3) is a task that does not have scanning complexity in the PEM model.

### 2.4.1 Sorting

Arge et al. presented an optimal algorithm for sorting $N$ records in the CREW PEM model under the assumptions that $M = B^{O(1)}$ and $P \leq \frac{N}{B^2}$ [Arg+08]. Greiner improved their analysis and thus managed to improve the parameter range:

**Theorem 2.1 (Greiner [Gre12, Theorem 2.12])**

Let $d = \max\{2, \min\{\frac{N}{PB}, \frac{M}{B}\}\}$. The parallel I/O-complexity of sorting $N$ records using the PEM merge sort algorithm is

$$\Theta\left(\lceil \frac{N}{PB} \rceil \log_d \frac{N}{B}\right),$$

no matter how many processors are used.
Note that it is common to give a processor bound up to which an algorithm performs (work-)optimally in parallel models. Most of the theorems and lemmas in this thesis are not stated like this. If no processor bound is given, the statements are true for all settings of \( P \) even though the algorithms may not be optimal for all settings of \( P \).

Due to the importance of sorting for computational problems in the PEM model, the term \( \lceil N \frac{P}{B} \rceil \log_d \frac{N}{B} \) is denoted by \( \text{sort}_P(N,M,B) \).

### 2.4.2 Permuting

The task of **permuting** is defined as follows: given a permutation \( \pi \) over \([N]\), create the output \((x_{\pi(1)}, \ldots, x_{\pi(N)})\) from the input \((x_1, \ldots, x_N)\). It is well known that permuting is easy in the RAM model and has linear complexity. Again, Greiner determined the parallel I/O-complexity of permuting as \( \Theta(\min\{ \frac{N}{P}, \text{sort}_P(N) \}) \) if \( P \leq \frac{N}{B} \) [Gre12, Theorems 2.7 and 4.1]. The term \( \min\{ \frac{N}{P}, \text{sort}_P(N) \} \) is denoted due to its importance by \( \text{perm}_P(N,M,B) \).

The term \( \text{perm}_P(N,M,B) \) satisfies a useful property, which is covered by the following lemma:

**Lemma 2.2**

If \( N \geq 4B, M \geq 2B, \) and \( P \leq \frac{N}{B} \), then

\[
4 \text{perm}_P(N, 2M, 2B) \geq \text{perm}_P(N, M, B).
\]

**Proof** This lemma is trivially true if \( \text{perm}_P(N, 2M, 2B) \) evaluates to \( \frac{N}{P2B} \) and \( \text{perm}_P(N, M, B) \) evaluates to \( \frac{N}{P2B} \).

In the following, the case that both terms evaluate to the sorting terms is considered. Since the term for permuting \( \text{perm}_P(N, M, B) \) takes the minimum of these two terms, the lemma then follows immediately.

Let \( d = \max\{2, \min\{ \frac{N}{P2B}, \frac{M}{B} \} \} \) and \( d' = \max\{2, \min\{ \frac{N}{P2B}, \frac{M}{B} \} \} \). Since both sorting terms are assumed to be minimal, the lemma states that

\[
4 \frac{N}{P2B} \log_d \frac{N}{2B} \geq \frac{N}{P2B} \log_{d'} \frac{N}{B}.
\]
2.4 Fundamental algorithms for the PEM model

This is equivalent to

\[ 2 \frac{\log \frac{N}{2B}}{\log d} \geq \frac{\log \frac{N}{B}}{\log d'} \]  

(2.1)

Observe, whenever \( N \geq 4B \) it is true that

\[ 2 \log \frac{N}{2B} = \log \frac{N^2}{4B^2} \geq \log \frac{N}{B} \].

(2.2)

In the following, the different cases for the value of \( d \) are distinguished.

1. If \( \frac{N}{PB} \leq 2 \leq \frac{M}{B} \), then \( d = 2 \). Furthermore, if \( \frac{N}{PB} \leq 2 \), then \( d' = 2 \) as well and Equation 2.1 is true:

\[ 2 \frac{\log \frac{N}{2B}}{\log d} = 2 \log \frac{N}{2B} \geq \log \frac{N}{B} = \frac{\log \frac{N}{B}}{\log d'} \].

Otherwise, if \( 2 \leq \frac{N}{PB} \), then \( d' = \min\{\frac{M}{B}, \frac{N}{PB}\} \) and \( \log \frac{N}{B} \geq \frac{\log \frac{N}{B}}{\log \min\{\frac{M}{B}, \frac{N}{PB}\}} \) yielding:

\[ 2 \frac{\log \frac{N}{2B}}{\log d} = 2 \log \frac{N}{2B} \geq \log \frac{N}{B} \geq \frac{\log \frac{N}{B}}{\log \min\{\frac{M}{B}, \frac{N}{PB}\}} = \frac{\log \frac{N}{B}}{\log d'} \].

Therefore, in this case the lemma is true.

2. If \( 2 \leq \frac{N}{PB} \leq \frac{M}{B} \), then \( d = \frac{N}{PB} \). Therefore, if \( \frac{N}{PB} \leq \frac{M}{B} \), then \( d' = \frac{N}{PB} \).

Note that \( \frac{\log \frac{N}{B}}{\log \frac{N}{PB}} \geq \log \frac{N}{PB} \) always holds, and thus

\[ 2 \frac{\log \frac{N}{2B}}{\log d} = 2 \log \frac{N}{2B} \geq \log \frac{N}{B} \geq \frac{\log \frac{N}{B}}{\log \frac{N}{PB}} = \frac{\log \frac{N}{B}}{\log d'} \]

is true.

On the other hand, if \( \frac{M}{B} \leq \frac{N}{PB} \), then \( d' = \frac{M}{B} \).

By the assumption of this case, it follows that \( \frac{\log \frac{N}{2B}}{\log \frac{N}{PB}} \geq \log \frac{M}{B} \), yielding:

\[ 2 \frac{\log \frac{N}{2B}}{\log d} = 2 \log \frac{N}{2B} \geq \log \frac{N}{B} \geq \frac{\log \frac{M}{B}}{\log \frac{N}{PB}} = \frac{\log \frac{N}{B}}{\log d'} \].
3. If $2 \leq \frac{M}{B} \leq \frac{N}{P \cdot 2B}$, then $d = \frac{M}{B}$. Furthermore, since $\frac{M}{B} \leq \frac{N}{P \cdot 2B} \leq \frac{N}{PB}$, it follows $d' = \frac{M}{B}$.

Thus, it follows

$$2 \log \frac{N}{2B} = 2 \log \frac{M}{B} \geq \frac{\log \frac{N}{B}}{\log \frac{M}{B}} = \log \frac{N}{B}. \quad (2.2)$$

Consequently, the lemma is true. \(\square\)

In this thesis, for the terms \(\text{sort}_P(N,M,B)\) and \(\text{perm}_P(N,M,B)\), the parameters \(M\) and \(B\) are omitted if they are clear from the context. Therefore, the expressions are abbreviated by \(\text{sort}_P(N)\) and \(\text{perm}_P(N)\). Furthermore, note that both terms \(\text{sort}_P\) and \(\text{perm}_P\) scale linearly in \(P\), and thus the corresponding algorithms use the \(P\) processors efficiently.

### 2.4.3 Prefix sums

**Definition 2.3**
Given an input array \(A\) of size \(N\), the **prefix sums** problem is solved if an output array \(C\) contains for all \(0 \leq i < N\) the values \(C[i] = \sum_{j=0}^{i} A[j]\).

The task of prefix sums computation is often used for load balancing in numerous computational problems. Furthermore, since the add operation can be replaced by any other operation over a semi-group, numerous simple functions can be computed by the prefix sums problem. For completeness the complexity of prefix sums is stated.

**Lemma 2.4 (Arge et al. [Arg+08, Theorem 1])**
If the input array \(A\) is located in contiguous main memory, the prefix sums problem can be solved in the CREW PEM model in \(O(\text{scan}_P(N) + \log P)\) parallel I/Os.

### 2.4.4 Load balancing for simple tasks

In an efficient parallel algorithm, all processors should perform roughly the same amount of work for solving a computational task. In a task like computing
an undirected graph from a DAG, this can be achieved easily by scanning the
input as described in Section 2.4.

However, there are tasks which are similar to the following problem:

**Definition 2.5**
The input of the *copying problem* consists of $N$ elements in contiguous cells of
the shared memory, where each input element $x_i$ is annotated by an integer $c_i$
such that the sum $\sum_{i=1}^{N} c_i$ is bounded by $O(N)$. The task is to create for each
element $x_i$, $c_i$ copies such that all copies are located in contiguous memory.

A naïve implementation by scanning may take up to $O(\frac{N}{P})$ parallel I/Os, which
is merely the performance of the sequential EM algorithm. The following
lemma shows that the copying problem can be solved in $O(sort_P(N))$ parallel
I/Os.

**Lemma 2.6**

The copying problem can be solved in $O(sort_P(N))$ parallel I/Os in the CREW
PEM model.

**Proof**  In a first step, for each element $x_i$ the prefix sum $p_i = \sum_{j=1}^{i} c_j$ is com-
puted. By Lemma 2.4, this takes $O(scan_P(N) + \log P)$ parallel I/Os. This defines
for each element $x_i$ an interval $[p_{i-1} + 1, p_i]$ where it should be copied to. The fol-
lowing steps assign to each processor a memory address of the input such that
each processor generates only one contiguous interval of at most $\lceil \frac{PN}{P} \rceil$ elements
of the output.

To this end, each processor $j$ writes at the memory position $N + j \cdot B + 1$
the entry $a_j = j \cdot \lceil \frac{PN}{P} \rceil$. After collecting the values $a_j$ in a compact layout
in $O(\log B)$ parallel I/Os, these start-values are merged with the values $p_i$
by sorting. The intuition of this sorted result is that the processor $j$ should
read from the block which contains the closest element $x_i$ that is located in the
sorted list left of the value $a_j$ and represented by $p_i$. Therefore to each repre-
sentative of a processor $a_j$ the smallest memory position $i$ which is larger than
the value $a_{j-1}$ is annotated. Since it could happen that a value $p_i$ is followed
by several entries $a_j, a_{j+1}, \ldots, a_{j+k}$, the memory position $i$ has to be annotated to
all $k$ values $a_j, a_{j+1}, \ldots, a_{j+k}$.
However, this can be done by a prefix sums computation that uses the max-operation on the values \( i \) for \( p_i \) and 0 for the elements \( a_j \). Thus, after this modified prefix sums computation each value \( a_j \) is annotated by a memory address \( i \). These values can be assigned to each processor by reverting the sort operation that merged the \( p_i \) and \( a_j \) values.

Thus, each processor knows the memory location \( i \) at which the chunk that it should process ends. Therefore each processor \( j \) can output the elements that are copied to the output locations \((j - 1) \cdot \lceil \frac{PN}{P} \rceil \) to \( j \cdot \lceil \frac{PN}{P} \rceil \).

The most expensive procedure is sorting, and thus, the bound on the number of parallel I/Os follows.

\[ \square \]

### 2.5 A fundamental lower bound for the PEM model

Part of the challenge of proving lower bounds (in any model) is to restrict the model in a way to be able to prove non-trivial bounds while identifying the features of the model that emphasize the hardness of a particular problem.

An example of such a restriction in external memory models (both sequential and parallel) is the so-called *indivisibility assumption* \[AV88\]. The assumption states that each element of the input is processed as a whole and that an output element cannot be assembled from parts of other elements. This disallows compression schemes that merge several items into one.

For most computational problems it is not clear how to prove lower bounds in the PEM model that exceed the information-theoretic lower bounds \[KR90; Arg+08\] of \( \Omega(\log \frac{N}{B}) \) parallel I/Os without using the indivisibility assumption. Furthermore, for most computational problems it is not clear how the indivisibility assumptions can be broken to gain an advantage over existing algorithms. However, for the dictionary problem, Iacono and Patrascu presented an optimal data structure that explicitly breaks the indivisibility assumption \[IP12\].

In the next section, the definition of the PEM model, presented in Section 2.2, is specified more precisely for proving permuting lower bounds. It can be regarded as the parallel analog of the model for proving permuting lower bounds in the sequential EM model \[AV88\]. Here it is called the atomic PEM model.
2.5 A fundamental lower bound for the PEM model

2.5.1 Modeling

Similar to the paper of Aggarwal and Vitter, in this thesis, each element of the input is assumed to be an indivisible unit of data, an *atom*, which consumes one memory cell in the cache or shared memory [AV88]. In the most basic version of a PEM machine, the *atomic PEM model*, atoms exist only if they are given by the input. A program or an algorithm has limited knowledge about the content of an atom. Unless stated otherwise (Section 3.5), in this thesis, it is assumed that a program does not know anything about the content of an atom. The atomic PEM model is restricted to the following operations: an I/O operation reads or writes one block of up to $B$ atoms in the shared memory, and atoms can be copied or deleted.

For providing lower bounds for other problems, the concept of the atomic PEM model is extended in later sections, for example by an operation that can create a new atom from two given atoms.

In this thesis, the terms algorithm and program are distinguished in the following way: in an *algorithm* the control flow might depend on the input. This means that conditional statements and therefore loops are allowed. In contrast, a *program* is only a sequence of valid instructions for a PEM model that is independent of the input atoms and thus has no conditional statements. A program can be seen as an instantiation of an algorithm for a given instance of a computational task to which all results of conditional statements and simple computations, such as index computations, are presented beforehand.

2.5.2 A permuting lower bound in the PEM model

Brent’s Theorem can often be used to obtain lower bounds for the PRAM model from lower bounds in the RAM model [Bre74; Vis84]. For the PEM model, no such theorem is known. Actually, there are counterexamples that show that there cannot be such a theorem in its full generality [Gre12, Chapter 2].

The permuting lower bounds presented in this thesis are based on the Proxi-mate Neighbors problem, initially defined by Chiang et al. [Ch+i95]. In this section, the permuting lower bound for the Proximate Neighbors problem of Chiang et al. is generalized to the PEM model.
The proof is based on a counting argument: the number of reachable configurations of a PEM machine after \( \ell \) parallel I/Os is compared to the number of input instances of size \( N \) that a program has to distinguish to solve the Proximate Neighbors problem. Thus, there is always at least one hard instance that requires at least \( \ell \) parallel I/Os, no matter which algorithm is used to solve the instance.

In the following, the Proximate Neighbors problem is defined formally.

**Definition 2.7 (Chiang et al. [Chi+95])**

A block permutation describes the content of the shared memory of a PEM configuration. It represents each block of the shared memory by a set of at most \( B \) atoms.

**Definition 2.8**

An instance of the Proximate Neighbors problem of size \( N \) consists of atoms \( x_i \) for \( i \in [N] \). All atoms are labeled by a labeling function \( \lambda: [N] \mapsto [N/2] \) with \( |\lambda^{-1}(i)| = 2 \). An output block permutation solves the input instance if it has the property that for every \( i \in [N/2] \) the two neighboring atoms \( \lambda^{-1}(i) \) are stored in the same block. The blocks in such an output may contain less than \( B \) atoms.

The following lemma is a simple generalization of [Gre12, Theorem 2.7] which is used to prove a lower bound for the Proximate Neighbors problem.

**Lemma 2.9**

Let \( c > 0 \) be a constant, and let \( X \) be a computational problem for which an algorithm has to be capable of generating at least \( \left( \frac{N}{eB} \right)^{cN} \) block permutations. For every input size \( N \), at least half of the input instances of \( X \) require \( \Omega(\text{perm}_P(N, M, B)) \) parallel I/Os in the CREW atomic PEM model with \( P \leq \frac{N}{B} \) processors.

The proof is omitted as it is a straightforward generalization of the proof of Theorem 2.7 in [Gre12]: In the proof of that theorem, on the right-hand side of the considered equations, \( N \log \frac{N}{eB} \) is just replaced by \( cN \log \frac{N}{eB} \), and \( c \) eventually becomes part of the constant hidden in the \( \Omega \)-notation.
The following lemma gives a lower bound on the block permutations that a correct algorithm has to generate to solve any input of the Proximate Neighbors problem of size $N$.

**Lemma 2.10**

Any algorithm that solves all proximate neighbors instances of size $N$ must be capable of producing at least $\frac{(N/2)!}{(B/2)^2}$ different block permutations.

**Proof**  For this proof, the inputs are restricted to a subclass $H$ of the Proximate Neighbors problem: the first $1 \leq i \leq \frac{N}{2}$ atoms are labeled such that $\lambda(x_i) = i$. Hence, there are $\frac{N}{2}$! different input instances of this type, as this is the number of matchings between the first and the second half of the input.

Some of the (seemingly) different inputs of $H$ are solved by the same output block permutation. In the following, the number of different input instances one block permutation can solve is analyzed. A block $i$ in an output block permutation $O$ contains a certain number $b_i$ of neighboring atoms. Naturally there are at most $b_i \leq \frac{B}{2}$ atoms of the first half of the input and $b_i$ atoms of the second half of the input. In the output block $i$, there are $b_i! \leq b_i^{b_i}$ ways to match the atoms of the first half of the input with atoms from the second half of the input. Thus, this block yields a correct solution for at most $b_i^{b_i}$ different inputs.

Note that $\sum_{i \in O} b_i = \frac{N}{2}$. Furthermore, the number of inputs solved by any output block permutation $O$ is $\prod_{i \in O} b_i^{b_i}$. By the previous argument it follows

$$\prod_{i \in O} b_i^{b_i} \leq \prod_{i \in O} \left( \frac{B}{2} \right)^{b_i} = \left( \frac{B}{2} \right)^{\sum_{i \in O} b_i} = \left( \frac{B}{2} \right)^{\frac{N}{2}}.$$  

$\square$

Combining the last two lemmas yields a permuting lower bound for the Proximate Neighbors problem for the PEM model. Thus, the following theorem is a generalization of [Chi+95, Lemma 2.2] to the PEM model:
Theorem 2.11
For every input size $N$, at least half of the instances of the Proximate Neigh-
bors problem require $\Omega(\text{perm}_p(N, M, B))$ parallel I/Os in the CREW atomic
PEM model with $P < N/B$ processors.

Proof Since $n! \geq (n/e)^n$, it follows that \[ \frac{(N/2)!}{(B/2)^{N/2}} \geq \left( \frac{N}{eB} \right)^{N/2} \] [DK03]. Thus, combining Lemma 2.9 with $c = \frac{1}{2}$ and Lemma 2.10 yields the theorem. 

The following remarks on the proof of Theorem 2.11 are noteworthy. It consid-
ers programs, and the copying and the delete operation of the atomic PEM do
not help to solve a problem due to the following observation. A program can
be stripped down to operations that operate only on the atoms which influence
the final result.

Furthermore, since a program only consists of valid operations in the
atomic PEM model, it is independent of the representation of the labeling
or permutation that relates the input to the output. A program may be fully
adapted to a certain permutation or labeling, and thus one can assume that the
program has full information on the permutation or labeling at any time. The
complexity of problems that have a permuting lower bound by this technique
follows from the complexity of moving the atoms to the correct places. But it
does follow from the complexity of obtaining full information of a structure
(permutation or labeling).
CHAPTER 3

List Ranking

The input for list ranking is a linked list on \( N \) vertices. Each vertex in a linked list has exactly one successor. The successor of the end of the list is the last vertex itself. The objective of list ranking is to compute for each vertex its rank, meaning the distance to the end of the list.

List ranking can be solved optimally in the RAM model in linear time by traversing the list. In other computational models, like the PRAM, EM, or PEM model, more advanced techniques have to be used to obtain optimal algorithms. Section 3.1 summarizes the most important techniques that led, over a period of 30 years, to different algorithms for the mentioned models [Wyl79; Vis84; CV86c; AM91; AM90; Chi+95]. Recently, Arge, Goodrich, and Sitchinava presented an algorithm for list ranking that has optimal complexity for many parameter settings of the PEM model. It takes \( O(sort_{p}(N)) \) parallel I/Os in the PEM model for up to \( P \leq \frac{N}{B^{2} \log B \log^{t}(N)} \) processors with some arbitrary constant \( t \geq 1 \), if \( M = B^{O(1)} \) [AGS10].

In Section 3.2 two new algorithms for the list ranking problem are presented that cover all parameter settings of \( P, M, \) and \( B \) of the PEM model. One of them, a randomized algorithm, has sorting complexity for many parameter settings, but in certain cases the algorithm needs \( O(\log^{2} N) \) parallel I/Os, which is in these cases strictly worse than sorting. For showing optimality, we observe that the hardness of list ranking is caused by two factors: (1) a close relationship of list ranking to the problem of permuting data, and (2) limited speed of discovery of the structure of the linked list due to limited information flow among the processors. First, in Section 3.3 it is shown that an information-theoretic lower bound, as known from the PRAM model, applies, by showing that list ranking is capable of solving the Parity problem.
The connection between list ranking and permuting was first pointed out by Chiang et al. [Chi+95]. They sketched a lower bound for list ranking by reducing the Proximate Neighbors problem to it [Chi+95]. As mentioned in Section 2.5.2, their lower bound does not generalize to the PEM model easily since there is no equivalent to Brent’s scheduling principle [Vis84; Bre74] in the PEM model [Gre12].

Therefore, permuting lower bounds are proven in Section 3.4 for two problems that are tightly related to the list ranking problem. They rely on reductions to the Proximate Neighbors problem. Essentially, those lower bounds exploit the fact that the input for a problem can be represented in many different layouts. For proving meaningful lower bounds, the atomic PEM model is extended. The extension allows a program to create new atoms from two arbitrary atoms while still keeping the indivisibility of atoms for proving non-trivial lower bounds.

Proving stronger lower bounds for the list ranking problems is a difficult task. Part of the challenge lies in the difficulty of combining the indivisibility assumption with the restrictions on how information on the structure of the linked list can be extracted without giving the model too much power. This challenge is addressed by the Guided Interval Fusion (GIF) problem. In Section 3.2 it is proven that GIF requires $\Omega(\log^2 N)$ parallel I/Os in a specific parameter setting. This means that the randomized algorithm of Section 3.2 is asymptotically optimal. GIF captures how all currently known techniques for solving list ranking in a parallel or distributed setting use information on the list structure. Therefore, breaking this lower bound requires completely new algorithmic techniques for list ranking. Furthermore, the lower bound for GIF in the PEM model implies an $\Omega(\log N)$ lower bound for the number of rounds for GIF in distributed models when $P = \Theta(M) = \Theta(\sqrt{N})$.

Most of the work presented in this chapter is joint work with Riko Jacob and Nodari Sitchinava. The central results are published [JLS14b; JLS14a]. Therefore, parts of this chapter are taken identically from these publications.

### 3.1 Related work: algorithms for list ranking

The algorithms for list ranking presented in Section 3.2, as well as the algorithm of Arge, Goodrich, and Sitchinava [AGS10], use ideas of several (classical)
3.1 Related work: algorithms for list ranking

Thus, a short chronological summary of the most important list ranking algorithms is presented.

### 3.1.1 PRAM algorithms

List ranking in the RAM model is trivially solvable in linear time. For the PRAM model, it took over a decade to develop work-optimal algorithms that have \( O(\log N) \) complexity for a list of size \( N \).

In 1979, the list ranking problem was considered for the first time in the PRAM model by Wyllie. He presented an \( O(\log N) \) time algorithm which performs \( O(N \log N) \) work. The algorithm, in parallel, repeatedly bridges out the successor of each vertex [Wyl79]. *Bridging out* a set \( S \) of vertices is done by setting for each \( x \in S \) the link of the predecessor of \( x \) to the successor of \( x \).

Many list ranking algorithms for the (P)EM or PRAM models use the same scheme that is organized in rounds. In each round, a fractional independent set \( S \) of vertices of the list is identified that is then bridged out. Repeating this process yields a series of lists of geometrically decreasing sizes. This process is repeated down to a certain size of the list that can be ranked efficiently. The ranks of all elements of the input list can be determined by reversing the process of bridging out elements [AGS10; Chi+95; AM90].

It was claimed that the algorithm of Wyllie is optimal. Vishkin refuted this claim in 1984 with a randomized construction running in \( O((\log N) \cdot \log \log N) \) time using \( \frac{N}{(\log N)^2 \log \log N} \) processors. Thus, this algorithm matches the trivial \( \Omega(N) \) lower bound for the work [Vis84]. The key idea enabling this algorithm, that is still used in many randomized algorithms, is to compute an independent set \( S \) by flipping a coin for each element. A vertex is selected for \( S \) if its coin turned “1” and the coin of its successors turned “0”. The expected size of \( S \) is at least \( \frac{N-1}{4} \).

A year later, Kruskal, Rudolph, and Snir gave the first deterministic algorithm which required \( o(N \log N) \) work [KRS85].

In 1986, Cole and Vishkin introduced the *deterministic coin tossing technique*, which is still the fundamental algorithm for breaking ties deterministically on seemingly identical elements. Given an \( N \)-colored list as input, it computes
an $O(\log N)$-ruling set by using $O(1)$ steps. It is not hard to obtain from this ruling set an independent set of size at least $O(\frac{N}{\log N})$.

The ruling set is defined by an $O(\log N)$-coloring for the list as follows. One of $O(\log N)$ colors is assigned to each vertex of the list by considering the binary representations of the colors of the elements and their successors: let $v$ be a vertex, $s_v$ its successor, $i_v$ the index of the least significant bit in which the binary representation of the colors of $v$ and $s_v$ differ, and $b_v$ the $i_v$-th least significant bit of the binary representation of the color of $v$. Then the color $c_v$ of $v$ is defined as $2 \cdot i_v + b_v$. A vertex is contained in the ruling set if it is the first vertex of the list or if the colors of both its neighbors are larger and it is not the neighbor of the first vertex. An example of computing the colors of the vertices of a list and their rulers is given in Figure 3.1.

Repeating the application of the deterministic coin tossing technique $c$ times, reduces the number of used colors to $\max\{\log^c N, 4\}$. Applying the deterministic coin tossing technique to the list ranking problem yields a deterministic work-optimal algorithm for list ranking that runs in $O((\log N) \cdot \log^* N)$ time in the PRAM model [CV86c; CV86b]. The list ranking algorithms obtained by deterministic coin tossing still suffered from some load balancing issues and have thus suboptimal time complexity.

However, the load balancing issues were shown to be tractable by the use of expander graphs. This established the first deterministic optimal $O(\log N)$ time algorithm for list ranking in the PRAM model [CV88; CV86a]. On the other hand, the expander construction yields huge constant factors in the running time that are hidden in the asymptotic notation. Therefore, this algorithm is not suited for practical implementations. Anderson and Miller gave an easier deterministic construction with reasonable constants [AM91; AM88].

A remarkably easy algorithm was presented by Anderson and Miller [AM90]: Each of the $\frac{N}{\log N}$ CPUs processes a queue of size $\log N$ by bridging out its head element repeatedly. If two processors try to bridge out neighboring vertices, contention is resolved by the coin flipping rule of Vishkin [Vis84]. Anderson and Miller show that all processors have finished their queue after $O(\log N)$ steps with high probability [AM90]. This observation yields an optimal randomized algorithm for list ranking.
3.1 Related work: algorithms for list ranking

Figure 3.1: Deterministic coin tossing or computing a ruling set. Top left: a linked list. Bottom left: computing color classes for each list element. Right: visualization of the color classes and the ruling set.

3.1.2 Algorithms for external memory models

The list ranking algorithms for the EM and the PEM models use basically the same ideas as the previously presented algorithms for the PRAM model. The algorithm for the EM model by Chiang et al. [Chi+95] and the algorithm for the PEM model by Arge, Goodrich, and Sitchinava [AGS10] repeatedly bridge out a fractional independent set of the input list.

List ranking in the EM model. Although the underlying structure of the list ranking algorithm of Chiang et al. for the EM model is the same as for the PRAM model, the challenge is of a different nature. While the PRAM model allows random access to the memory, the access to the memory is coarse grained in the EM model but of sequential nature. Nonetheless, algorithmically, the most difficult part is finding a fractional independent set (FIS) in the list.

An easy algorithm for computing an FIS of a list in the EM model uses a priority queue. A priority queue can be implemented by using an EM-efficient implementation of \((a,b)\)-trees [Arg03; Arg95]. Given a list of size \(N\), a 3-coloring, that implies an FIS of size at least \(\frac{N}{3}\), can be computed in the following way:

### Table: Example of FIS Computation

<table>
<thead>
<tr>
<th>(v)</th>
<th>(s_v)</th>
<th>(i_v)</th>
<th>(b_v)</th>
<th>(c_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>111</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>101</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>001</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>001</td>
<td>110</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>010</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

...
assume that the list is doubly linked, which is not a restriction [Chi+95]. All elements of the list are processed in the (increasing) order of their memory location. Furthermore, a priority queue $Q$ keeps track of which colors may be used for a vertex. If the successor and the predecessor of a vertex $v$ are located in a larger memory location than $v$, then $v$ is assigned an arbitrary color. For each of the neighbors $x$ of $v$ an entry with the memory address of $x$ and the color of $v$ is added to $Q$. Otherwise there is at least one (minimal) entry for the memory location of $v$ in $Q$ (but at most two) which contains the forbidden color(s) for $v$. Thus, choosing a color which is not forbidden and adding for the (memory-location-wise) larger neighbors corresponding entries to $Q$ yields a 3-coloring. Since the $O(N)$ operations on the priority queue need at most $\text{sort}(N) = O(\frac{N}{B} \log \frac{M}{B})$ I/Os [Arg03], $O(\text{sort}(N))$ I/Os suffice to compute the FIS.

Their algorithm takes $\sum_{i=0}^{\infty} O(\text{sort}(c^i N)) = O(\text{sort}(N))$ I/Os for $\frac{2}{3} \leq c < 1$, since repeatedly bridging out elements yields lists of geometrically decreasing sizes [Chi+95].

Note that a priority queue can be used to evaluate topologically sorted circuits with sorting complexity. This technique is called time-forward processing (TFP) [Arg03; Chi+95].

**List ranking in the PEM model**  Arge, Goodrich, and Sitchinava obtained for certain parameter ranges of $P, M$, and $B$ a nearly optimal algorithm for list ranking that finishes in $O(\text{sort}_P(N) \cdot \log^* N)$ parallel I/Os. The algorithm follows from the next lemma, which is a consequence of [AGS10, Lemma 3.2].

**Lemma 3.1**
Let $L$ be a linked list of size $N$ that is given as input. For every $t \in \mathbb{N}$, an $O(\log (t) N)$-ruling set can be computed for $L$ in $O(t \cdot \text{sort}_P(N))$ parallel I/Os in the CREW PEM model by applying $t$ rounds of deterministic coin tossing.

Using Lemma 3.1 with $t = \log^* N$ yields an FIS-algorithm. Bridging out an independent set $S$ can be implemented in the PEM model by a constant number of sorting operations on the list [AGS10, Theorem A.1]. This yields a list ranking algorithm which needs $O(\text{sort}_P(N) \cdot \log^* N)$ parallel I/Os if $P \leq \frac{N}{B^2 \log B}$ and $M = B^{O(1)}$ [AGS10].
3.2 An improved algorithm for list ranking

Delayed pointer processing is a new technique for the PEM model that is similar to time-forward processing on a linked list. It computes an FIS for a list graph using $O(sort_P(N))$ parallel I/Os in the PEM model if $P \leq \frac{N}{B^2 \log(t)}$ and $M = B^{O(1)}$, where $t$ is an arbitrary natural number influencing only the constants hidden by the $O$-notion [AGS10]. Based on delayed pointer processing, Arge, Goodrich, and Sitchinava present an algorithm that ranks a list in only $O(sort_P(N))$ parallel I/Os. This bound is shown for every CREW PEM machine that has $P \leq \frac{N}{B^2 \log B \log(t) N}$ and $M = B^{O(1)}$.

3.2 An improved algorithm for list ranking

For improving the parameter range of the algorithm of Arge, Goodrich, and Sitchinava for list ranking [AGS10], the parameter range of the PEM model is partitioned into two areas. In the first area, an input list is reduced to a list of size at most $P \cdot \min\{\log P, B\}$, by recursively bridging out an independent set to a list of size at most $P \cdot \min\{\log P, B\}$ (cf. Figure 3.2). The resulting list is then ranked by simulating a work-optimal PRAM algorithm. Again, reversing the process of bridging out yields for each element of the input list its rank. Note that bridging out an independent set $S$ can be implemented in the PEM model by a constant number of sorting operations on the list [AGS10, Theorem A.1]. First the details of the PRAM simulation are given.

3.2.1 Ranking a small list

Lemma 3.2

A list of size $N \leq P \cdot \min\{\log P, B\}$ can be ranked by using $O(\log P)$ parallel I/Os in the CREW PEM model.

Proof Let $\mathcal{A}$ be a work-optimal deterministic PRAM algorithm for ranking a list of size $N$ in $O(\log N)$ time steps [AM91; CV88]. This lemma follows from simulating $\mathcal{A}$ in the PEM model, where each processor of the PEM model simulates a processor of the PRAM algorithm and for each time step of $\mathcal{A}$ one parallel I/O is accounted.
Chapter 3 List Ranking

3.2 Making a list small

**Lemma 3.3**

Let \( I \) be an algorithm that computes a fractional independent set of size at least \( \frac{N}{c} \) for a list of size \( N \) in \( O(q \cdot \text{sort}_P(N)) \) parallel I/Os in the CREW PEM model. Then, a list with \( N \geq P \cdot \min\{\log P, B\} \) elements can be reduced to a list that has at most \( P \cdot \min\{\log P, B\} \) elements in \( O(q \cdot \text{sort}_P(N) + (\log P) \cdot \log \frac{B}{\log P}) \) parallel I/Os in the CREW PEM model.
3.2 An improved algorithm for list ranking

Proof Consider the following algorithm which is similar to the algorithm of Arge, Goodrich, and Sitchinava [AGS10, Algorithm 1]. Recursively apply the following procedure until a list of size at most $PB$ is obtained: apply $I$ to compute a fractional independent set $S$ which is then bridged out.

Thus, the number of parallel I/Os to reduce the list to a size of at most $PB$ is given by the recurrence $T(N) = O(q \cdot sort_p(N)) + T(N - \frac{N}{c})$ if $N$ is bigger than $PB$ and at most $O(scan_p(N)) = O(1)$ otherwise. Due to its geometrically decreasing character, $T(N)$ solves to $O(q \cdot sort_p(N))$. Thus, if $B \leq \log P$, the lemma holds.

If $B > \log P$, the list has to be reduced further to a size of $P \cdot \log P$. This is done by the same recursive algorithm. Since $N < PB$, sorting takes $\log P$ parallel I/Os. Furthermore, $O(\log \frac{PB}{P \log P}) = O(\log \frac{B}{\log P})$ rounds of bridging out an independent set are needed. Thus, for this part the I/O-complexity sums up to $q \cdot (\log P) \cdot \log \frac{B}{\log P}$ parallel I/Os. This proves the lemma.

3.2.3 Independent set construction

Lemma 3.1 yields that a fractional independent set can be computed deterministically in $O(\log^* N \cdot sort_p(N))$ parallel I/Os by using the deterministic coin tossing technique [CV86c]. Thus, by combining Lemma 3.2 and Lemma 3.3 with this independent set construction, a deterministic algorithm for list ranking can be obtained.

Theorem 3.4
The number of parallel I/Os to solve the list ranking problem of size $N$ in the CREW PEM model is

$$O\left(\log^* N \cdot \left(\text{sort}_p(N) + (\log P) \cdot \log \frac{B}{\log P}\right)\right).$$
A slightly faster algorithm can be obtained by a randomized independent set construction, following the ideas of Vishkin [Vis84].

**Lemma 3.5 (Arge, Goodrich, and Sitchinava [AGS10, Section III.A])**

For a linked list of size $N$, an independent set whose expected size is at least $\frac{N-1}{4}$ can be computed in the CREW PEM model in $O(sort_P(N))$ parallel I/Os.

Again, by Lemma 3.2 and Lemma 3.3 with the independent set construction of Lemma 3.5 it follows easily:

**Theorem 3.6**

The expected number of parallel I/Os to solve the list ranking problem of size $N$ in the CREW PEM model is

$$O\left(sort_P(N) + (\log P) \cdot \log \frac{B}{\log P}\right).$$

The following lemma considers list ranking algorithm if the delayed pointer processing technique is used. It states the complexity of list ranking algorithm of Arge, Goodrich, and Sitchinava for all settings of $P$ in the parameter range of the PEM model.

**Lemma 3.7 (Arge, Goodrich, and Sitchinava [AGS10, Theorem 3.5])**

Let $t \in \mathbb{N}$. The number of parallel I/Os to solve the list ranking problem of size $N$ in the CREW PEM model with $M = B^{O(1)}$ is

$$O\left(sort_P(N) + B \cdot \log B \cdot \log(t) N \cdot \log \frac{M}{B} \right).$$

Note that the algorithm of Arge, Goodrich, and Sitchinava has sorting complexity if $P \leq \frac{N}{B^2 \log(t) N \log B}$ and $M = B^{O(1)}$. On the other hand the two algorithms presented in this section obtain (near) sorting complexity as long as the term $(\log^* N) \cdot (\log P) \cdot \log \frac{B}{\log P}$ does not exceed complexity of sorting. If $\frac{N}{B(\log B) \log N} \leq P$, the term $(\log^* N) \cdot (\log P) \cdot \log \frac{B}{\log P}$ is asymptotically larger than $sort_P(N)$. Therefore, the algorithms presented here extend the parameter range for list ranking algorithms where it has sorting complexity. Additionally, the algorithms remove the assumption $M = B^{O(1)}$ which seems to be a small cache assumption in contrast to the commonly used tall cache assumption ($M \geq B^2$).
3.3 Lower bound via simple functions

The first lower bound for list ranking is based on an observation on information distribution among processors in the PRAM model [CDR86]. Cook, Dwork, and Reischuk showed for functions \( f \) that have a critical input of length \( n \) a lower bound of \( \Omega(\log n) \) time steps in the CREW PRAM model. A critical input \( I \) for a function \( f \) is a binary input string of \( n \) bits \( x_1x_2\ldots x_n \) such that for each \( 1 \leq k \leq n \) input \( I(k) = x_1x_2\ldots\overline{x_k}\ldots x_n \), which has the \( k \)-th bit flipped, it holds \( f(I) \neq f(I(k)) \). There are many functions which have critical inputs: among others, there is the OR-, AND-, XOR- and the PARITY-function. Arge et al. showed that the argument of Cook, Dwork, and Reischuk generalizes to the PEM model showing that it takes at least \( \Omega(\log N_B) \) parallel I/Os to compute the value of a function over \( N \) bits which has a critical input [Arg+08].

Literature states on several occasions that list ranking is at least as hard as computing the parity over a set of bits by a simple reduction [KR90; KR88]. As this simple reduction does not seem to be stated explicitly in other places it is presented here.

**Lemma 3.8**

A deterministic list ranking algorithm requires \( \Omega(\log N_B) \) parallel I/Os in the CREW PEM model with an arbitrary number of processors.

**Proof** For an instance \( I = x_1x_2\ldots x_n \) of the PARITY problem, a list ranking instance \( L \) of size \( N = 3(n+1) \) is created in \( O(\text{scan}_P(N)) \) parallel I/Os. The graph of \( L \) is given by the following construction. For each bit in \( I \) three edges are created:

- If bit \( x_i = 1 \), the edge \((i, i + 1)\) and the edges \((n + 1 + i, 2(n + 1) + i)\) and \((2(n + 1) + i, n + i + 2)\) are created.

- If bit \( x_i = 0 \), the edges \((i, 2(n + 1) + i), (2(n + 1) + i, i + 1)\), and the edge \((n + 1 + i, n + i + 2)\) are created.

Additionally, the edges \((n + 1, n + 2)\) and \((2(n + 1), 3(n + 1))\) are created. A visualization of the reduction for a small example is given in Figure 3.3. The least significant bit of the rank of vertex \( n + 1 \) solves the PARITY problem. \( \square \)
Figure 3.3: Visualization of the reduction of Lemma 3.8

The following conjecture is due to a stronger lower bound for the Parity problem of Dietzfelbinger, Kutylowski, and Reischuk [DKR94; DKR90]. Unfortunately, their results on the Parity problem have not yet been transferred formally to the PEM model — and will not be transferred in this thesis.

Conjecture 3.9

A (Las Vegas) randomized list ranking algorithm takes at least $\Omega(\log \frac{N}{B})$ parallel I/Os in the CREW PEM model.

3.4 Counting lower bounds for the list ranking problem

In this section a lower bound for the list ranking problem is presented that exploits the permuting nature of the list ranking problem. Based on the hardness of the Proximate Neighbors problem, first it is shown that the Semigroup Evaluation problem has permuting complexity. In Section 3.4.2 the Semigroup Evaluation problem is reduced to contracting a list. Section 3.4.3 concludes by arguing about the relation between contracting a list and the list ranking problem.

3.4.1 Semigroup evaluation

In this section a lower bound for evaluating a very simple type of expressions in the PEM model is given. More precisely, the expressions over a semigroup
3.4 Counting lower bounds for the list ranking problem

are considered. Throughout this section let \( \mathcal{S} \) be a semigroup. Recall that \( \mathcal{S} \) is defined on the set \( S \) that has the associative binary operator \( \cdot : S \times S \rightarrow S \). To prove non-trivial lower bounds for the \textsc{Semigroup Evaluation} problem, the atomic PEM model is extended. The extension allows a program to perform an operation on two atoms that creates a new atom while keeping the indivisibility of atoms and is defined as follows:

**Definition 3.10**
The \textit{semigroup PEM} model is an extension of the atomic PEM model by the binary operation \( s \). Let \( x \) and \( y \) be two atoms in the cache of a processor containing elements \( a, b \) of a semigroup \( \mathcal{S} \). Then, \( s(x, y) \) yields a new atom \( z \) containing the semigroup element \( c = a \cdot b \).

**Definition 3.11**
The input of the \textsc{Semigroup Evaluation} problem is an array \( x \) of input atoms \( x_i \) for \( 1 \leq i \leq N \). Each atom \( x_{\pi(i)} \) contains a semigroup element \( a_i \in S \), where \( \pi \) is a permutation over \([N]\).

An instance \( I_\pi \) of the \textsc{Semigroup Evaluation} problem is solved if an atom is created that contains the semigroup element \( \prod_{i=1}^{N} a_i \). A program is called correct if it solves \( I_\pi \) for any given semigroup.

This section proves that there are \( \Omega(\text{perm}_p(N)) \) parallel I/Os required to solve the \textsc{Semigroup Evaluation} problem in the semigroup PEM model. To this end, the following fact on programs that use only the semigroup operation to solve an instance of the \textsc{Semigroup Evaluation} problem is proven.

**A fact on programs that evaluate an expression over a semigroup**

Consider a program \( E \) to evaluate \( \prod_{j=1}^{N} a_j \). Then, \( E \) defines for every \( i \in [N - 1] \) certain values \( 1 \leq c_i \leq i \) and \( i + 1 \leq k_i \leq N \). The fact states that program \( E \) has to compute for every \( i \) an (intermediate) result \( x_i = y_i \cdot z_i \) such that \( y_i = \prod_{j=c_i}^{i} a_j \) and \( z_i = \prod_{m=i+1}^{k_i} a_m \). For proving this fact, semigroup operations performed by programs are described through graphs that are similar to algebraic circuits.
Definition 3.12
Let $E$ be a program to compute the product $\prod_{i=1}^{N} a_i$ of a semigroup from the inputs $a_j$, with $1 \leq j \leq N$. Its calculation DAG $D_E$ is defined as follows: for each input variable, each intermediate result and the output variable, called result, there is a vertex in $D_E$. For each application of the semigroup operation $z = x \cdot y$ (through $s$) where the operands are represented in $D_E$ by $v_x$, $v_y$, and $v_z$, there are the directed edges $(v_x, v_z)$ and $(v_y, v_z)$ in $D_E$. If there is a directed path from $v_x$ to $v_y$, $x$ is used for computing $y$. A calculation DAG is called normalized if all variables, corresponding to vertices, are used for the result.

A calculation DAG $D$ is called correct if, independent of the underlying semigroup and the values of the input variables, the output variable $r$, corresponding to $v_r$, has the value $\prod_{i=1}^{N} a_i$.

Lemma 3.13
If there is a correct calculation graph $D_E$, then there is a normalized correct calculation graph $D$.

Proof Removing vertices that are not used for the result from $D_E$ cannot change its correctness.

Thus, all calculation graphs can be assumed to be normalized and correct.

Lemma 3.14
Let $b$ an intermediate result in a normalized correct calculation DAG $D$ for a semigroup evaluation of $r = \prod_{i=1}^{N} a_i$. Then, $b$ represents a product $b = \prod_{i=j}^{k} a_i$, with $1 \leq j \leq k \leq N$.

Proof Assume there is an intermediate result $b$ not of the claimed form. Then $b$ can be written as $b = s \cdot a_j \cdot a_k \cdot t$, with $j + 1 \neq k$ and where $s$ and $t$ are arbitrary products over the inputs. By induction, every intermediate $u$ result that uses $b$ can be written as $u = s' \cdot a_j \cdot a_k \cdot t'$, where $s'$ and $t'$ are arbitrary products over the inputs. Since $D$ is normalized, also the result can be written in this form, which contradicts the correctness of the program: Consider for example the concatenation semigroup, $a_j = 'a'$, $a_{j+1} = 'b'$, $a_k = 'c'$ and all other $a_i = \varepsilon$. Then the correct result is 'abc', whereas the computed result contains the substring 'ac'.
Lemma 3.15
Consider a normalized correct calculation DAG $D$ of a semigroup evaluation that computes an (intermediate) result $r = \prod_{j=c}^{d} a_j$ with $1 \leq c \leq d-1 \leq N-1$. Then $D$ defines for every $i \in [c, d-1]$ values $c \leq h_i \leq i$ and $i+1 \leq k_i \leq d$ such that the following is true: for every $i \in [c, d-1]$ a vertex $v_i$ applies the semigroup operation to intermediate results $y = \prod_{j=h_i}^{i} a_j$ and $z = \prod_{m=i+1}^{k_i} a_m$ to compute $x = y \cdot z$.

Proof  Strong induction over $d - c$:
Base case $d - c = 1$: $r = \prod_{j=c}^{d} a_j = a_c \cdot a_d = a_c \cdot a_{c+1}$. ✓
Otherwise, by Lemma 3.14, the computation represented by $v_r$ is $\prod_{j=c}^{d} a_j = (\prod_{m=c}^{\ell} a_m) \cdot (\prod_{o=\ell+1}^{d} a_o)$.
Induction Step: If $i = \ell$: ✓
If $c \leq i < \ell$: By induction the vertex can be found in the intermediate results, calculating $\prod_{j=c}^{\ell} a_j$. ✓
If $\ell + 1 \leq i \leq d - 1$: By induction the vertex can be found in the intermediate results, calculating $\prod_{j=\ell+1}^{d} a_j$. ✓

Note that programs executed in the semigroup PEM model are programs which solve an instance of the Semigroup Evaluation problem by only applying the semigroup operation $\cdot$, respectively $s$.

Proving the lower bound

For the following theorem, it is assumed that all PEM programs are aligned. This means that in each parallel I/O a processor can be either active or idle, and all active processors perform the same type of task (read or write). Note that the I/O-complexity of an aligned PEM program is at most twice as large as the I/O-complexity of an algorithm that implements both operations, read and write, by different processors within the same parallel I/O operation.

The next theorem is the main result of this section:
Theorem 3.16
For every input size $N$ there is at least one instance of the Semigroup Evaluation problem that requires $\Omega(\text{perm}_{P}(N,M,B))$ parallel I/Os in the CREW semigroup PEM model with $P \leq \frac{N}{B}$ processors.

Proof  By Theorem 2.11 there is at least one hard instance of the Proximate Neighbors problem. It is therefore sufficient to provide a non-uniform reduction from the Proximate Neighbors problem to Semigroup Evaluation to prove the theorem.

Construction and idea. Let $I^P_{\lambda}$ be an instance of the Proximate Neighbors problem of size $N$ over the input atoms $X = \{x_i \mid i \in [N]\}$ with its labeling function $\lambda$. In the following, the Semigroup Evaluation problem is considered for the semigroup on the set $X^2$ with the semigroup operation $(a,b) \cdot (c,d) = (a,d)$, where $a,b,c,d \in X$.

The construction of the semigroup instance $I^S_{\pi}$ is visualized in Figure 3.4. The instance $I^S_{\pi}$ of size $N$ of this semigroup evaluation problem has input atoms $\xi_i$ for $1 \leq i \leq N$. Each atom $\xi_i$ contains the semigroup element $a_i = (x_i,x_i)$. Let $\pi$ of $I^S_{\pi}$ be one of the permutations over $[N]$ such that for all $j \in [\frac{N}{2}]$, $[\pi(2j-1),\pi(2j)] = \lambda^{-1}(j)$ holds.

Let $P^S_{\pi}$ be an aligned program solving $I^S_{\pi}$ in $t_S(N,M_S,B_S)$ parallel I/Os for parameters $M_S$ and $B_S$ on $P$ processors. In the following it is shown that $P^S_{\pi}$ can be transformed into a program $P^P_{\lambda}$ using $P$ processors, and solving $I^P_{\lambda}$ by using $t_P(N,M_P,B_P) \leq 5t_S(N,M_S,B_S)$ parallel I/Os, where $M_P = 2M_S$, and $B_P = 2B_S$. Since Theorem 2.11 provides a lower bound for $t_P(N,M_P,B_P)$, a lower bound follows for $t_S(N,M_S,B_S)$.

Figure 3.4: Visualization of the input instance $I^P_{\lambda}$ (left) and the corresponding instance $I^S_{\pi}$ on PEM machines with $B = 2$. The labeling function $\lambda$ is visualized by colors, and the permutation $\pi$ is visualized by the linked list.
Transcribing programs. The key idea is to write for each application of the semigroup operation \((a,b)\cdot(c,d)\) (induced by \(s\)) the pair \(\{b,c\}\) as a result for \(I^P_\lambda\) to the output. Lemma 3.15 yields that, among others, all pairs \(\{a_{\pi(2i-1)}, a_{\pi(2i)}\} = \{x,y\}\), this means, all atoms of \(I^P_\lambda\) with \(\lambda(x) = \lambda(y)\), are written as output.

This may affect the number of parallel I/Os heavily: there may be a processor writing up to \(2M_S\) pairs due to repeated application of the semigroup operation in every parallel I/O. The remainder of this proof argues that for each parallel I/O of \(P^S_\pi\) there are at most two output operations needed to write all pairs to the output. Since \(P^S_\pi\) is aligned, \(I^S_\pi\) is solved in \(t_S(N,M_S,B_S)\) parallel I/Os such that all processors perform either a write operation or a read operation. From this program, the program \(P^P_\lambda\) for solving \(I^P_\lambda\) is constructed. It takes \(t_P(N,M_P,B_P) \leq 3t_S(N,M_S,B_S) + \frac{3N}{P_B P} \) parallel I/Os. More precisely, for each read operation of \(P^S_\pi\) there are at most two additional write operations in \(P^P_\lambda\).

To this end, assume that \(P^S_\pi\) fulfills the following invariant: let the intermediate result \(w = u \cdot v\) be the result of the product of two intermediate results \(u\) and \(v\). Consider a read operation \(t\) after that \(u\) and \(v\) are in the memory of one processor. If \(w\) is used in the result of \(I^S_\pi\), then the invariant says that \(w\) is calculated immediately after \(t\). Conversely, if \(w = u \cdot v\) is used in the result of \(I^S_\pi\), there is no read operation \(t\) such that intermediate results \(u\) and \(v\) are in the memory of one processor, and the semigroup operation is not applied (until the next read operation). Thus, if the invariant holds, for each intermediate result \(v\) in an input block, there are at most two intermediate results \(u\) and \(w\) in a processor’s cache such that \(u \cdot v \cdot w\) has to be computed. Since each processor reads in one I/Os at most \(B_S\) atoms there can be applied at most \(2B_S\) semigroup operations in one I/O. Thus, for each processor, there are at most \(2B_S\) pairs which are written to the shared memory.

Assume, there is an input operation in \(P^S_\pi\) not satisfying the invariant. Then it can be adjusted by applying the semigroup operation \(w = u \cdot v\) after the input operation \(t\). Further semigroup operations, using \(u\) are replaced by using \(w\), while intermediate results \(x = v \cdot y\) are replaced by \(x = y\). This yields by the associativity of the semigroup still the correct intermediate results. Furthermore, this adjustment does not increase the memory consumption of \(P^S_\pi\): the storage used temporarily for \(v\) is not used anymore, and \(u\) is replaced by \(w\), which has the same memory consumption as \(u\).
Because $P^P_\lambda$ is expected to work on full blocks of size $B_P$, but $P^S_\pi$ operates on blocks with $B_S = \frac{B_P}{2}$ semigroup atoms, an additional scan is needed to create blocks of the correct size. This reads one block $x_1, \ldots, x_{B_P}$ and writes two blocks $(x_1, x_1), (x_2, x_2), \ldots, (x_{B_S}, x_{B_S})$ and $(x_{B_S+1}, x_{B_S+1}), \ldots, (x_{B_P}, x_{B_P})$ and thus takes $\frac{3N}{PB_P}$ parallel I/Os. However, $\text{perm}_P(N, 2M_S, 2B_S) \geq \frac{1}{4} \text{perm}_P(N, M_S, B_S)$ is true by Lemma 2.2, and it follows:

$$3t_S(N, M_S, B_S) + \frac{3N}{PB_P} = t_P(N, M_P, B_P) \geq \text{perm}_P(N, 2M_S, 2B_S) \geq \frac{1}{4} \text{perm}_P(N, M_S, B_S).$$

Because every correct program for $t_S(N, M_S, B_S)$ must read the complete input, it holds $t_S(N, M_S, B_S) \geq \frac{N}{PB_S} = \frac{2N}{PB_P}$, such that the above inequality leads to

$$t_S(N, M_S, B_S) \geq \frac{1}{18} \text{perm}_P(N, M_S, B_S).$$

\[\square\]

### 3.4.2 Atomic edge contraction

In order to obtain a permuting lower bound for list ranking, another intermediate problem is defined. The underlying structure is, as in the list ranking problem, a list which is represented by directed edges.

**Definition 3.17**

The *edge-contracting PEM* model is an extension of the atomic PEM model by the binary operation $c$. Let $x$ and $y$ be two atoms in the cache of a processor containing edges $a = (u, v)$ and $b = (v, w)$. Then, $c(x, y)$ yields a new atom $z$ containing the edge $(u, w)$.

**Definition 3.18**

The input of the *Atomic Edge Contraction* problem is an array $x$ of input atoms $x_i$ for $1 \leq i \leq N$. Each input atom $x_i$ contains one of the edges $e_i$ of a directed $st$-path on $N+1$ vertices. The instance is solved if an atom containing the edge $(s, t)$ is created and written to shared memory.
3.4 Counting lower bounds for the list ranking problem

Theorem 3.19
For every input size $N$ there exists at least one instance of the atomic edge contraction problem that requires $\Omega(\text{perm}_p(N, M, B))$ parallel I/Os in the CREW edge-contracting PEM model with $P \leq \frac{N}{B}$ processors.

Proof Let $I^S_\pi$ be an instance of the Semigroup Evaluation problem of size $N$ and $\pi$ its permutation. Let $I^E$ be the instance of the Atomic Edge Contraction problem of size $N$ such that $e_{\pi(i)} = (\pi(i), \pi(i+1))$ for $1 \leq i \leq N$ (assume for completeness that $\pi(N+1) = N+1$). Furthermore, in $I^E$, the atom that is located in the memory cell $j$ of the shared memory contains the edge $e_j$.

This paragraph argues that for each atom of $I^S_\pi$ that contains $a_j$, in $I^E$ there is an atom at the same input position which points to the atom containing $a_{j+1}$ in $I^S_\pi$: the atom $x_i$ at the $i$-th memory position of the input of $I^S_\pi$ contains the semigroup element $a_{\pi^{-1}(i)} = a_j$. By definition, the edge that is contained in the input atom at the $j$-th memory position of $I^E$ is $(j = \pi^{-1}(i), \pi(\pi^{-1}(i) + 1))$. At this target position, in $I^S_\pi$, the atom $x_{\pi(\pi^{-1}(i)+1)}$ that contains the semigroup element $a_{\pi^{-1}(i)+1} = a_{j+1}$ is located.

A program $P^E$ solving $I^E$ is transformed into a program $P^S_\pi$ solving $I^S_\pi$ by the following steps. Replacing in $P^E$ the operation “move atoms containing edges” by “move the atoms containing semigroup elements”, and “merging two atoms containing edges $(a, b)$ and $(b, c)$” by “applying the semigroup operation”, yields program $P^S_\pi$ for solving $I^S_\pi$. By Theorem 3.16 there is an instance $I^S_\pi$ for which there are needed at least $\Omega(\text{perm}_p(N, M, B))$ parallel I/Os to solve it. Thus, there are needed at least $\Omega(\text{perm}_p(N, M, B))$ parallel I/Os for solving $I^E$.

3.4.3 Randomization and relation to list ranking

A randomized algorithm for computational problems is allowed to execute random experiments, using access to unbiased random bits. However, a Las Vegas (randomized) algorithm provides a correct solution to a computational problem independently of the value of the random bits. Since the random experiments may exhibit worst case scenarios only very seldomly, the expected running time of such an algorithm may be faster than the running time of a deterministic algorithm. For example, the worst case running time of the
quick-sort algorithm is $O(N^2)$. If for the pivot rule a random element is selected, the expected running time is in $O(N \log N)$ and thus asymptotically optimal.

Each possible outcome of a random experiment can be seen as an instantiation of a program. Thus, the expected number of parallel I/Os of a Las Vegas (randomized) algorithm can be interpreted as a convex combination of the number of parallel I/Os of programs. Therefore it follows:

**Theorem 3.20**
For each of the problems Proximate Neighbors, Semigroup Evaluation, and Atomic Edge Contraction there exists for every input size $N$ at least one instance that requires $\Omega(\text{perm}_P(N, M, B))$ parallel I/Os for any Las Vegas (randomized) algorithm in the corresponding CREW PEM model with $P \leq \frac{N}{B}$ processors.

In general, the permuting lower bound applies to all algorithms for list ranking that solve the Atomic Edge Contraction problem. The algorithms presented in Section 3.2, solve the list ranking problem by pointer doubling, respectively by bridging out elements. This means that they solve, among others, the atomic edge contraction problem. Therefore the lower bound applies to these algorithms. Furthermore, note that the $\Omega(\log \frac{N}{B})$ lower bound for list ranking of Section 3.3 dominates $\text{perm}_P(N, M, B)$ whenever $P \leq \frac{N}{B}$. By combining those observations it follows:

**Theorem 3.21**
Any deterministic algorithm to solve a list ranking instance of size $N$ that can be used to solve the corresponding instance of the Atomic Edge Contraction problem requires $\Omega(\text{perm}_P(N, M, B) + \log \frac{N}{B})$ parallel I/Os in the CREW PEM model with an arbitrary number of processors.

Combining Theorem 3.20 and Conjecture 3.9 yields the following conjecture:

**Conjecture 3.22**
Any Las Vegas randomized algorithm to solve a list ranking instance of size $N$ that can be used to solve the corresponding instance of the Atomic Edge Contraction problem requires $\Omega(\text{perm}_P(N, M, B) + \log \frac{N}{B})$ parallel I/Os in the CREW PEM model with an arbitrary number of processors.
3.5 The guided interval fusion problem

Actually, to the author’s knowledge, the only technique for solving the list ranking problem is the pointer doubling technique and therefore, the lower bound applies to all known algorithms. This raises the question, in which way algorithms can violate the atomicity assumptions to obtain faster algorithms for the list ranking problem in the PEM model.

A similar discussion applies on the Semigroup Evaluation problem. But in this case it is clear that, for example, assuming commutativity in a semigroup changes the complexity of the task. Commutativity yields directly an algorithm that only has to scan the input and gather the intermediate results from all processors in $O(\log P)$ additional parallel I/Os.

Furthermore, note that the permutation $\pi$ in the Semigroup Evaluation problem can be presented in several ways to an algorithm or a program solving an instance. As in the Proximate Neighbors problem, the lower bounds of Section 3.4.1 and Section 3.4.2 consider programs which move data, as from permuting it. Note that for an instance the structure of the underlying data is fully known to a program. This means that the programs are fully adapted to the underlying permutations of the problems. Therefore the hardness of the lower bounds of this section originates from the fact, that programs have to move atoms in a certain way to solve the considered problems.

3.5 The guided interval fusion problem

This section introduces the Guided Interval Fusion (GIF) problem. GIF is very similar to the Atomic Edge Contraction problem. In this section, for a specific point in the parameter range of the PEM model, a lower bound of $\Omega(\log^2 N)$ parallel I/Os is given for the GIF problem. This complements the upper bound for list ranking of Section 3.2. For obtaining stronger lower bounds, in contrast to the previous section, in the GIF problem an algorithm is not granted unlimited access to the permutation $\pi$.

The PEM model considered in this section operates in one specific point of the parameter range of $P, M,$ and $B$. Precisely, for an input of size $N = 2^x$ with $x \in \mathbb{N}$, a PEM model with parameters $P = M = 2B$ and $N = PM$ is considered. The latter restriction is relaxed in Section 3.5.5 to $N = M^{\frac{3}{2}+\epsilon}$.
The GIF problem is similar to the Atomic Edge Contraction problem, but it considers intervals instead of edges for an easier understandable notation.

3.5.1 Definitions

**Definition 3.23**
An interval PEM is an extension of the atomic PEM: Two atoms $x$ and $y$ containing intervals $I_x$ and $I_y$, located in one cache, can be fused if $I_x \cap I_y \neq \emptyset$. Fusing creates a new atom $z$ containing the interval $I_z = I_x \cup I_y$. An atom $z$ is derived from $x$ if $z$ is the result of zero or more fusing operations starting from atom $x$.

**Definition 3.24**
The Guided Interval Fusion problem is a game between an algorithm and an adversary, played on an interval PEM. The algorithm obtains a GIF instance $G$ of size $N$ which is initially located in the shared memory as an array of uniquely named atoms $x_i$ for $1 \leq i \leq N$. Each initial atom $x_i$ contains the (to the algorithm invisible) closed interval $I_{x_i} = [a-1, a]$ for $a = \pi(i)$ with $1 \leq a \leq N$.

The layout permutation $\pi$ over $[N]$ is gradually revealed by the adversary in the form of boundaries $p = (i, j)$ for the initial atoms $x_i$ and $x_j$, meaning that $x_i$ and $x_j$ contain neighboring intervals (this means, $\pi(j) = \pi(i) + 1$). The adversary must guarantee that at any time, for each existing atom at least one boundary is revealed. The game ends as soon as an atom containing $[0,N]$ exists.

Note the following: copying and deleting of atoms is allowed because the interval PEM extends the atomic PEM. Furthermore, if the algorithm fuses two atoms successfully, their intervals share a point. Therefore, a new atom is created and the corresponding boundary is said to be solved. The algorithm may try to fuse two atoms even though by the revealed boundaries this is not guaranteed to succeed. If a fuse operation succeeds even though the boundary for the initial atoms has not yet been revealed, this phenomenon is called a chance encounter.

For the lower bound, we assume that the algorithm is omniscient. More precisely, we assume that there exists a central processing unit, with unlimited
computational power that has all currently available information on the location of atoms and what is known about boundaries. Based on this information, the unit can decide on how atoms are moved and fused before each parallel I/O.

Thus, as soon as all boundary information is known to the algorithm, the instance is solvable in $O(\log N)$ parallel I/Os: the central unit can virtually list rank the atoms, virtually group the atoms by rank into $P$ groups of consecutive rank. By permuting, to every processor one group of $O(M)$ atoms can be moved. Afterwards, each processor can fuse all elements of its group to one atom in $O(1)$ I/Os. The solving atom can be generated by gathering all intermediate results in $O(\log P)$ parallel I/Os.

Hence, careful revealing of the boundary information is crucial. To define the revealing process for GIF instances, the atoms and boundaries of a GIF instance $G$ of size $N$ are related to a perfect binary tree $T_G$. The tree $T_G$ has $N$ leaves, $N-1$ internal vertices and every leaf is at distance $h = \log N$ from the root. More precisely, the $a$-th leaf in the in-order traversal of $T_G$ corresponds to the atom $x_i = x_{\pi^{-1}(a)}$ that contains the interval $I_{x_i} = [a-1, a]$. Each internal vertex $v_p$ of $T_G$ corresponds to the boundary $p = (i, j)$ such that $v_p$ is the lowest common ancestor of the $a$-th and the $b$-th leaf, where $a = b-1$ and $\pi(a) = i$ and $\pi(b) = j$. The atoms $x_i$ and $x_j$ that correspond to these leaves are called the defining atoms of $p$. Note that every boundary $p$ represents one point $a = b - 1 = [a-1,a] \cap [b-1,b]$. The levels of $T_G$ are numbered bottom up: the leaves have level 1 and the root has level $\log N$ (corresponding to the revealing order of boundaries).

The protocol for boundary announcement, is one of the central building blocks for showing that solving a random GIF instance $G$, with a deterministic algorithm takes $\Omega(\log^2 N)$ parallel I/Os.

**Definition 3.25**

The tree $T_G$ is the guide of a GIF instance $G$ if boundaries are revealed in the following way. Let $x$ be an atom of $G$ containing the interval $I_x = [a,b]$. If neither of the boundaries represented by $a$ and $b$ is revealed, the boundary whose vertex in $T_G$ has smaller level, is revealed. If both have the same level, the boundary represented by $a$ is revealed.

Note that for the analysis it is irrelevant how to break ties in situations when
Chapter 3 List Ranking

the two not yet revealed boundaries have the same level. Recall that the algorithm is omniscient by assumption. Thus, when a boundary \( p \) is revealed, this information is immediately known for all intervals having \( p \) as boundary. Therefore, the guide ensures that at any time for each atom at least one initial atom is known with which it can be fused.

A vertex \( v \in T_G \) is called solved if there is an atom of \( G \) that contains an interval that covers the intervals of the leaves of the subtree of \( v \). The boundary \( p \) is only revealed by the guide if at least one child of \( v_p \) is solved.

An easy (omniscient) algorithm solving a GIF instance can be implemented as follows: in each of \( O(\log N) \) rounds, the atoms are permuted such that for each atom \( x \) there is at least one atom \( y \) in the same cache of a processor such that \( x \) and \( y \) are known to be fuseable. All pairs of neighboring atoms are fused, reducing the number of atoms by a factor of at least 2 and revealing new boundary atoms. Permuting and fusing is repeated until the instance is solved. Because the permuting step can be achieved in \( O(\log P) = O(\log N) \) parallel I/Os, this algorithm finishes in \( O(\log^2 N) \) parallel I/Os. This natural way of solving a GIF instance can be understood as solving in every of the \( \log N \) a proximate neighbors instance. It makes the \( \Omega(\log^2 N) \) lower bound for GIF reasonable as it matches the \( O(\log^2 N) \) upper bound. Note that solving a boundary by fusing resembles bridging out one element of an independent set in the classical list ranking scheme. Thus, most list ranking algorithms use information in the same fashion as it is presented to an algorithm which solves a GIF instance \( G \) guided by \( T_G \) for a randomly chosen permutation \( \pi \).

### 3.5.2 Construction of the proof

The construction for the \( \Omega(\log^2 N) \) lower bound for the GIF problem is given hereafter. To this end, the vertices \( W \) of a certain level \( k \) of \( T_G \) are considered and partitioned into \( s \) progress stages. For each of these stages it is shown that there are \( \Omega(\log N) \) parallel I/Os required to complete the stage.

For every \( e \in W \), the set \( B_e \) is defined as the boundaries that correspond to the vertices in the subtree of \( e \) in \( T_G \). The progress measure towards solving \( e \) is the highest level of a solved boundary in \( B_e \). More precisely, \( e \in W \) is called unsolved on level \( i \) if all boundaries of level \( i \) of \( B_e \) are unsolved. Initially every \( e \)
3.5 The guided interval fusion problem

is unsolved on level 2. The solved level increases by one at a time if only revealed boundaries are solved, but chance encounters may increase it faster.

The execution of a deterministic algorithm defines the following $s$ progress stages. Let $s = k - 1 = h = \log N = \frac{2 \log M}{16}$ and $X = \frac{|W|}{s} = \frac{2^{h+1-k}}{s}$. In each stage $1 < i \leq s$ increase at least $X$ elements of $W$ their level to $i$. Over time, the number of elements of $W$ that are unsolved on level $i$ decreases. The parallel I/O $t_i$ is defined to be the last parallel I/O at which the number of elements of $W$ that are unsolved on level $i + 1$ is at least $|W| - iX$. Additionally, $t_0 = 1$ and $W_0 = W$ is defined. The I/O operation $t_i$ ends stage $i$. To the vertices of $W$ that are unsolved at level $i+1$ after stage $i$ is referred to as $W_i$. Further, let $S_i$ be the vertices (at least $X$) that in stage $i$ (in the time-frame from $t_{i-1}$ to $t_i + 1$) get solved on level $i + 1$ or higher. By the choice of $k - 1 = \frac{h}{16}$ it follows that $X = \frac{2^{h+1-k}}{s} \geq 2^{15h/16}/s = M^{15/8}/s > M^{7/4}$, because $s = \frac{2 \log M}{16} < M^{1/8}$ for $M > 0$.

In the beginning of stage $i$, the solved level of each $v \in W_i$ is at most $i - 1$, and hence all vertices on level $i$ that are a descendant of $W_i$ are not yet revealed to the algorithm. The set $P_i$ is defined as the boundaries for which progress is traced as follows: for every $e \in S_i$ there are $2^{k-i}$ vertices $L_e$ of level $i$ in the subtree of $e$. Let $v_e$ be the vertex of $L_e$ that is solved first (break ties arbitrarily). Each $v_e$ corresponds to one boundary $p_e$. The set $P_i$ consists of all those boundaries $p_e$ of vertices $e \in S_i$. Let $a_e$ and $b_e$ be the two defining atoms of $p_e$. Then all intervals having boundary $p_e$ are derived from $a_e$ or $b_e$. Solving the boundary $p_e$ means fusing an interval derived from $a_e$ with an interval derived from $b_e$. Furthermore, a traced boundary $p$ is considered solved if in

![Figure 3.5: Visualization of the traced vertices of a guide of a GIF problem.](image)
its interval $I$ (the one corresponding to an element of $W$) a chance encounter solves a boundary of level greater than $i$, although $p$ might not be covered by $I$.

To trace the progress of the algorithm towards fusing the atoms of one stage, the graph $G^i_t = (V, E^i_t)$ is defined from the block and cache contents of the interval PEM after $t_i + t$ parallel I/Os. For each cache and each non-empty block of the shared memory (independent of $t$), there is a vertex in $V(G^i_t)$. There is an edge (self-loops allowed) $\{u, v\} \in E^i_t$ if for some $e \in W_i$ some atom derived from $a_e$ is at $u$ and some atom derived from $b_e$ is at $v$ or vice versa. The multiplicity of an edge counts the number of such $e$. The multiplicity of the graph is the maximal multiplicity of an edge.

Note that a boundary $p_e$ is solved if there is a vertex $v_e$ of $T_G$ that counts as a self-loop in some graph. Hence, the sum of the multiplicities of self-loops are an upper bound on the number of solved vertices, and for the stage to end, this means at time $t_{i+1} + 1$ the sum of the multiplicities of loops must be at least $X$. After each parallel I/O, chance encounters may happen. The number of chance encounters is given by $P$ times the multiplicity of self-loops at the beginning of the stage since they can essentially occur only after a read operation.

Two vertices of $T_G$ are called indistinguishable if they are on the same level and exchanging them could still be consistent with the boundary information given so far. Boundaries that correspond to vertices which have higher level than $k$ may be announced or solved (not only due to chance encounters) and thus they may be distinguishable for the algorithm. To account for that, we assume that all such boundaries between the intervals corresponding to $W$ are solved. Hence, the algorithm is aware of the leftmost and rightmost solved interval belonging to these boundaries, and this may extend to other intervals by revealed boundaries. Only the vertices of level $i$ that correspond to this leftmost or rightmost interval might be identifiable to the algorithm, all other vertices of level $i$ are indistinguishable. Because $i < k$, for all traced pairs $a_e, b_e$ at least one of the elements belongs to this big set of indistinguishable vertices. We mark identifiable vertices. Hence, at stage $i$ the algorithm has to solve a random proximate neighbors instance of the traced pairs where each marked vertex has a neighbor that is unmarked. Note that this proximate neighbors instance corresponds to a matching where all marked vertices are matched with unmarked ones and unmarked ones might be matched with marked or unmarked ones.
The analogy between a proximate neighbors instance and a matching can be used to show that the multiplicity of a graph $G_i^0$ is bounded. The following corollary states that in the beginning of each progress stage $i$ the multiplicity of $G_i^0$ is bounded by $M^{5/8}$ with high probability. Its proof is deferred to the proof of the more general Lemma 3.36 in Section 3.5.5 for not repeating the very similar arguments.

**Corollary 3.26**

Let there be an interval PEM with $P = M = 2B = \sqrt{N}$, and let $N = 2^x$ for $x \in \mathbb{N}$. Furthermore, let $G$ be a GIF instance of size $N$ for a uniformly chosen layout permutation $\pi$, and let $T_G$ be its guide. Consider in this interval PEM an arbitrary deterministic GIF algorithm $A$ that operates on $G$. Assume that $A$ is by the previously defined stages in stage $i$ and thus $G_i^0$ represents the current state of the interval PEM. Let $p(i,M)$ be the probability over all inputs of $A$ that lead to $G_i^0$ such that $G_i^0$ has multiplicity at most $M^{5/8}$. Then there is an $x'$ such that for all $x \geq x'$ and for each $i$ it holds $p(i,M) \geq 1 - \frac{1}{M^2}$.

The following lemma states that the increase of the multiplicity caused by one parallel I/O on any $G_i^i$ cannot be too large.

**Lemma 3.27**

If the multiplicity of graph $G_i^i$ is at most $m$, then the multiplicity of $G_{i+1}^i$ is at most $4m$.

**Proof** The parallel I/O operation $t_i + t + 1$ that yields $G_{i+1}^i$ consists of several read and write operations. Each I/O of a processor is considered on its own and can be a read or a write operation.

If the I/O operation is a write operation, the graph $G_{i+1}^i$ can differ from $G_i^i$ only by creating a new vertex $v_n$, or replacing a vertex $v_r$. If $v_n$ is created, the multiplicity of the edges incident to the vertex $v$ of the cache of the processor is partitioned among vertex $v_n$ and $v$. Furthermore, if $v_r$ has been replaced, this cannot increase the multiplicity. Thus, in any case, a write operation cannot increase the multiplicity.

If the I/O operation is a read operation, then each vertex representing a cache of a processor can be identified with one vertex representing a block. In the
Chapter 3 List Ranking

worst case, the new edge considered is between two such united vertices. Thus, the multiplicity can stem from four former edges \(e_1, e_2, e_3,\) and \(e_4\) in Figure 3.6).

![Figure 3.6: Contracting two edges of a matching in \(G_i^i\).](image)

Considering a self loop, it can stem from two self-loops and one edge \((m_1, \ell_1,\) and \(\ell_2,\) respectively \(m_2, \ell_3,\) and \(\ell_4\) in Figure 3.6). Hence, the multiplicity of each edge of \(G_{i+1}^i\) is at most four times the multiplicity of \(G_i^i\).

\[ \square \]

**Lemma 3.28**

Let there be an interval PEM with \(P = M = 2B = \sqrt{N},\) and let \(N = 2^x\) for \(x \in \mathbb{N}\). Furthermore, let \(G\) be a GIF instance of size \(N\) for a uniformly chosen layout permutation \(\pi,\) and let \(T_G\) be its guide. Consider in this interval PEM an arbitrary deterministic GIF algorithm \(A\) that operates on \(G.\) Assume that \(A\) is by the previously defined stages in stage \(i + 1\) and that \(t_i < \log^2 N.\) Let \(p(i, M)\) be the probability that stage \(i\) took \(\Omega(\log N)\) parallel I/Os. Then there is an \(x'\) such that for all \(x \geq x'\) and for each \(i\) it holds \(p(i, M) \geq 1 - \frac{1}{M}.\)

**Proof** At time \(t_{i-1}\), the multiplicity of \(G_{i-1}^i\) is by Corollary 3.26 at most \(M^{\frac{5}{8}}\) with probability \(1 - \frac{1}{M^2}.\) Therefore, by Lemma 3.27 the multiplicity of \(G_{i-1}^i\) is at most \(4^t M^{\frac{5}{8}}\) with probability \(1 - \frac{1}{M^2}.\) By the assumption \(t_i < \log^2 M,\) the number of vertices in \(G_{i-1}^i\) is \(P \log^2 M.\) Hence, at time \(t_{i-1} + t,\) the total multiplicity of loops in \(G_{i-1}^i\) is \(M^{\frac{5}{8}} P 4^t \log^2 M.\) By taking chance encounters into account, the total multiplicity of loops in \(G_{i-1}^i\) is bounded by \(M^{\frac{5}{8}} P (4^t + t) \log^2 M.\) For the stage to finish, this number must be at least \(X > M^{\frac{7}{2}}.\) Hence, \(M^{\frac{5}{8}} P (4^t + t) M^{\frac{7}{2}} \log^2 M > M^{\frac{7}{2}}\) implies \(4^t + t > M^{\frac{7}{2}} - \frac{3}{\log^2 M}\) and hence \(t = \Omega(\log M).\)

By definition of \(s\) and \(k,\) there are \(\Theta(\log N)\) stages. Each stage takes \(\Omega(\log N)\) parallel I/Os. Thus, a union bound over stages \(i < s\) yields:
3.5 The guided interval fusion problem

Lemma 3.29
Let there be an interval PEM with \( P = M = 2B = \sqrt{N} \), and let \( N = 2^x \) for \( x \in \mathbb{N} \). Furthermore, let \( G \) be a GIF instance of size \( N \) for a uniformly chosen layout permutation \( \pi \). Then there is an \( x’ \) such that for all \( x \geq x’ \), solving \( G \) in the interval PEM requires \( \Omega(\log^2 N) \) parallel I/Os with high probability (\( p > 1 - 1/M \)).

By Yao’s principle [Yao77] this lemma implies a lower bound for Las Vegas randomized algorithms:

Theorem 3.30
Let there be an interval PEM with \( P = M = 2B = \sqrt{N} \) and \( N = 2^x \) for \( x \in \mathbb{N} \). The expected number of parallel I/Os to solve a GIF instance of size \( N \) with a Las Vegas randomized algorithm on such an interval PEM is \( \Omega(\log^2 N) \).

3.5.3 Significance of the lower bound

The GIF problem is an attempt to formulate how the known algorithms for list ranking use information by revealing it gradually. Most known algorithms for list ranking use fusing respectively contracting of edges on an independent set of edge-pairs (bridging out edges). This means that every edge is used (if at all) as one of the two atoms in the fusing. This choice of the algorithm is taken without complete information on the list structure. Hence it is reasonable to replace it by an adversarial choice, leading to the definition of the guide of a GIF instance. Note that presenting boundaries to the algorithm corresponds in list ranking algorithms to presenting the algorithm an independent set for free.

Allowing chance encounters and providing for every edge at any time some possible fuse partner should be seen as making GIF to solve as easy as possible. Allowing both concepts simultaneously might give the algorithm a chance to use some kind of pipelining, similar to the efficient PRAM sorting algorithms [AKS83; Pat90]. However, this lower bound indicates that this seems difficult for list ranking since there is no efficient possibility to perform different stages (matchings) in parallel.

At this stage, the lower bound is more a bound on a class of algorithms. It remains a challenge to formulate precisely what this class is. This raises the
question, in which way algorithms can violate the atomicity assumptions to obtain faster algorithms for the list ranking problem in the PEM model. Additionally, it would be nice to show lower bounds in a less restrictive setting, like giving up the all atomicity assumptions and allowing each atom to consist of \( \log N \) bits.

**List ranking in distributed models**

The best known solution for list ranking in the MapReduce model is via the simulation results of Karloff, Suri, and Vassilvitskii [KSV10]. They simulate a \( \mathcal{O}(\log N) \) time work-optimal PRAM algorithm [AM91], yielding \( \mathcal{O}(\log P) \) rounds in the MapReduce model. Thus, the number of communication rounds needed for list ranking is strictly worse than both sorting and permutation, which need \( \mathcal{O}(1) \) communication rounds.

Similar to the MapReduce model, in the BSP model, the number of communication rounds needed to solve list ranking is \( \mathcal{O}(\log P) \) [Deh+02; Các+97]. It uses the deterministic coin tossing technique to create smaller lists, which then can be ranked sequentially by each processor.

Up to now, no non-trivial lower bounds, this means, stronger than \( \Omega(\log M P) = \Omega(1) \) communication rounds, are known for list ranking in the BSP and MapReduce models. An easy reduction yields a lower bound for a certain parameter setting in the BSP and MapReduce model.

**Theorem 3.31**

Solving the GIF problem of size \( N \) in the BSP or in the MapReduce model with \( N = PM \) and \( P = \Theta(M) = \Theta(\sqrt{N}) \) takes \( \Omega(\log N) \) communication rounds.

**Proof** Communication of each round in the distributed models can be implemented in the PEM model in \( O(\perm_P(N)) \) parallel I/Os [GJ12; Gre12]. In the considered parameter setting evaluates \( \perm_P(N) \) to \( O(\log P) \). Thus, \( o(\log N) \) communication rounds in the BSP or the MapReduce model with \( P = \Theta(M) = \Theta(\sqrt{N}) \) would imply \( o(\log^2 N) \) parallel I/Os in the interval PEM model. \( \square \)
3.5.4 On usability of information

As mentioned in Section 3.5.1, if the complete permutation $\pi$ of a GIF instance $\mathcal{G}$ is revealed at once to an (omniscient) algorithm, $\mathcal{G}$ can be solved in $O(\log N)$ parallel I/Os by virtually list ranking the instance. Therefore the process of information revealing turned out to be crucial and led to the definition of the guide of a GIF instance (Definition 3.25). Note that presenting $\pi$ at once is equivalent to revealing for each initial atom $x$ the value $a$ of its interval $I_x = [a, b]$. For the sake of short notation, here, this point $a$ is called the rank of an atom. Note that the value of $a$ implies the value of $b = a + 1$ and vice versa. Therefore, the point $b$ could be defined as the rank of an atom as well.

This section shows how the amount of known information on an instance, measured in bits of the rank per atom, influences the hardness of the GIF problem in the parameter setting of the previous section ($\sqrt{N} = M = P = 2B$).

Fast algorithms despite incomplete information

In the following, three algorithms are presented that show how a GIF instance can be solved in the PEM model in $O(\log N)$ parallel I/Os if for each atom the same $\frac{1}{2}\log N = \log P$ bits of its rank are known.

Most significant $\log P$ bits are given

Assume for each atom there are (at least) the most significant $\log P$ bits of its rank known. This partitions the atoms into $P$ sets $U_i$ such that all atoms in $U_i$ have the same most significant $\log P$ bits, where $1 \leq i \leq P$.

The $M$ atoms of $U_i$ represent $M$ consecutive intervals since the least significant $\log P = \log M$ bits are different for each pair in $U_i$. Thus, each processor $i$ can read and store its complete set $U_i$ into its cache. This can be done by permuting the input in $O(\log N)$ parallel I/Os. By using chance encounters, each processor can fuse its $M$ atoms to one atom $u_i$ which is written to the shared memory. This step takes for all processors in total only one parallel I/O. By collecting all intermediate results $u_i$ in one processor, the solving atom can be created. While collecting takes at most $O(\log P)$ parallel I/Os, the final fusing takes at most one other parallel I/O. This yields the following lemma.
Lemma 3.32
Let $P = M = 2B = \sqrt{N}$, and let $\mathcal{G}$ be a GIF instance of size $N$. Assume that for each atom of $\mathcal{G}$ the most significant $\log M$ bits of the rank are known. There exists an algorithm that solves $\mathcal{G}$ in $O(\log N)$ parallel I/Os in the CREW PEM model.

Note that this yields an algorithm which takes $O(\log N)$ parallel I/Os for instances where a $\log P - 1$ approximation for the rank of each atom is known. Such an approximation may be given for example by an oracle. Furthermore, in contrast to the following algorithms, this algorithm does not have to be omniscient.

Least significant $\log P$ bits are given

The next two algorithms exploit that the lower bound of Section 3.5.2 admits the algorithm to be omniscient. It is unclear if the assumption of the omniscience of the following two algorithms can be removed. Nonetheless, they show the limits of the lower bound for the GIF problem.

Lemma 3.33
Let $P = M = 2B = \sqrt{N}$, and let $\mathcal{G}$ be a GIF instance of size $N$. Assume that for each atom of $\mathcal{G}$ the least significant $\log M$ bits of the rank are known. Then there is an omniscient algorithm $A$ that solves $\mathcal{G}$ in $O(\log N)$ parallel I/Os in the CREW PEM model.

Proof  
As argued in Section 3.5.1, if all neighborhood relationships respectively boundary information among the atoms are discovered, an omniscient algorithm can solve $\mathcal{G}$ in $O(\log N)$ parallel I/Os. Therefore, this proof argues how the neighborhood relationships can be revealed by using chance encounters efficiently due to the given rank information.

Again, the rank information partitions the input into $M$ sets $L_i$ for $1 \leq i \leq M$. Each set $L_i$ contains $M$ atoms. Thus, by permuting, to each processor $i$ a set $L_i$ can be assigned, each containing all atoms $x_j$ with $\text{rank}(x_j) \mod P = i$.

For an atom $x_j$ which is a neighboring atom of an atom contained in $L_i$ it holds $\text{rank}(x_j) = i + 1 \mod P$ or $\text{rank}(x_j) = i - 1 \mod P$. Therefore, for
all $1 \leq i \leq M$ all neighbors of $L_i$, are stored in $L_{(i-1) \mod p}$, called successors, and $L_{(i+1) \mod p}$, called predecessors.

All neighborhood relationships among the atoms of $L_i$ and $L_{(i+1) \mod p}$ can be revealed by using $6 = O(1)$ I/Os on one processor, since $2B = M$. Such a (brute force) search for chance encounters in two sets is called here a *chance encounter scanning round*. Thus, all neighborhood relationships among the predecessors and the successors for all $L_i$, and thus all neighborhood relationships of the input atoms can be revealed in parallel by one chance encounter scanning round in $O(1)$ parallel I/Os.

Note that the concept of a *chance encounter scanning round* among the atoms of two sets $L_i$ and $L_j$ is used in the following algorithm, too.

**Any fixed set of $\log P$ bits is given**

The two previous algorithms considered two special cases of sets of bits that are given for the rank of each atom of a GIF instance: the sets $\{1, ..., \log P\}$ and $\{\log P + 1, ..., 2\log P\}$. If those bits are known for each atom of a GIF instance, it can be solved in $O(\log N)$ parallel I/Os. Lemma 3.34 shows that this is true even if for each atom the same arbitrary chosen $\log P$ bits of the rank are known.

In the following we consider a GIF instance $G$ such that for each atom a certain set of bits of the rank is given, namely $W(N) = \{1, 3, 5, ..., 2\log P - 3, 2\log P - 1\}$. In this section we argue that if there is a fast algorithm solving $G$, then for any $\log P$-sized subset of $[2\log P]$ there is a GIF algorithm that takes $O(\log N)$ parallel I/Os. Again, the given bits partition the input atoms into sets $A_i$ for $1 \leq i \leq M$. Each set $A_i$ has size $M$. The previous algorithm used the fact that each set $A_i$ had neighboring atoms in only two other sets. However, the atoms of set $A_i$ may have neighboring atoms in more than two set $A_j$. This is visualized in the graph $G_{W(N)}$, which indicates if there is an atom in a set $A_i$ that has a neighboring atom in $A_j$. This graph is shown in Figure 3.7 and Figure 3.8 for $W(2^{10})$.

In the previous case, where the $\log P$ least significant bits were given, this graph was just a circle $C_M$ with $M$ vertices. The algorithm covered by one chance encounter scanning round all edges of $C_M$. Here, the strategy of the algorithm...
Figure 3.7: $W(H_{310})$: Each vertex $i - 1$ represents a set $A_i$ whose bits are set to the binary representation of the vertex name. An edge indicates neighboring atoms in the sets.
3.5 The guided interval fusion problem

Figure 3.8: $W(H_{210})$: Each vertex $i - 1$ represents a set $A_i$ whose bits are set to the binary representation of the vertex name. An edge indicates that $A_i$ and $A_j$ contain neighboring atoms. Edges which are only increase edges are dashed.

that solves $G$ is to cover all edges of $G_{W(N)}$ with a constant number of chance encounter scanning rounds.

**Lemma 3.34**

Let $P = M = 2B = \sqrt{N}$, and let $C \in \left(\frac{\log P}{\log P}\right)$. Furthermore, let $G$ be a GIF instance of size $N = 2^x$ for $x \in \mathbb{N}$. Assume that for each atom of $G$, the bits $C$ of the rank are known. There is an omniscient algorithm $A$ that solves $G$ in $O(\log N)$ parallel I/Os in the CREW PEM model.

**Proof** Consider the graph $G_{W(N)}$. There are two types of edges which are defined in the next paragraph by considering a binary adder.

To distinguish the edges of the two classes, consider two neighboring atoms $a$ and $b$ of $G$. Let $\text{rank}(a) + 1 = \text{rank}(b)$ and consider the rank restricted to the given bits $\text{rank}|_{W(N)}(a)$, respectively $\text{rank}|_{W(N)}(b)$. Then $(\text{rank}|_{W(N)}(a)) + 1$ equals $\text{rank}|_{W(N)}(b)$ if and only if the least significant 0-bit of $\text{rank}|_{W(N)}(a)$ is also the least significant 0-bit in $\text{rank}(a)$. In this case an (undirected) *increase edge* exists in $G_{W(N)}$. If the least significant 0-bit of $\text{rank}|_{W(N)}(a)$ is not the least significant 0-bit in $\text{rank}(a)$, then an (undirected) *decrease edge* exists in $G_{W(N)}$. 

63
Note that the previous two algorithms only had to cover increasing edges. Furthermore, observe the set of increase edges $I$ is not disjoint from the set of decrease edges $D$.

The main argument of this proof is that $D$ forms a tree, which leads to a fast scheme for chance encounter scanning: each vertex $v$ of a rooted tree has only one parent $p$. Thus, the processor assigned to $v$, with the partition $U_v$, can scan for chance encounters in $U_p$, the partition assigned to processor $p$. Therefore, any rooted tree can be covered with one round of chance encounter scanning. In combination with the previous algorithm all edges of $G_{W(N)}$ can be discovered in $O(1)$ rounds of chance encounter scanning, solving the problem for $W(N)$.

By induction over $|W(N)| = n$ it follows that $D$ for $W(N)$ forms a binomial tree $B_{|W(N)|}$. Base case: for $n = 1$ and $n = 2$ see Figure 3.9a and Figure 3.9b.

Induction Step $n \to n + 1$:

Let $x$ be the most significant bit of $W(N)$. First observe that there is only one decrease edge from vertices representing a partition with $x = 1$, to vertices representing a partition with $x = 0$. This edge is from the class $\sqrt{N} - 1 = 2^n - 1$ to class 0 and is only created if all bits, including the bit $x$, are set to 1. All other decreasing edges only occur among vertices which have the same bit $x$.

Therefore, it follows by induction that the subgraph of $D$ on the vertices where $x = 0$ is a $B_n =: A$. Also the subgraph of $D$ with $x = 1$ is a $B_n$ by induction. Together with the edge from vertex $\sqrt{N} - 1$ to vertex 0, the root of $A$, this is just the definition of a binomial tree $B_{n+1}$ (see Figure 3.10).

Thus, the lemma holds for $W(N)$. However, it can be generalized to arbitrary log $P$ sized sets $W$ of the $2 \log P$ bits of the rank by the following argument. Consider a class $U$, and let $i$ be the least significant bit of the fixed bits for $U$ which is 0. A decrease edge from a class $U$ to a (smaller) class can exist to at most $i$ other classes if the following occurs: a carry bit may not be propagated (due to not fixed bits which are 0) to the $i$-th fixed bit. All such decrease edges are covered by $G_{W(N)}$ and thus the presented algorithm works for any log $P$ sized subset of fixed bits.  \qed
3.5 The guided interval fusion problem

<table>
<thead>
<tr>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$\text{Edge} \in D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>${0,1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>${0,1}$</td>
</tr>
</tbody>
</table>

(a) Case $n = 1$

<table>
<thead>
<tr>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$\text{Edge}$</th>
<th>$\in D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$[0,1]$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$[0,1]$</td>
<td>✓</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$[1,0]$</td>
<td>✓</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$[0,1]$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$[1,2]$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$[2,3]$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$[3,2]$</td>
<td>✓</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$[2,3]$</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$[3,0]$</td>
<td>✓</td>
</tr>
</tbody>
</table>

(b) Case $n = 2$

Figure 3.9: Base cases for the induction in the proof of Lemma 3.34.

Figure 3.10: Definition of binomial trees.
Hardness results on the number of given bits

In this section, it is shown that if the number of given bits of the rank of each atom is smaller than \( \frac{1}{2} \log M \), there are still required \( \Omega(\log^2 N) \) parallel I/Os to solve the instance. This complements the algorithmic results for the case that many bits of the rank for each atom are given.

**Lemma 3.35**

Let \( C \) be for any \( \frac{1}{2} \log M \geq \varepsilon \geq 1 \) a subset of \([2 \log P]\) of size at most \( \frac{\log M}{2} - \varepsilon \). Furthermore, let \( G \) be a GIF instance of size \( N \) such that for each atom only the bits \( C \) of the rank are known. Then it takes \( \Omega(\log^2 M) \) parallel I/Os to solve \( G \) on an interval PEM with \( P = M = 2B = \sqrt{N} \).

**Proof**  The proof considers the execution of an algorithm to solve \( G \). From this, a program is generated to solve a smaller GIF instance \( S \) with \( N' = M^{\frac{3}{2} + \varepsilon_c} \) on an interval PEM with the same parameters for \( M, B \) and \( P \). As shown in Section 3.5.5 (Theorem 3.40), it requires for every \( \varepsilon_c > 0 \), \( \Omega(\log^2 M) \) parallel I/Os to solve \( S \) on such a PEM.

The instance \( S \) has to be defined more precisely, depending on the order of the atoms of \( G \). For \( \varepsilon_c = \frac{\varepsilon}{\log M} \), let \( x|C \) be the \( 2^{\frac{3}{2} \log M + \varepsilon} = M^{\frac{3}{2} + \varepsilon_c} \) sized (and order preserving) array of atoms of the input array \( x \) of \( G \) such that the bits \( C \) of the rank of these atoms are all set (without loss of generality) to 0. Then, the rank of the \( i \)-th input atom of \( S \) is the value of the binary representation of the rank of the \( i \)-th atom in \( x|C \) with removed bits \( C \). This relates each input atom of \( S \) to a specific atom of \( G \).

Note that by one permutation of size \( M^2 \) the input elements of \( S \) can be written to the positions of their corresponding input atoms \( G|C \) of \( G \). Let \( A \) be a program for solving \( G \) which only contains operations on atoms derived from \( G|C \). Then applying \( A \) of \( G \) to the distributed input of \( S \) solves \( S \).

Due to Theorem 3.40, solving an instance of size \( M^{\frac{3}{2} + \varepsilon_c} \) requires \( \Omega(\log^2 M) \) parallel I/Os. Solving \( S \) by one permutation and \( A \) cannot be faster and thus solving \( G \) requires \( \Omega(\log^2 M) \) parallel I/Os, too. \( \square \)
3.5 The guided interval fusion problem

3.5.5 Improving the lower bound for GIF

Considering the construction of the proof of Lemma 3.29, one of the proof requirements is that \( N = M^2 \). This condition is needed, since the level \( k \) is chosen such that \( |W| = M^{15/8} \) and there are \( \frac{2\log M}{16} \) stages considered by Lemma 3.29 yielding \( N \geq |W| \cdot 2^s = M^{15/8 + 1/8} \). By small changes, Theorem 3.40 shows that the condition \( N = M^2 \) can be relaxed to \( M^2 \geq N \geq M^{3/2} + \varepsilon \). One first step in improving the condition can be made easily by reducing the number of stages \( s \) to \( \frac{2\log M}{32} \) but keeping the size of \( W \). Then it follows directly that \( N \geq M^{15/8} \cdot 2^s = M^{15/8 + 1/16} = M^{2 - 1/16} \) suffices.

The overall strategy is to maximize the number of possible stages, but then use only a fraction as small as possible of it. To this end, the same construction of the proof as described in Section 3.5.2 is used. The next paragraph shows the choice of the parameters \( s \) and \( k \) and some consequences.

For any \( \frac{1}{2} \geq \varepsilon > 0 \) with \( N = M^{3/2 + \varepsilon} \) choose \( \varepsilon_t = \frac{3}{4} \varepsilon, \varepsilon_s = \varepsilon_r = \varepsilon_m = \frac{1}{4} \varepsilon \) and \( \varepsilon_X = \varepsilon_t - \varepsilon_s \). Observe: \( \varepsilon = \varepsilon_t + \varepsilon_s \) and \( \varepsilon_t = \varepsilon_s + \varepsilon_r + \varepsilon_m \). For the remainder of this section let \( k - 1 = (\varepsilon - \varepsilon_t)\log M \). Thus, \( |W| = M^{3/2 + \varepsilon_t} \). Furthermore, by choosing \( s = \varepsilon_s \log M \) and observing that \( s \in O(M^{\varepsilon_s}) \), it holds for sufficiently large \( M \): \( X = \left\lfloor \frac{|W|}{s} \right\rfloor \geq M^{3/2 + \varepsilon_X} \).

Before showing that even by this choice of \( s \) and \( k \) each progress stage requires \( \Omega(\log M) \) parallel I/Os, the lemma that provides with high probability an upper bound on the multiplicity of a graph \( G_i^0 \) is presented. An easy implication of this lemma is the previously used Corollary 3.26.

**Lemma 3.36**

Let there be an interval PEM with \( P = M = 2B \), and let \( N = 2^x \) for \( x \in \mathbb{N} \). Furthermore, let \( G \) be a GIF instance of size \( M^2 \geq N \geq M^{3/2} \) for a uniformly chosen layout permutation \( \pi \), and let \( T_G \) be its guide. Consider in this interval PEM an arbitrary deterministic GIF algorithm \( A \) that operates on \( G \). Assume that \( A \) is by the previously defined stages in stage \( i \) and thus \( G_i^0 \) represents the current state of the interval PEM. Let \( p(i,M) \) be for any \( 1 \geq \varepsilon_m > 0 \) the probability over all inputs of \( A \) that lead to \( G_i^0 \) such that \( G_i^0 \) has multiplicity at most \( M^{1/2 + \varepsilon_m} \). Then for every \( \varepsilon_m \) there is an \( x' \) such that for all \( x \geq x' \) and for each \( i \) it holds \( p(i,M) \geq 1 - \frac{1}{M^x} \).
Proof In the following, an even even stronger result is shown: the probability that $G_0^i$ has multiplicity at most $M^{1/2} \sqrt{5 \ln M}$ is at least $1 - \frac{1}{M^2}$. Since for every $1 \geq \epsilon_m \geq 0$, $\sqrt{5 \ln M}$ is in $O(M^{\epsilon_m})$ the lemma follows.

By the previous definitions, there are at least $|W_i|$ boundaries that are solved in stage $i$. Since each boundary corresponds to an edge in $G_0^i$, there are at least $2|W_i| > M^{3/2} + \epsilon_X > M^{3/2}$ vertices of $T_G$ that define $G_0^i$. Those vertices of $T_G$ are called traced vertices, respectively traced atoms since each vertex corresponds to an atom in the interval PEM. Since the layout permutation is chosen uniformly at random, it implies that the vertices of level $l$ of the tree form a uniform matching among the subtrees of their children. The marked subtrees are identifiable, but the unmarked ones are indistinguishable, and the matching does not connect marked subtrees.

In the following, the probability $p'$ that there is an edge between two arbitrary vertices $u$ and $v$ of $G_0^i$ is considered ($u = v$ is allowed here). The vertices $u$ and $v$ represent a block or a cache $M_u$ and $M_v$ (that have maximal size $M$) of the interval PEM. There is an edge between $u$ and $v$ if there are atoms in $M_u$ and $M_v$ that are defined by the random proximate neighbors instance or the corresponding matching. Here, each traced atom is represented by a marble. At most half of the marbles are marked by unique numbers or identifiable, the others are unmarked. For the worst case scenario assume that half of them are marked. Drawing the edges of the matching (a uniform one because $\pi$ was uniform) can be done by considering the marbles in an arbitrary order and choosing the matching partner uniformly at random from the remaining possible neighbors. We choose as order to first take all marked marbles at $u$ (and choose a neighbor uniformly at random from the remaining unmarked marbles) and then the (remaining) unmarked marbles at $u$ (choose a random unmarked marble). In the end, the random variable $Y$ of interest is the number of times that this random choice leads to a connection to $M_v$. The conditional probability $p'$ of increasing $Y$ is at most $\frac{M}{\sqrt{2(|W_i| - 2M)}}$, since there may be only one out of $|W_i| - 2M$ marbles that are unmarked (thus $\frac{1}{2}(|W_i| - 2M)$) that creates an edge between $u$ and $v$, but this can happen $M$ times at $M_u$.

Hence, the expected value of $Y$ is asymptotically at most $\frac{M^2}{|W_i|} \leq M^{1/2}$. The random variable $Y$ is called big if $Y \geq M^{1/2} \sqrt{5 \ln M}$. The probability $p$ that $Y$
is big can be bounded from above by Hoeffdings considerations on Bernoulli experiments [Hoe63]:

**Lemma 3.37 (Hoeffding [Hoe63])**

When tossing \( n' \) times a coin with success probability \( p' \) then the probability to have more than \( T = (p' + \epsilon')n' \) successes is obtained is at most \( p = e^{-2\epsilon'^2 n'} \) for \( \epsilon' > 0 \).

Here, \( n' \) equals \( M \) and choosing \( \epsilon' = \sqrt{\frac{5\ln M}{2M}} \) yields a bound on the probability that \( Y \) is at most \( T = (p' + \epsilon')n' = O(M^{\frac{1}{2}}\sqrt{\ln M}) \). The bound of Hoeffding states that \( Y \) is big with probability at most \( p = e^{-2\epsilon'^2 n'} \). Calculating \( 2\epsilon'^2 n' = 2\left(\sqrt{\frac{5\ln M}{2M}}\right)^2 M = 2\frac{5\ln M}{2M} M = 5\ln M \), leads \( p \leq e^{-5\ln M} = \frac{1}{M^5} \). Interpreting this in the context of \( G_0^i \), the probability that any edge or self-loop has high multiplicity is at most \( p \).

A union bound is applied to bound the overall multiplicity of \( G_0^i \). The lower bound is only useful for algorithms that perform at most \( \log^2 M \) parallel I/Os. Thus, the number of vertices in \( G_0^i \) is at most \( N_B + P\log^2 N = 2M + M4\log^2 M \), and hence the number of edges (pairs) in \( G_0^i \) is at most \( 28M^2\log^4 M \). Multiplying this with \( p \) this yields that the probability that any edge in the graph has multiplicity more than \( M^{\frac{1}{2}}\sqrt{\ln M} \) is at most \( 28M^2\log^4 M e^{-M^{\frac{1}{4}}/2} \), which tends to zero faster than \( 1/M^2 \).

Due to the differing choice of \( s \) and \( k \), Lemma 3.28 is recap with updated values.

**Lemma 3.38**

Let there be an interval PEM with \( P = M = 2B \), and let \( N = 2^x \) for \( x \in \mathbb{N} \). Furthermore, let \( G \) be a GIF instance of size \( M^2 \geq N \geq M^{\frac{3}{2} + \epsilon} \) for a uniformly chosen layout permutation \( \pi \), and let \( T_G \) be its guide. Consider in this interval PEM an arbitrary deterministic GIF algorithm \( A \) that operates on \( G \). Assume that \( A \) is by the previously defined stages in stage \( i + 1 \) and that \( t_i < \log^2 N \). Let \( p(i,M) \) be the probability that stage \( i \) took \( \Omega(\log N) \) parallel I/Os. Then there is an \( x' \) such that for all \( x \geq x' \) and for each \( i \) it holds \( p(i,M) \geq 1 - \frac{1}{M^x} \).
Proof  At time \( t_{i-1} \), the multiplicity of \( G_{i-1} \) is by Lemma 3.36 at most \( M^{1/2 + \varepsilon_m} \) with probability \( 1 - \frac{1}{M^2} \). Therefore, by Lemma 3.27 the multiplicity of \( G_t \) is at most \( 4^t M^{1/2 + \varepsilon_m} \) with probability \( 1 - \frac{1}{M^2} \). By the assumption \( t_i < \log^2 M \), the number of vertices in \( G_t \) is \( P \log^2 M \). Hence, at time \( t_i + t \), the total multiplicity of loops in \( G_t \) is \( M^{1/2 + \varepsilon_m} \) with probability \( 1 - \frac{1}{M^2} \). By the assumption \( t_i < \log^2 M \), the number of vertices in \( G_t \) is \( P \log^2 M \). Hence, at time \( t_i + t \), the total multiplicity of loops in \( G_t \) is \( M^{1/2 + \varepsilon_m} \) with probability \( 1 - \frac{1}{M^2} \). By taking chance encounters into account, the total multiplicity of loops in \( G_t \) is bounded by \( M^{1/2 + \varepsilon_m} P(4^t + t) \log^2 M \). For the stage to finish, this number must be at least \( X > M^{3/2 + \varepsilon_X} \). Hence, \( M^{1/2 + \varepsilon_m} M(4^t + t) \log^2 M > M^{3/2 + \varepsilon_X} \) implies \( 4^t + t > \frac{M^{3/2 + \varepsilon_X - 1/2 - \varepsilon_m}}{\log^2 M} = M^{\varepsilon r} \log^2 M \) and hence \( t = \Omega(\log M) \). □

There are \( \Theta(\log N) \) stages, each taking at least \( \Omega(\log N) \) I/Os, yielding by a union bound over all \( G_t \) for all progress stages \( i < s \):

**Lemma 3.39**

Let there be an interval PEM with \( P = M = 2B \), and let \( N = 2^x \) for \( x \in \mathbb{N} \). Furthermore, let \( G \) be a GIF instance of size \( M^2 \geq N > M^{3/2} \) for a uniformly chosen layout permutation \( \pi \). Then there is a \( x' \) such that for all \( x \geq x' \), solving \( G \) in the interval PEM requires \( \Omega(\log^2 N) \) parallel I/Os with high probability \( (p > 1 - 1/M) \).

By Yao's principle [Yao77] this lemma implies a lower bound for Las Vegas randomized algorithms:

**Theorem 3.40**

Let there be an interval PEM with \( P = M = 2B \), and let \( N = 2^x \) for \( x \in \mathbb{N} \). Furthermore, let \( G \) be a GIF instance of size \( M^2 \geq N > M^{3/2} \). The expected number of parallel I/Os to solve \( G \) with a Las Vegas randomized algorithm on such an interval PEM is \( \Omega(\log^2 N) \).
CHAPTER 4

An algorithmic toolbox for the PEM model

In this chapter, PEM-efficient algorithms for several computational problems are presented. In Chapter 5 these algorithms are combined to an algorithm to compute a tree decomposition of width $k$ for a graph.

This chapter is divided into three parts which each present algorithms of a certain class. The most fundamental algorithms are introduced in Section 4.1.

In general, a successful theme for designing efficient algorithms for the PEM model is using alternating rounds of local operations and information distribution. This scheme can be found in the previously mentioned algorithms of Lemma 3.1 and Lemma 3.5. For example, the randomized fractional independent set algorithm (Lemma 3.5) works in three such rounds: first, by a local operation it flips a coin for each vertex, then it distributes information by sending each coin value to its predecessor, and finally a local operation determines the membership in the independent set. This scheme led to an efficient PRAM simulation technique in the EM model [Chi+95]. Section 4.2 generalizes this scheme to the PEM model. It is applicable to many simple algorithms for various tasks and some of them are shortly presented.

While this technique yields for several simple tasks very efficient algorithms in the PEM model, for more involved problems this is not the case. Some subroutines for computing a tree decomposition of width $k$ are considered in Section 4.3.

Many algorithms presented in this section rely fundamentally on list ranking. As discussed in Section 3.2, there are three algorithms for list ranking with slightly different properties, depending on the procedure that computes the fractional independent set. For the functionality of the algorithms of this section it does not matter which of these algorithms is used. Therefore, in
the following chapters, the exact running time of a list ranking algorithm is denoted by the placeholder listRankP.

**Definition 4.1**
Let \( \mathcal{I} \) be an algorithm to compute a fractional independent set on list graphs. The expression “listRankP \( \mathcal{I} \parallel 0 \) parallel I/Os” represents for a computational problem of size \( N \) one of the following terms and directly implies the corresponding properties of the PEM model in which an algorithm is analyzed:

1. If for \( \mathcal{I} \) the randomized independent set construction is used, then Lemma 3.5 implies that the expression listRankP \( \mathcal{I} \parallel 0 \) parallel I/Os is in expectation in \( \mathcal{O}(\text{sort}(N) + (\log P) \cdot \log \frac{B}{\log P}) \) and the PEM machine accesses random bits.

2. If for \( \mathcal{I} \) the deterministic coin tossing technique is used, then Lemma 3.1 implies that listRankP \( \mathcal{I} \parallel 0 \) parallel I/Os is in \( \mathcal{O}(\log^* N \cdot (\text{sort}(N) + (\log P) \cdot \log \frac{B}{\log P})) \) and the PEM machine is deterministic.

3. If for \( \mathcal{I} \) the delayed pointer processing technique is used, by Lemma 3.7 the term listRankP \( \mathcal{I} \parallel 0 \) parallel I/Os is in \( \mathcal{O}(\text{sort}(N) + B \cdot \log B \cdot \log t N \cdot \log \frac{M}{B^0}) \) and the PEM machine is deterministic and the cache size is bounded by \( M = B^0 \).

Most of the work presented in this chapter is joint work with Riko Jacob and Matthias Mnich. The central results are published [JLM14]. Therefore, parts of this chapter are taken identically from this publication.

### 4.1 Simple tasks for the PEM model

#### 4.1.1 Duplicate removal

**Definition 4.2**
The input of the duplicate removal problem is a multiset \( X \) of \( N \) elements that is taken from a total order. The task is to return the maximal set \( Y \subseteq X \) that is not a multiset in contiguous memory.
Lemma 4.3
If the input elements are taken from a total order, the duplicate removal problem can be solved in $O(sort_P(N))$ parallel I/Os in the CREW PEM model.

Proof The input set is sorted by its natural order. Thus, elements that are equal are located in contiguous memory. For each of the unique elements of $X$, the element at the smallest memory position can be labeled. This is done by scanning, where each processor is allowed to peek at the last element of its left element. By another sorting step, all labeled elements can be returned in contiguous memory. The number of parallel I/Os follows easily. □

4.1.2 Selecting elements with some property

Definition 4.4
The input to the filter problem is a multiset $X$ that is taken from a total order $U$ and a filter set $F \subseteq U$, and let $|X| + |F| = N$. Let $\equiv$ be a constant time computable equivalence relation. Then, for the filter problems it is true that for each $y \in F$, the set $I_y = \{x \in X | x \equiv y\}$ is an interval. The result of the filter problem is $\bigcup_{y \in F} I_y$. This means, all elements of $X$ which are equivalent, with respect to $\equiv$, to an element of $F$ are returned.

Lemma 4.5
The filter problem can be solved in $O(sort_P(N))$ parallel I/Os in the CREW PEM model.

Proof The problem is solved by the following procedure: By scanning a new set $F'$ is created that contains for each element $x$ of $F$ two elements $x_0$ and $x_1$. Furthermore, the element $x_0$ ($x_1$) is selected such that after sorting $X \cup F'$, it is a direct predecessor (successor) of elements which are equivalent to $x$.

Let $A$ be the disjoint union of $X$ and $F'$. By scanning, to every element of $X$ ($F'$) the value 0 (1) is assigned. Then $A$ is sorted by the total order and for $A$ all prefix sums modulo 2 are computed by Lemma 2.4.

Sorting the elements of $X$ by their prefix sum values, and returning the elements having a prefix sum value of 1 solves the filter problem. Since all steps are dominated by sorting, the lemma follows. □
4.1.3 Compute a spanning tree of a DAG

**Definition 4.6**
A 1-in-tree is a directed acyclic graph such that the in-degree of each vertex is at most 1.

**Lemma 4.7**
For a DAG $D^s$ of size $N$ with only one source $s$, a spanning 1-in-tree $T$ can be computed in $O(sort_P(N))$ parallel I/Os in the CREW PEM model.

**Proof**
By labeling for each vertex of $V(D^s) \setminus \{s\}$ exactly one incoming edge, $T$ is obtained. This can be implemented efficiently by sorting edges by their target vertices, labeling by scanning one incoming edge, and again sorting the edges of $D^s$ by the label for obtaining $T$ in a compact representation. Therefore, the procedure takes $O(sort_P(N))$ parallel I/Os.

Thus it is left to show that this selection of the edges yields a spanning 1-in-tree. The selected edges cannot contain a cycle, since such a cycle must be directed, but $D^s$ is acyclic by definition. Furthermore, the selected edges must be connected since there is only one source $s$ in $D^s$. Since for each vertex, except $s$, one edge is selected, there are $N - 1$ edges selected. Therefore, $T$ is a tree and $T$ is obviously spanning. Furthermore, every vertex has in-degree 1 by the selection of the edges. $\square$

4.2 Simulation of PRAM algorithms

In the following it is shown how PRAM algorithms which use at most $O(N)$ space can be simulated in the PEM model. This yields easily efficient algorithms in the PEM model for all problems that have $O(1)$ time algorithms in the PRAM model. The construction of the proof follows exactly the work of Chiang et al. and is stated here for completeness [Chi+95]. The result follows essentially from using the PEM-efficient algorithms for sorting and scanning, instead of the algorithms for the EM model.
### Lemma 4.8

Let $A$ be a CREW PRAM algorithm that uses $N$ processors and $O(N)$ space. A single step of $A$ can be simulated in $O(sort_P(N))$ parallel I/Os in the CREW PEM model.

**Proof**  Algorithm $A$ is assumed to be free of indirect memory accesses since each indirect memory access of $A$ can be removed by replacing an indirect memory access by $O(1)$ steps. The memory used by the algorithm $A$ is represented by an $O(N)$-sized array $C$ in the shared memory of the PEM machine.

A step of a processor uses only a constant number of operands. Each operand is represented by its memory location, the processor where it is used and the operation that uses the operand. Knowing these operands as a (multi-)set $L$ of size $O(N)$, for each processor its operands can be assigned using a copying problem (Definition 2.5). First, $L$ is sorted by the memory locations of the operands. Then, the copying problem is solved as described in Lemma 2.6 in $O(sort_P(N))$ parallel I/Os. Sorting $L$ by the processor and the operation where an operand is used, yields in $O(sort_P(N))$ parallel I/Os for each processor its list sorted by the order of operation that use the operand.

By scanning $L$, all operations of the considered step of $A$ can be executed. The outputs for all processors are stored in a set $O$. Each of the $O(1)$ outputs of an operation of a processor consists of a value which the operation computed and the memory location where it is stored to. These outputs have to be reintegrated into the memory array $C$ of the simulated PRAM machine. This can be achieved by sorting the union of $C$ and $O$ by the memory address, and scanning the result, which replaces old values. Note that there are now write conflicts in $O$ since $A$ operates on a CREW PRAM machine. Therefore, one step can be simulated in $O(sort_P(N))$ parallel I/Os.  

The following theorem is similar to [Chi+95, Theorem 3.1]. It states that PRAM algorithms can be simulated in the PEM model since each step of a PRAM algorithm can be simulated by Lemma 4.8.

### Theorem 4.9

Let $A$ be a PRAM algorithm which uses $N$ processors, $t$ parallel time steps, and $O(N)$ space. Then $A$ can be simulated in $O(t \cdot sort_P(N))$ parallel I/Os.
In the following, two computational problems are considered for which by simulating PRAM algorithms an efficient algorithm for the PEM model is obtained. Those PRAM algorithms use an adjacency list graph representation as assumed by Bodlaender and Hagerup [BH98]. A graph which has \( N \) vertices is represented by an adjacency list as follows: each vertex has a unique identifier which is at most \( O(N) \) and a pointer to its doubly linked adjacency list. An adjacency list for a vertex \( v \) has for each of its neighbors \( w \) an entry which contains the name of \( w \) and a cross-pointer which refers to the edge entry of \( v \) in the adjacency list of \( w \).

Note that the adjacency list graph representation can be converted into an edge list representation easily by scanning by an \( O(1) \) time PRAM algorithm: consider the adjacency list of a vertex \( v \). For each entry \( w \), simply the edge \( (v, w) \) is written to the output edge list. Also the vertex list can be created by scanning the vertex set of the adjacency list representation.

Furthermore, observe that an adjacency list representation can be computed in \( O(sort_p(N)) \) parallel I/Os from an edge list representation: the adjacency lists are obtained by sorting all edges by their source vertex. These are not yet the final adjacency lists, but the entries are modified by adding pointers for the doubly linked list and the cross-pointers.

By a prefix sums computation (Lemma 2.4), the size of the neighborhood for each vertex can be computed efficiently. Thus, the adjacency lists \( A \) can be transformed into doubly linked lists, by scanning the adjacency lists and adding for each vertex a pointer to its predecessor and its successor.

By creating a copy of \( A \) in which each entry is labeled with its memory address in \( A \), the cross-pointers can be created as follows: sort the copy by their target edges, and break ties by the natural order of their source vertices. Then by scanning \( A \) and its copy in parallel, the source vertex of the \( i \)-th entry in \( A \) can be replaced by the memory address of the \( i \)-the entry in the sorted copy.

The vertex identifiers can be created by duplicate removal easily.

In the following, short applications of the PRAM simulation are presented. Note that Theorem 4.9 yields an easy proof for the complexity of the randomized independent set construction on list graphs (Lemma 3.5) and for the deterministic coin tossing technique (Lemma 3.1).
4.2 Simulation of PRAM algorithms

4.2.1 Computing acquainted twins

**Definition 4.10 (Bodlaender and Hagerup [BH98])**

Two vertices $v$ and $w$ of a graph $G$ are said to be *twins* if their neighborhood is the same ($N(v) = N(w)$).

Let $d$ be a constant. Two non-adjacent vertices $v$ and $w$ are *$d$-acquainted* if they have a common neighbor $x$ such that $v$ and $w$ are listed in the adjacency list of $x$ at most $d$ positions apart.

**Lemma 4.11**

Let $d$ and $s$ be a constants. Given a graph $G$ of size $N$, all $d$-acquainted twins, such that both $d$-acquainted vertices have maximal degree $s$, can be computed in $O(sort_p(N))$ parallel I/Os In the CREW PEM model.

**Proof** For the PRAM model there exists an $O(1)$-time algorithm for this task [BH98]. Thus the lemma follows directly from Theorem 4.9. For completeness, the algorithm for the PRAM model is given here.

This task is easily parallelizable for every vertex $v$ and thus all vertices can be handled simultaneously. The processor assigned for vertex $v$ checks if the degree of $v$ is at most $s$ by scanning its adjacency list. If so, for every neighbor $u$ of $v$, by the cross-pointer of the $u$, each entry $w$ in the adjacency list of $u$ which is at most $d$ positions apart from the entry of $v$ in the adjacency list of $u$, is a candidate for a $d$-acquainted vertex. If the degree of $w$ is at most $s$, then $v$ and $w$ are $d$-acquainted vertices which have maximal degree $s$. Since this can be computed with three nested loops that each scan $O(s)$ vertices, the $d$-acquainted vertices that have maximal degree $s$ can be computed in $O(s^3) = O(1)$ time steps in the PRAM model.

Furthermore, checking if two candidates are twins, meaning their neighborhood is the same, can be done easily by two nested loops by at most $O(s^2)$ steps in the PRAM model. Therefore at most $O(poly(s)) = O(1)$ time steps are needed to compute all $d$-acquainted twins with small degree of a graph.  

$$\square$$
4.2.2 Computing an FIS for a bounded degree graph

Lemma 4.12
Let $I$ be an algorithm to compute a fractional independent set for a list graph. For a graph $G$ of size $N$ that has maximal degree $h$, a fractional independent set can be computed in $O(\text{listRank}_p^I(N))$ parallel I/Os in the CREW PEM model.

Algorithm 4.1: Compute an FIS for a graph of bounded degree.

**Input:** A graph $G$ of bounded degree $h$ and a partition of its edges into $i < 2h$ list graphs $E_j$

**Output:** A fractional independent set $S$ for $G$

1. $S \leftarrow V(G)$
2. for $1 \leq j < i$
3. \hspace{1em} $I_j \leftarrow I(E_j \cap S)$ \hspace{1em} // $I$ computes an FIS on list graphs
4. \hspace{1em} $S \leftarrow S \setminus (V(E_j) \setminus I_j)$
5. return $I$

**Proof** Dadoun and Kirkpatrick presented an algorithm which takes $O(1)$ time steps in the PRAM model to partition the edge set of $G$ into a set of at most $2h$ list graphs. Then, Algorithm 4.1 can be applied to the list graphs and yields an FIS for $G$. Their algorithm can be implemented by Theorem 4.9 efficiently in the PEM model. In the following, the PEM-efficient implementation of Algorithm 4.1 is shown and it actually computes an FIS for $G$.

Let $E(G)$ be partitioned into list graphs $E_j$ for $1 \leq j \leq i \leq 2h$. The set $S$ is a set of vertices which are candidates to be in the resulting fractional independent set. Initially, this set $S$ is $V(G)$, but over time vertices are removed and thus they are called survivors. In the $i$-th round there are the survivors $S$ left. Computing an FIS on $E_j \cap S$ (Step 3) yields an independent set $I_j$ for $E_j$ which remains in $E_j \cap S$. The set $R = V(E_j) \setminus I_j$ conflicts with the choice $I_j$. Thus, it cannot be contained in $S$ if $S$ is an independent set for $\bigcup_{1 \leq k \leq j} E_k$. Therefore, in Step 4 those vertices $R$ are removed from $S$.

In each application of $I$ on a list graph, the fraction of vertices of $E_j$ considered for the FIS is at least $c$, for some constant $0 < c \leq \frac{1}{2}$. Thus, after $2h$ rounds, there are at least $c^{2h} \cdot |V(G)|$ vertices in the returned set $S$. Note that the set
4.3 Other fundamental algorithms for the PEM model

4.3.1 The Euler tour technique

The Euler tour technique was introduced by Tarjan and Vishkin [TV85]. It is used in several algorithms in the PRAM model [TV85; MSV86; DS04], the EM model [Chi+95; IU03], and the PEM model [AGS10].

An Euler tour for an undirected tree $T$ is a linked list that starts and ends at a root vertex $r$ of $T$. The list contains for each edge $e$ of $T$ two edges which traverse $e$ with respect to $r$ in each direction once. It is visualized in Figure 4.1.

Arge, Goodrich, and Sitchinava state the complexity for computing an Euler tour for a tree in the PEM model only for up to $P \leq \frac{N}{B^2}$ processors [AGS10]. Since their algorithm uses only sorting and scanning, their bound can be improved easily to the full parameter range of $P, M,$ and $B$ for the PEM model as stated in the following lemma.

**Lemma 4.13 (Arge, Goodrich, and Sitchinava [AGS10, Theorem 4.1])**

An Euler tour for a tree $T$ can be computed in $O(\text{sort}_P(N))$ parallel I/Os in the CREW PEM model.

![Figure 4.1: A tree (a) and its Euler tour (b).](image-url)
By splitting the Euler tour at the root of the tree, a list can be obtained. Furthermore, by list ranking and prefix sums computation several non-trivial problems can be solved. For example a tree can be rooted by the following steps. Compute an Euler tour for the tree, rank the list and sort the edges accordingly. Then for each vertex \( v \) of \( T \), the maximal rank value from all vertices that correspond to \( v \) can be computed in \( O(\text{scan}_P(N) + \log P) \) parallel I/Os. By scanning the edges of the Euler tour, the edges whose target’s values in \( T \) are smaller than their source vertex’ values can be labeled. By a final sorting step, those labeled edges can be written to a compact output tree where each edge is directed towards the root.

### 4.3.2 Making a tree decomposition binary

**Lemma 4.14**

A tree decomposition with a tree \( T' \) with \( N \) nodes of arbitrary degree can be transformed into a rooted binary tree decomposition with \( O(N) \) nodes in \( O(\text{listRank}_P(N)) \) parallel I/Os in the CREW PEM model.

**Proof**  From a computational point of view this task is trivial: each node with \( b \) children is replaced by a path with \( b \) nodes, where each of the \( b \) nodes represents the same bag. Thus, it is important to provide an algorithm which works efficiently in the PEM model.

By the Euler tour technique, Lemma 4.13 yields that the tree \( T' \) can be rooted in \( O(\text{listRank}_P(N)) \) parallel I/Os. By sorting the edges of \( T' \) by their target node, with a prefix sums computation for each node \( v \) the degree of each node \( v \) can be determined. For a vertex \( v \) in \( T' \) that has degree \( \text{deg}(v) \), in the output tree decomposition on \( T \) there are vertices \( v_i \), with \( 1 \leq i \leq \text{deg}(v) \), whose bags contents are the contents of the bag of \( v \).

The following formal description of the output is visualized in a small example in Figure 4.2. Let \( e = \{u, v\} \) be the \( i \)-th children edge of \( v \) of \( T' \). Then the following edges are created in \( T \): \((u_1, v_i)\) and \((v_{i+1}, v_i)\) if the vertex \( v_{i+1} \) exists. Generating the output takes only \( O(\text{scan}_P(N)) \) parallel I/Os, since for each edge \( e \) in \( T' \) there is only a constant number of operations needed because the bags are of constant size.
In total, sorting and computing the degree takes \( O(sort_p(N)) \) parallel I/Os, which is the dominating term for the parallel I/Os of the algorithm. \( \square \)

### 4.3.3 Computing a contraction tree for a tree

The contraction tree technique enabled several efficient algorithms for the PRAM model \([KR90; NNS89]\). In the following section, it is shown that this is also true for the PEM model. But first it is shown that a contraction tree can be computed efficiently in the PEM model.

**Definition 4.15 (Bodlaender and Hagerup \([BH98]\))**

A tree-contraction algorithm applied to a binary tree \( T \) defines a tree sequence \( T = T_0 = (V_0,E_0), T_1 = (V_1,E_1), \ldots, T_r = (V_r,E_r) \), where each tree \( T_{i+1} \) is obtained from its predecessor \( T_i \) by contracting a set of edges \( F_i \subseteq E_i \). The set \( F_i \) has two properties.

1. The set \( F_i \) is a matching in \( E_i \).
2. Each edge in \( F_i \) has at least one endpoint with maximal degree two.

This tree sequence defines a contraction tree \( T_b \) of \( T \) as follows. The node set of \( T_b \) is defined as \( X := \bigcup_{i=0}^r V_i \). The nodes of \( T_0 \) are the leaves of \( T_b \) and called base nodes. For each edge \( e = \{u,v\} \) which is contracted to a node \( x \) in the tree sequence from tree \( T_{i-1} \) to \( T_i \), the edges \((u,x)\) and \((v,x)\) are added to the edge set of \( T_b \). For simplicity it is assumed, that a vertex, which is not incident to a contracted edge in a step from \( T_{i-1} \) to \( T_i \), has the same identifier in \( V_{i-1} \) and \( V_i \), and thus appears only once in \( X \).

The next lemma states that \( T_b \) can be computed efficiently.

**Lemma 4.16**

Let \( I \) be an algorithm for computing a fractional independent set on a set of list graphs. Computing a contraction tree \( T_b \) for a binary tree \( T' \) of size \( N \) takes \( O(listRank_p^T(N)) \) parallel I/Os in the CREW PEM model.

**Proof** First it is shown how the tree sequence can be computed. To compute all trees of the tree sequence, an algorithm is presented that works in rounds as
the algorithm for list ranking. In each round $i$ the following computations are done: from the tree $T_i$ a set $L_i$ of list graphs is computed. Then, by $I$, at least a constant fraction of the edges of $L_i$, the set $F_i$, is extracted. Finally $F_i$ is then contracted.

Therefore it is essential to compute $L_i$ for a tree $T_i$. This is described in the following two paragraphs. In a first step all edges of $T_i$ that are incident to a vertex of degree at most two are selected. Note that these edges are satisfying the second property of $F_i$ and yield a forest $F$ of induced sub-trees of $T_i$.

However, $F$ may still contain vertices of degree 3, but for applying $I$, list graphs are needed. By removing for each vertex $v$ of degree 3 in $V(F)$ an arbitrary edge to a child of $v$, a set $L_i$ of lists is obtained. Note that each of these steps is computable in $O(\text{sort}_P(N))$ parallel I/Os in the PEM model.

In this paragraph it is proven that set $L_i$ contains at least as many edges as a maximal solution $M^*$ for $F_i$: each edge in $M^*$ has a vertex that has maximal degree two. Thus, if such an edge is not contained in $L_i$, then it has been removed in favor of its sibling edge, which conversely cannot be included in $M^*$, since $M^*$ is a matching. Furthermore, Miller and Reif show as a byproduct of their Theorem 1 that the size of the matching $M^*$ is at least a constant fraction of the size of $T_i$ [MR85]. Thus, computing a fractional independent set of $L_i$, yields a fractional independent set of $T_i$.

Therefore, the size of the trees $T_i$ is geometrically decreasing. Computing $F_i$ and bridging out (contracting) edges dominates the number of parallel I/Os.

Figure 4.2: Making an arbitrary tree (left) a binary tree (right).
Furthermore, there is an optimal algorithm for computing a contraction tree sequence in the PRAM model, which can be simulated in the PEM model for small tree sizes [Abr+89]. Therefore the analysis of Section 3.2 can be applied, yielding that the tree sequence can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os in the PEM model.

The contraction tree $T_b$ is defined implicitly by the tree sequence of $T'$ and can be created explicitly by writing the edges of $T_b$ to the memory after each round of contracting edges. This takes at most another $\text{scan}_P(|T_i|)$ parallel I/Os per round, and thus is covered by $O(\text{listRank}_P^T(N))$ parallel I/Os. In this process, to each vertex of the contraction tree, the current round, the current parallel I/O (time stamp), and the processor identifier can be added without changing the asymptotic complexity since this is constant-sized information.

**Definition 4.17**

A contraction tree is called *annotated* if each vertex (in an edge) is annotated by the processor $p$, the round $i$, and the time, respectively the parallel I/O $t$, after which it has been created, respectively the corresponding edge has been contracted.

By Lemma 4.16 a contraction tree can be computed in $O(\text{listRank}_P^T(|T'|))$ parallel I/Os. Therefore an annotated contraction tree can be computed in the same number of parallel I/Os:

**Corollary 4.18**

Let $I$ be an algorithm for computing a fractional independent set on a set of list graphs. An annotated contraction tree $T_b$ for a binary tree $T'$ of size $N$ can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os in the CREW PEM model.
4.3.4 Processing an annotated contraction tree

Lemma 4.19
Let $T$ be an annotated contraction tree of size $O(N)$ that has been computed in $t \in \Omega(sort_P(N))$ parallel I/Os. Then a bottom-up (top-down) computation on the vertices of $T$, such that each computation on a vertex of $T$ depends only on its two children (one parent) and takes only a constant number of I/Os, can be performed in $O(t)$ parallel I/Os in the CREW PEM model.

Proof
In this proof, only the bottom-up approach is considered, since the top-down approach is just a reversed bottom-up computation.

Processing $T$ is done in the exact same rounds as the computation of $T$. To this end, sets of intermediate results $R_i \subseteq E_T$ are managed. Each intermediate result is labeled with an edge of $T$, which due to its parent vertex is labeled by a round value $i$, in which the edge has been created, a time stamp $t$, which determines the I/O operation, when it has been created in the round, and a processor $p$ which created the vertex. Those information are known for each vertex of $T$ since $T$ is annotated. Note that during the creation of the contraction tree, in each round all vertices, respectively edges of a round can be sorted without violating the asymptotic bound on the running time of the algorithm that computed $T$.

Initially, $R_1$ consists of the edges from the base nodes of $T$ to their parents with assigned values for the computation of the base nodes. Before round $i$ starts, the set $R_i$ is sorted, such that the intermediate results are ordered by round as a primary key, the processor as a secondary key, and the time stamp as a tertiary key, and in case of ties by the natural order of the target vertices. In round $i$, each processor $p$ scans all intermediate results which are labeled by $i$ and $p$. Since those intermediate results are ordered by time stamp and vertex order, the operation for two intermediate results whose edges have a vertex $v$ of $T$ as a target can be calculated immediately. The results of this operations are written block-wise to a set $I_V$ of intermediate results, each labeled by the corresponding vertex $v$ of $T$. After round $i$ for the vertices $I_V$, the edges $I_E$ that have vertices of $I_V$ as sources are selected by a filter problem. The set $R_{i+1}$ is defined as $R_i \cup I_E$. Note that all these tasks have complexity of $O(sort_P(N))$ parallel I/Os. □
Note that by Corollary 4.18 an annotated contraction tree can be computed in \(O(\text{listRank}_P^T(N))\) parallel I/Os. Thus, a bottom-up or top-down computation as described in the previous lemma can be evaluated in \(O(\text{listRank}_P^T(N))\) parallel I/Os as well.

This last observation leads to the definition of an efficiently evaluable tree. It means that any bottom-up or top-down computation that consumes only \(O(1)\) time and space can be performed on a contraction tree in \(O(\text{listRank}_P^T(N))\) parallel I/Os in the CREW PEM model. This is in certain parameter ranges faster than the naïve evaluation, in which each layer is processed in a single parallel I/O which would lead to at least \(\Omega(\log N)\) parallel I/Os. Furthermore, this definition is extended to tree decompositions.

**Definition 4.20**
An annotated contraction tree \(T\) of size \(N\) is called an *efficiently evaluable* tree if it has been computed in \(O(\text{listRank}_P^T(N))\) parallel I/Os in the CREW PEM model.

A tree decomposition \((T, B)\) that has an efficiently evaluable tree \(T\) is called an *efficiently evaluable tree decomposition*.

**4.3.5 Converting an implicit tree decomposition into an explicit tree decomposition**

In this section a procedure of the sequential algorithm of Maheshwari and Zeh is considered that constructs a tree decomposition for a graph \(G\) explicitly [MZ09, pp. 455–456 (Phase 4)]. Despite the fact that this procedure is central to the tree decomposition algorithm presented in Chapter 5, it can be considered isolated and is therefore presented here, such that it can be referred to as a procedure in Chapter 5. The input to their procedure is an implicit representation of a tree decomposition which is defined as follows.

**Definition 4.21**
An *implicit representation* of a tree decomposition for a graph \(G\) consists of several implicitly represented path decompositions for subgraph \(G_i\) and a link list \(\mathcal{L}\). The link list \(\mathcal{L}\) indicates which bags of path decompositions have to be connected with an edge to form the final explicit tree decomposition.
An implicit representation of a path decomposition is defined on a directed acyclic graph $D_{st}$ which has only one source $s$ and only one sink $t$. Each vertex $\alpha$ of $D_{st}$ is annotated by sets $L_{\alpha} \subseteq V(G)$, $R_{\alpha} \subseteq V(G)$ and a stretch value $\rho_{\alpha} \in \mathbb{N}$ [MZ09, p. 446 (G2)]. Furthermore the following properties are fulfilled:

1. For every vertex $\alpha_l$ on a directed $st$-path ($s=\alpha_0, \alpha_1, ..., \alpha_l, ..., \alpha_{k-1}, \alpha_k=t$) in $D_{st}$ the value $\sum_{j=0}^{l} \rho_{\alpha_j}$ is the same, independently of the chosen $st$-path [MZ09, p. 446 (G3)].

2. For each $x \in V(G)$ there are two vertices $\mu(x)$ and $\nu(x)$ in $D_{st}$ such that $x \in L_{\mu(x)}$ and $x \in R_{\nu(x)}$ [MZ09, p. 446 (G4)].

The intuition of $L_{\alpha}$, $R_{\alpha}$, and $\rho_{\alpha}$ is the following: a vertex $\alpha$ represents a part of a path decomposition, which covers the vertices of $L_{\alpha}$, and all vertices of $G$ which are on a path from $s$ to $\alpha$ and have not been removed by a set $R_{\beta}$ where $\beta$ is a vertex on a $sa$-path. The value $\rho_{\alpha}$ represents how often the contents of bag(-parts) represented by $\alpha$ have to be repeated at least.

A tree or path decomposition is said to be explicitly represented in a PEM if it is given as follows:

**Definition 4.22**

An explicit representation of a tree decomposition $T = (T, B)$ for a graph $G$ consists of a rooted tree $T$ in the edge representation. The bags of $T$ are represented by pairs of $V(T) \times G$: if $v \in G$ is in bag $b \in V(T)$, then there is an entry $(v, b)$.

An explicit representation of a path decomposition uses the same representation, except the fact that $T$ is a path.

Algorithm 4.2 gives an outline of the procedure of Maheshwari and Zeh to convert an implicit representation of a path decomposition into an explicit representation efficiently in the EM model. Note that in their description [MZ09, p. 455] the first step is to compute from a “flippable DAG” $G(e)$ a DAG $G'(e)$. In contrast, here the input to the procedure includes already this DAG, called here $D_{st}$. This is why the first step of their procedure is omitted in this description. Furthermore, note that their final step for linking path decompositions by the links of $L$ to a tree decomposition is mentioned neither in Algorithm 4.2 nor in Algorithm 4.3, but explained in the proof of Lemma 4.23.
Despite the fact that in the following the procedures refer to only one (implicitly represented) path decomposition, all methods can be executed simultaneously for all path decompositions of an implicit tree decomposition. Each of the path decompositions is identified by an integer $i$ to distinguish for which subgraph $G_i$ of $G$ they are designed.

Algorithm 4.2: Serial algorithm for converting an implicit path decomposition to explicit representation.

**Input**: An implicit path decomposition for a graph $G_i$ which is a DAG $D^{st}_i$ of which each vertex is annotated by $R_\alpha, L_\alpha,$ and $\rho_\alpha.$

**Output**: A path decomposition $P = (I, B)$ for $G_i.$

1. Compute for all $\alpha \in V(D^{st}_i)$: $I(\alpha) = [b + 1, b + \rho_\alpha],$ where $b$ is defined by an in-neighbor $\alpha'$ of $\alpha$ with $I(\alpha') = [a, b].$

   /* Compute an interval $I(x)$ in $P$ for every $x \in G$ */

2. For all $\alpha \in V(D^{st}_i),$ and all $x \in L_\alpha \cup R_\alpha$: create an entry $(i, x, I(\alpha))$ in a set $Q.$

3. Sort $Q$ by $(i, x).$

4. Compute $I(x)$: if there is only one $(i, x)$ triple in $Q$ then $I(x) = I(\alpha),$ otherwise there are two such triples and $I(x)$ is the smallest interval containing $I(\mu(x))$ and $I(\nu(x))$ [MZ09, p. 446 (G5)].

   /* Compute from intervals a path decomposition */

5. For all $x \in V(G_i)$ and for each $y \in I(x),$ create $(i, y, x)$ by scanning.

6. Sort this by $(i, y)$ yielding the bags of $P.$

7. Add edges $((i, y - 1), (i, y))$ for every $y$ to define the path decomposition.

Algorithm 4.3 shows pseudo code for a PEM-efficient implementation of Algorithm 4.2. The only difference to Algorithm 4.2 is that the time-forward processing, that is needed for Step 1, is replaced.

**Lemma 4.23**

Let $I$ be an algorithm for computing a fractional independent set on a set of list graphs. An explicit representation of a tree decomposition $T$ can be computed from an implicit representation in $O(\text{listRank}_P^T(N))$ parallel I/Os in the CREW PEM model.

**Proof**  The algorithm consists of Algorithm 4.3 and some procedure that translates the link list $\mathcal{L}$ into edges of the tree $T.$ First, it is argued that Algorithm 4.3
Algorithm 4.3: Convert an implicit path decomposition to an explicit representation in the PEM model.

**Input:** An implicit path decomposition for a graph $G_i$ which is a DAG $D_i$ of which each vertex is annotated by $R_\alpha, L_\alpha$, and $\rho_\alpha$.

**Output:** A path decomposition $\mathcal{P} = (I, \mathcal{B})$ for $G_i$.

/* Step 1 of Algorithm 4.2 */
1. Compute a spanning 1-in-tree $T$ of $D_i$.
2. Compute an Euler tour for $T$.
3. Compute for each $\alpha \in D_i$ the interval $I(\alpha) = [a,b]$ by prefix sums.
   /* Remainder is a copy of Step 2 - Step 7 of Algorithm 4.2 */
   /* Compute an interval $I(x)$ in $\mathcal{P}$ for every $x \in G$ */
4. For all $\alpha \in V(D_i)$, and all $x \in L_\alpha \cup R_\alpha$: create an entry $(i, x, I(\alpha))$ in a set $Q$.
5. Sort $Q$ by $(i, x)$.
6. Compute $I(x)$: if there is only one $(i, x)$ triple in $Q$ then $I(x) = I(\alpha)$, otherwise there are two such triples and $I(x)$ is the smallest interval containing $I(\mu(x))$ and $I(\nu(x))$ [MZ09, p. 446 (G5)].
   /* Compute from intervals a path decomposition */
7. For all $x \in V(G_i)$ and for each $y \in I(x)$, create $(i, y, x)$ by balanced scanning.
8. Sort this by $(i, y)$ yielding the bags of $\mathcal{P}$.
9. Add edges $((i, y - 1), (i, y))$ for every $y$ to define the path decomposition.
can be implemented efficiently in the PEM model. Step 4 - Step 9 of Algorithm 4.3 are an identical copy of Step 2 - Step 7 of Algorithm 4.2. Since they involve only scanning and sorting, for these steps the corresponding PEM methods must be used, and some additional load balancing must be applied in Step 7, which can be resolved as described in Section 2.4.4.

Therefore, the next paragraph shows how the first step of Algorithm 4.2 is replaced and that after Step 3 of Algorithm 4.3 for each vertex $\alpha$ of $D^st_i$ the same values $I(\alpha) = [a, b]$ are known.

Replacing time-forward processing is possible since [MZ09, p. 446 (G3)] states, that for every vertex $\alpha_l$ on an $st$-path $(s = \alpha_0, \alpha_1, ..., \alpha_l, ..., \alpha_{k-1}, \alpha_k = t)$ in $D^st_i$ the interval values $I(\alpha_l) = [a, b]$ are independent of the directed path leading to $\alpha_l$ (with $a = 1 + \sum^{l-1}_{j=0} \rho_{\alpha_j}$ and $b = \sum^{l}_{j=0} \rho_{\alpha_j}$). The first two steps of Algorithm 4.3, computing a spanning tree $T$ of $D^st_i$ and computing an Euler tour for $T$, can be computed efficiently by Lemma 4.7 and Lemma 4.13. Thus, by the following procedure the $(a)$-values for Step 3 can be computed I/O-efficiently: note that an outgoing edge $c_\alpha$ of a vertex $\alpha \in D^st_i$ corresponds in the Euler tour of $T$ (Step 2) to an outgoing edge $\vec{c}_\alpha$ and an incoming edge $\vec{c}_\alpha$. By assigning to every $\vec{c}_\alpha$ the value $\rho_\alpha$ and to every $\vec{c}_\alpha$ the value $-\rho_\alpha$ and the outgoing edges of $s$ the value 1, respectively $-1$ for the incoming edges of $s$, a prefix sums problem has been defined. A similar construction leads the $b$ values for each $I(\alpha)$.

Since after Step 3 of Algorithm 4.3, both algorithms hold the same information, there is no need for arguing about the correctness of the algorithm, as it is proven by Maheshwari and Zeh [MZ09]. Thus, Algorithm 4.3 computes in $O(\text{listRank}_{p}(N))$ parallel I/Os simultaneously for each implicit path decomposition an explicit path decomposition.

As already described by Maheshwari and Zeh, the following modifications can be applied to compute an explicit representation of a tree decomposition $T$ [MZ09].

For linking the path decompositions to a tree decomposition, in the initial computation of the intervals $I(\alpha) = [a, b]$, for all vertices $\alpha$ there is added an entry $(\alpha, i, a)$ to a link translation table $LT$. By sorting the link translation table $LT$ and the link list $L$ by their first entries, the entries $(\alpha, \beta)$ of the link list can be translated into $((i_1, x), \beta)$. By sorting the link list $L$ by its second entries,
the entries \(((i_1, x), \beta)\) can be translated to \(((i_1, x), (i_2, y))\) and added to the final tree decomposition.

The bound on the number of parallel I/Os follows easily since all steps except the third one have sorting or scanning complexity. Since the complexity of the third step is due to a prefix sums computation over a linked list \(O(\text{listRank}_P^I(N))\), and this term dominates sorting and scanning, the overall complexity is \(O(\text{listRank}_P^I(N))\). \(\square\)
CHAPTER 5

Computing tree decompositions in the PEM model

The treewidth of a graph is an important property that measures the similarity of a graph to a tree. Several notoriously hard problems become efficiently solvable if they are considered on graphs of bounded treewidth. One of the first results that exploits that a graph has a tree decomposition of bounded width $k$ shows that the \textsc{NP}-hard problems \textsc{Independent Set} and \textsc{Hamiltonian Circuit} are solvable in polynomial time on such graphs [AP89; Arn85]. Subsequently, those results were generalized to a broader class of \textsc{NP}-hard problems [Bod88] and \textsc{PSPACE}-hard problems [Bod93b]. Also, checking if a monadic second order logic formula is satisfiable by a graph structure $G$ can be done efficiently if for $G$ a tree decomposition of bounded width is known [Cou90].

However, deciding if a graph $G$ has treewidth $k$ is \textsc{NP}-hard on its own. On the other hand, for several computational models there are asymptotically optimal (fixed parameter tractable) algorithms that compute for $G$ a tree decomposition of fixed width $k$ or decide that $\text{tw}(G) > k$. All important algorithms use the general scheme of the first (non-constructive) algorithm to decide treewidth for a fixed $k$ [RS95]. They compute essentially an approximate tree decomposition which is a basis for solving the problem exactly.

The non-constructive algorithm of Robertson and Seymour was improved in several respects. In 1993, Bodlaender and Kloks introduced the first linear time algorithm for the RAM model to compute for a graph $G$ a tree decomposition of a fixed width $k$ [BK96; Bod96; Bod93a]. Later, Bodlaender and Hagerup introduced a work-optimal algorithm for the \textsc{PRAM} model for up to $\frac{N}{\log^2 N}$ processors [BH98]. Maheshwari and Zeh introduced an optimal algorithm for the \textsc{EM} model that has sorting complexity [MZ09].
More precisely, all mentioned constructive algorithms use the following recursive approach: by recursive application of reduction rules, the graph $G$ is reduced to a graph $G_r$ of constant size. Thus, for $G_r$, a tree decomposition of width $k$ can be computed easily by brute force. By undoing a round of the application of the reduction rules, a tree decomposition is obtained which is only slightly larger than $k$. The width of this tree decomposition can be reduced to $k$ again, by using the dynamic programming approach of Bodlaender and Kloks [BK96]. Using the dynamic programming approach of Bodlaender and Kloks, the width of this tree decomposition can be reduced to $k$ again [BK96].

In the following, for every fixed $k$, a PEM-efficient algorithm is presented that computes for a graph $G$ a tree decomposition of width at most $k$ or decides that $\text{tw}(G) > k$. To this end, this chapter is structured as follows: the first section provides an overview on the algorithm and its three central methods. Section 5.2 and Section 5.3 describe these procedures in detail. The analysis of the running time for the complete algorithm is provided in Section 5.4, which is complemented by a fitting lower bound in Section 5.5.

Most of the work presented in this chapter is joint work with Riko Jacob and Matthias Mnich. The central results are published [JLM14]. Therefore, parts of this chapter are taken identically from this publication.

### 5.1 Overview on the algorithm

The PEM-efficient algorithm to compute for a graph $G$ a tree decomposition of width $k$ that is presented in this chapter also uses the classical recursive scheme. Due to the parallel and the external memory character of the model, the algorithm is based on two algorithms: the algorithm of Bodlaender and Hagerup, and the algorithm of Maheshwari and Zeh. The PRAM algorithm of Bodlaender and Hagerup introduced a framework for load balancing, which is also used for this algorithm in the PEM model. The I/O-efficient algorithm for the EM model by Maheshwari and Zeh can be implemented in the PEM model for actually computing a tree decomposition.

In the following, an overview on the three central methods, `reduce`, `balance`, and `treeDecompositionOfAtMost`, and their most important properties is presented. Their description, together with the pseudo code presented in Algo-
Algorithm 5.1, shows the overall structure of the algorithm. Note that its correctness, meaning that the algorithm actually computes a tree decomposition of width \( k \), follows directly from the original works [BH98; MZ09]. Therefore the focus of this chapter is to show the efficient implementation of the algorithm in the PEM model instead of proving the correctness of the algorithm.

**Algorithm 5.1**: Computing a tree decomposition of width \( k \) in a parallel model.

- **Input**: A graph \( G \) and a natural number \( k \) that parametrizes the algorithm.
- **Output**: A tree decomposition \( D \) of width \( k \) for \( G \), if \( \text{tw}(G) < k \).

\[
\begin{align*}
G_0 & \leftarrow G \\
r & \leftarrow c \cdot \log n \quad // \quad c > 2 \text{ depends on an independent set algorithm } I \\
\text{for } i = 1 \text{ up to } r \text{ do} & \\
G_i & \leftarrow \text{reduce}(G_{i-1}) \\
D_r & \leftarrow \text{treeDecompositionOfAtMost}(G_r, k) \\
\text{for } j = r \text{ down to } 1 \text{ do} & \\
D_{j-1} & \leftarrow D_j \cup (G_{j-1} \setminus G_j) \quad // \text{undoing of reduce round } j \text{ in } D_{j-1} \\
D_{j-1} & \leftarrow \text{balance}(D_{j-1}) \\
D_{j-1} & \leftarrow \text{treeDecompositionOfAtMost}(D_{j-1}, k) \\
\text{return } D_0
\end{align*}
\]

### 5.1.1 Decreasing the size of a graph by a constant fraction

The procedure \( \text{reduce} \) was presented first by Bodlaender and Hagerup [BH98]. One invocation (in round \( i \)) of the \( \text{reduce} \) method decreases the size of the input graph at least by a constant fraction \( \frac{1}{2} \). This is done by finding a certain set of disjoint pairs of vertices of \( G \). Due to their disjointness, all pairs of vertices can be identified (reduced) in parallel. Therefore, after at most \( c \cdot \log n \) rounds, the resulting graph has constant size and thus a tree decomposition of constant width. Furthermore, when revoking the vertex identifications of round \( i \) the width of a tree decomposition of width \( k \) for \( G_{i+1} \) increases by at most \( k + 1 \).
5.1.2 Balancing a tree decomposition

The method balance computes, from an arbitrary tree decomposition \( T = (T, B) \) of width \( k \) for a graph \( G \), an efficiently evaluable tree decomposition \( D_e \) of width at most \( \ell = 7k \) (Definition 4.20). To this end, a contraction tree of \( T \) is computed. Thus, by the results of Chapter 4 on efficiently evaluable trees, several types of computations can be executed in a PEM-efficient manner. Such a computation is used for calculating the contents of the bags for \( D_e \). The implementation of the balance method is a simplified version of the algorithm of Bodlaender and Hagerup [BH98], yielding \( \ell = 7k \), instead of \( \ell = 3k \). However, this affects only hidden constants in the \( O \)-notation of the complexity measure.

The methods reduce and balance induce the framework for efficiently parallelizing the computation. Section 5.2 argues how they can be implemented efficiently in the PEM model.

5.1.3 Computing a tree decomposition of width \( k \)

In their PRAM algorithm, Bodlaender and Hagerup [BH98] use the algorithm of Bodlaender and Kloks [BK96] to compute a tree decomposition of width \( k \) for a graph \( G \) from a tree decomposition of width \( \ell \) for \( G \). This is possible since the dynamic programming approach [BK96] is parallelizable in a straightforward manner.

Similar to their approach, here a modification of the sequential algorithm for the EM model [MZ09] is used to implement treeDecompositionOfAtMost. For efficient parallelization, the algorithm requires a nice efficiently evaluable tree decomposition of width \( \ell \) as input.

The method treeDecompositionOfAtMost is explained in detail in Section 5.3.

5.2 A PEM-efficient load balancing framework

In this section PEM-efficient implementations of two procedures which have been introduced by Bodlaender and Hagerup for the PRAM model are presented [BH98]. First, the procedure reduce is presented. Afterwards, in Section 5.2.2, the procedure balance is described, which transforms a binary tree
decomposition of width $k$ into an efficiently evaluable tree decomposition of width $7k$. 

### 5.2.1 Decreasing the size of a graph by a constant fraction

The procedure `reduce` is designed for decreasing the size of the vertex set of a graph by one application by at least a constant fraction. Therefore, the following reduction rule, introduced by Bodlaender and Hagerup, is applied several times in parallel [BH98].

**Definition 5.1 (Reduction rule, Bodlaender and Hagerup [BH98])**

Let $v$ and $w$ be two vertices of $G$ that are adjacent or twins. The graph in which the pair $\{v,w\}$ is reduced is obtained from $G$ by identifying the vertices $v$ and $w$ and removing duplicate edges. The vertices $v$ and $w$ are called reduction partners.

The reduction rule has two properties that are especially important [BH98]. The first is, that any graph of reasonable size and bounded treewidth has many disjoint reduction partners.

**Lemma 5.2 (Bodlaender and Hagerup [BH98, Lemma 3.1])**

Let $d$ and $n_{max} = \mathcal{O}(k)$ be constants that depend only on $k$. Furthermore, let a vertex be of small degree if its degree is less than $d$. There is a constant $c' > 1$, such that in every graph of size $N$ with at least $n_{max}$ vertices and treewidth of at most $k$ there is a set of $\frac{N}{c'}$ pairwise disjoint reduction partners of small degree.

The disjointness of the reduction partners implies that all $\frac{N}{c'}$ reductions can be carried out in parallel. This implies that after applying `reduce` $\mathcal{O}(\log N)$ times to $G$, a graph $G_r$ of constant size is obtained. Note that therefore Step 5 of Algorithm 5.1 can be computed by brute force in a constant number of I/Os.

The second fundamental property is that undoing all reductions of a round, increases the treewidth of a tree decomposition of width $k$ by at most $k + 1$. This is, because all $k$ vertices in a bag can be expanded to at most two vertices. Thus, in the worst case, after expanding the vertices, there are at most $2k + 2$ vertices in a bag.
To actually use the reduction rule, an algorithm is needed to compute a set of at least $c \cdot N$ reduction partners. To this end, a conflict graph $C$ is computed in which each vertex represents two vertices of $G$ that are possible reduction partners. Bodlaender and Hagerup presented such an algorithm which takes constant time for the PRAM model [BH98, pp. 1736–1737]. Therefore, it can be simulated by Theorem 4.9 in $O(sort_P(N))$ parallel I/Os in the PEM model.

The conflict graph $C$ is defined as follows. For each adjacent pair of $G$ and for each pair of small degree vertices of $G$ that are $d$-acquainted twins, there is a vertex in the vertex set $V(C)$ (see Section 4.2.1 for computing $d$-acquainted twins). An edge between two vertices of $C$, representing pairs $(u_1, u_2)$ and $(v_1, v_2)$, implies that it is not possible to reduce both reduction partners in parallel (in the PRAM model). This can happen in three cases which are explained in the following.

1. If $u_1 = v_1$ (respectively $u_1 = v_2, u_2 = v_1$, or $u_2 = v_2$), there could be identified three or more vertices. Thus, this case is excluded by an edge $((u_1, u_2), (v_1, v_2))$ in $C$.

2. If $u_1$ and $v_1$ (respectively $u_1$ and $v_2$, $u_2$ and $v_1$, or $u_2$ and $v_2$) are adjacent, several edges may be modified at the same time. Thus, there is an edge $((u_1, u_2), (v_1, v_2))$ in $C$.

3. If $u_1$ and $v_1$ (respectively $u_1$ and $v_2$, $u_2$ and $v_1$, or $u_2$ and $v_2$) are adjacent entries in some adjacency list, there is an edge $((u_1, u_2), (v_1, v_2))$ in $C$ to prevent conflicts in editing adjacency lists.

Computing a fractional independent set on $C$ yields a set of vertices which can be reduced in parallel.

**Lemma 5.3**

Let $I$ be an algorithm which computes an independent set for a list graph of size $N$ in $O(q \cdot sort_P(N))$ parallel I/Os. Then, the procedure reduce can be implemented for a graph $G$ of size $N$ such that it takes $O(q \cdot sort_P(N))$ parallel I/Os in the CREW PEM model.

**Proof** As argued, the conflict graph $C$ can be computed in $O(sort_P(N))$ parallel I/Os by simulating the constant time PRAM algorithm of Bodlaender and
Hagerup. Since only small degree vertices and $d$-acquainted twins are considered, the degree of $C$ is bound by $O(\text{poly}(k))$.

Therefore, by Lemma 4.12, in $O(q \cdot \text{sort}_P(N))$ parallel I/Os, a fractional independent set $R$ of $C$ can be computed.

The reduction step, this means identifying reduction partners, can be implemented in $O(\text{sort}_P(N))$ parallel I/Os in the PEM model. For each pair $(v, w)$ of reduction partners all appearances of $w$ have to be replaced by $v$ in the edge set of $G$. This can be done for the source vertices of $E(G)$ by sorting $E(G)$ by the source vertices and scanning. By a second pass, all target vertices can be replaced, too. Since removing duplicate edges can be done by Lemma 4.3 efficiently, the reduced graph can be computed in $O(q \cdot \text{sort}_P(N))$ parallel I/Os. 

5.2.2 Making a tree decomposition efficiently evaluable

In order to apply the classical dynamic programming approach in a bottom-up manner efficiently to a tree decomposition $D$, it has to provide some advantageous structure. For example, a path decomposition is unfavorable, since each bag depends on its predecessor and thus a dynamic programming approach might be difficult to parallelize. In the PRAM model this favorable structure is a “balanced” tree decomposition of height $O(\log N)$ [BH98]. For the PEM model this is not sufficient, since for some parameter settings $\log N \in \omega(\text{listRank}_P^T(N))$. Therefore, for the PEM model this favorable structure is an efficiently evaluable tree decomposition $D_e$ (Definition 4.20).

The purpose of the procedure $\text{balance}$ is to transform an arbitrary tree decomposition $D = (T, B)$ of width $k$ into an efficiently evaluable tree decomposition $D_e$ of width $6k + 5$. This is done in three steps. Actually it still leads a balanced tree decomposition, since the procedure for computing an efficiently evaluable tree decomposition is based on the approach of Bodlaender and Hagerup that computes a balanced tree decomposition:

1. Starting with the tree $T$ of an arbitrary tree decomposition $D$ of width $k$, a binary tree decomposition $D_b = (T', B')$ of same width is generated.
2. An efficiently evaluable tree decomposition $D_a$ of width 5 for the binary tree $T'$ is computed.

3. The bags $B'$ of $D_b$ are embedded into the vertices of $D_a$. This results in an efficiently evaluable tree decomposition $D_e$ of width at most $6k + 5$.

In general the algorithm is the same as the algorithm of Bodlaender and Hagerup, but two steps which decrease the width of $D_a$ to 2 are omitted. Since these steps are technically easy, they can be implemented presumably efficiently in the PEM model, too, but are omitted here for a more compact representation. All the steps listed above are part of [BH98, Chapter 2]. Therefore, their correctness is not proven again, but the focus of this section is showing their efficient computability in the PEM model.

However, the construction of Bodlaender and Hagerup yields a tree decomposition of bounded width by using a contraction tree of $T'$. It exploits essentially the fact that long paths in $T'$ can be folded to a tree of logarithmic height, similar to an Euler tour construction where the path is the Euler tour of the tree with logarithmic height. Such a folding yields directly a tree decomposition the path of logarithmic height with small width.

**Lemma 5.4**

A rooted tree decomposition of width $k$ for a graph $G$ can be transformed into an efficiently evaluable tree decomposition of width $6k + 5$ for $G$ in $O(\text{listRank}_P^T(N))$ parallel I/Os in the CREW PEM model.

**Proof**  The first step, computing a binary tree decomposition $D_b = (T', B')$ from $D$ can be implemented by Lemma 4.14 in $O(\text{sort}_P(N))$ parallel I/Os in the PEM model as described in Section 4.3.2.

Step 2 is more complex. The tree $T_{D_a}$ of tree decomposition $D_a$ is an efficiently evaluable contraction tree of $T'$. Lemma 4.16 states that $T_{D_a}$ can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os. Recall that the leaves of the contraction tree $T_{D_a}$ are the vertices of $T'$ and are called base nodes. Furthermore, note that every internal vertex $x$ of $T_{D_a}$ represents a contraction of an edge $e_x$ of $T'$, and that the subtree rooted at $x$ corresponds to a subtree $T'(x)$ of contracted edges in $T'$.
For completing the description of $D_a$, its bags have to be computed. In this paragraph, the formal definition of the bags as stated by Bodlaender and Hagerup is given [BH98], while the next paragraphs show that the implied bags can be computed efficiently in the PEM model. To each node $x$ of $T_{D_a}$ a set $B(x)$ is assigned, which is called the border of $x$. Bodlaender and Hagerup define the border of $x$, as the base nodes of $T'(x)$ which are incident in $T'$ to at least one edge that has not been contracted in the subtree $T'(x)$ [BH98, p. 1728]. By the definition of the contraction sequence, here every border set contains at most 3 nodes [BH98, p. 1728]. The bag for the resulting tree decomposition $D_a$ of an internal node $x$ of $T_{D_a}$ that has children $y$ and $z$ is defined as $B(y) \cup B(z)$ [BH98, p. 1729]. Furthermore, the bag of a base node $x$ in $D_a$ is defined as $\{x\}$. Thus, since the border size of a vertex is at most 3, the treewidth of $D_a$ is at most 5.

It is not too hard to compute the border sets and thus the bags of $D_a$.

1. Note that the border sets for a leaf (base node) $x$ is $\{x\}$ and thus are trivially computable.

2. The border set for each internal node $x \in T_{D_a}$ can be determined from the borders of its children $y$ and $z$. In order to do so, the (border) sets $B(v) = \{v\}$ of leaves are annotated with the edges that are incident to $v$. By definition, the border of $x$ is the union of the borders of its children without the vertices which are only incident to the contracted edge $e_x$. The set $B(x)$ consists of the vertices of $B(y)$ and $B(z)$ which are not annotated by a mutual edge, which is the edge $e_x$. Keeping track of the edges with which the vertices of the border sets are annotated is visualized in Figure 5.1 and Figure 5.2.

Since the contraction tree $T_{D_a}$ is efficiently evaluable and the computation described in the last paragraph is performed in a bottom-up manner depending only on the direct children of a node, it can be computed by Lemma 4.19 in $O(\text{listRank}_P^I(N))$ parallel I/Os in the PEM model.

Step 3 can be easily implemented by sorting and scanning the bags of $D$ and the bags of $D_a$, and replacing the vertices in $D_a$ with their corresponding bags in $D$, yielding $D_e$.

Therefore, the number of parallel I/Os is bounded by $O(\text{listRank}_P^I(N))$. \qed
Figure 5.1: Visualization of a contraction tree sequence: starting with $T_0$, red edges are contracted to $T_1$. Further contracting yields $T_2, T_3, \ldots$ as depicted in Figure 5.2. To each tree $T_i$, to its left, the so far contracted edges are colored green in the original tree $T$. 
5.2 A PEM-efficient load balancing framework

Figure 5.2: Computing border sets of a contraction tree $C$: for each base node $x \in T$, $C$ has a leaf containing its incident edges that indicate the border of $x$. For an internal vertex, the border is determined from its children as the union of their borders without the vertices incident to the contracted edge $e$. The edge $e$ can be determined by keeping track of the incident edges as visualized by $C$. 
5.2.3 Making a tree decomposition nice

For the next section, the efficiently evaluable tree decomposition $D_e$ has to be a nice tree decomposition, such that the dynamic programming approach of Bodlaender and Kloks is applicable [BK96].

Since the underlying tree is binary, the task is to make sure that every bag is a join, forget, introduce or a start bag. First, every bag $x$ which has two children $y$ and $z$, has to be a join bag, meaning that the bags of $x$, $y$ and $z$ have the same content. This can be implemented easily by creating for each such bag $x$ two new bags $x_l$ and $x_r$, having the same contents, and connecting the old children $y$ and $z$ to $x_l$ and $x_r$ appropriately. The number of parallel I/Os needed are $O(sort_p(N))$ for sorting the edges of $D_e$ by degree and then scanning for introducing nodes and bags.

The tree decomposition $D_e$ may have chains of nodes of degree two whose bags coincide. However, by a simple bottom-up procedure these chains can be identified and afterwards left out. Since $D_e$ is an efficiently evaluable tree decomposition, by Lemma 4.19, the bottom up-procedure can be applied in $O(listRank^T_p(N))$ parallel I/Os in the PEM model.

Furthermore, the (set) difference between a bag $j$ and its parent bag $i$, if $i$ is not a join node, has to be exactly one vertex of the graph $G$. Thus, let the set $I = X_j \setminus X_i$ be the vertices introduced and $F = X_i \setminus X_j$ be the vertices forgotten between bag $i$ and bag $j$. This yields the number of bags by which an edge between $i$ and $j$ has to be replaced with: since the width of $D_e$ and thus $|F|$ and $|I|$ is at most $k' = 6k + 5$, a nice path decomposition between $i$ and $j$ can be computed in $f(k') = O(1)$ I/Os. First all vertices of $F$ are removed, one by one, and then the vertices of $I$ are added, one by one.

Thus in total a tree decomposition with $N$ bags can be transformed into a nice tree decomposition in $O(listRank^T_p(N))$ parallel I/Os.

Note that each bag is stretched to at most $2k' + 3$ bags. Thus the annotation of an efficiently evaluable tree can easily be modified, such that each processor processes for each bag its at most $2k' + 3$ bags while not exceeding the asymptotic bound for being efficiently evaluable.
5.3 Dynamic programming for refining tree decompositions

This section presents a PEM-efficient implementation for the procedure `treeDecompositionOfAtMost`. It computes from a nice efficiently evaluable tree decomposition of width $\ell$ a rooted tree decomposition of width $k$. If there is no such tree decomposition, the algorithm rejects. To this end, this section is divided into two parts. In the first part it is argued how a data structure, called “flippable DAG” [MZ09], can be replaced since it is not easily implementable in the PEM model in an efficient way. The second part describes the implementation of the just mentioned procedure `treeDecompositionOfAtMost`.

5.3.1 Replacing flippable DAGs

Maheshwari and Zeh introduced a data structure which they call “flippable DAG” for constructing path decompositions [MZ09]. They use flippable DAGs to construct implicit representations of path decompositions which are directed from a bag $s$ to a bag $t$. The construction of such a path decomposition is done iteratively. Similar to series parallel graphs, each construction step is either a serial composition or a parallel composition of two flippable DAGs. The direction of a path decomposition is stored in the direction of the edges of a graph. During these construction steps it may happen that a path decomposition has to change its direction. In this case, all edges of a graph have to be flipped such that all edges indicate the correct direction.

As this can happen asymptotically in every construction step, it is too expensive to perform those flips explicitly. Flippable DAGs are designed to store those flips and to output the final direction of an edge in a post processing step called `untangling`. The untangling procedure of Maheshwari and Zeh cannot be transferred easily to the PEM model since one of its steps uses time-forward processing (TFP) [Chi+95]. Actually, TFP is equivalent to the topologically sorted circuit value problem which is $P$-hard and therefore notoriously hard to parallelize [Lad75; GHR95].

Thus, in this PEM-efficient algorithm for reducing the width of a tree decomposition, flippable DAGs are replaced. For simplifying the replacement in Section 5.3.2, the following interface is defined.
Definition 5.5

An implicit path decomposition data structure $\mathcal{P}$ is a data structure, representing several graphs whose vertices are annotated. Each represented graph is connected, directed, and acyclic and has two labeled vertices $s$ and $t$. Such a graph is said to be directed from its only source vertex $s$ to its only sink vertex $t$. Every graph has a unique identifier $D_{st}$. Each vertex $\alpha$ of a represented graph is annotated by some constant-sized information, two sets $L_\alpha$ and $R_\alpha$ of vertices of a graph, and a natural number $\rho_\alpha$. The data structure supports 5 operations:

- Let $S$ be a set (valid) of graph identifiers. The untangle operation $\text{untangle}(S)$ returns in $O(\text{listRank}_{\mathcal{P}}(N))$ parallel I/Os for each graph identifier $D_{st}$ in $S$ a DAG, where $N$ is the number of the following operations applied (except $\text{flip}$). Each vertex $\alpha$ of a (output) DAG is annotated by the constant-sized information $L_\alpha, R_\alpha$ and $\rho_\alpha$. Furthermore, a graph represented by $D_{st}$ must satisfy the following properties of the operations.

- A new graph, representing only one vertex which is thus $s = t$, can be created by the operation $\text{new}(L_\alpha, R_\alpha, \rho_\alpha)$. It returns a unique graph identifier $D_{st}$ in $O(1)$ I/Os. The operation $\text{untangle}(D_{st})$ returns a single vertex $\alpha$.

- A serial composition $\text{serial}(D_{sxtx}, D_{syty})$ creates from the two graph identifiers $D_{sxtx}$ and $D_{syty}$ of $\mathcal{P}$ in $O(1)$ I/Os a new graph identifier $D_{sxty}$ in $\mathcal{P}$ such that $\text{untangle}(D_{sxtx}) \cup \{(t_x, s_y)\} \cup \text{untangle}(D_{syty})$ equals the DAG $\text{untangle}(D_{sxty})$.

- A parallel composition $\text{parallel}(D_{sxtx}, D_{syty})$ creates from graph identifiers $D_{sxtx}$ and $D_{syty}$ of $\mathcal{P}$ in $O(1)$ I/Os a graph identifier $D_{st}$ in $\mathcal{P}$ with new vertices $s$ and $t$. Furthermore, the DAG $\text{untangle}(D_{st})$ equals the DAG $\{(s, s_x), (s, s_y)\} \cup \text{untangle}(D_{sxtx}) \cup \text{untangle}(D_{syty}) \cup \{(t_x, t), (t_y, t)\}$.

- A flip operation $\text{flip}(D_{st})$, creates from a graph identifier $D_{st}$ in $\mathcal{P}$ a new graph identifier $D_{ts}$ in $\mathcal{P}$ in $O(1)$ I/Os such that $\text{untangle}(\text{flip}(D_{st})) = \text{untangle}(D_{ts})$ is true.
The operands of the operations $\text{flip}$, $\text{serial}$, and $\text{parallel}$ are, after the corresponding operation has been applied, not longer valid. A graph identifier $D_{st}$ allows access to the annotated information of $s$ and $t$: $L_s, R_s, \rho_s, L_t, R_t$, and $\rho_t$ in $O(1)$ I/Os. While $\rho_s$ and $\rho_t$ must be modifiable, the others may be read only. If $D_{st}$ is untangled, then in the resulting DAG every edge is on a path from $s$ to $t$.

It is easy to see that the flippable DAGs in the algorithm of Maheshwari and Zeh only use the operations provided by an implicit path decomposition data structure $\Pi$ [MZ09, pp. 446–456]. Thus, any data structure implementing the interface is a sufficient replacement for flippable DAGs.

### A parallel data structure

The following approach for replacing flippable DAGs seems to be fairly easy and thus simplifies the construction of Maheshwari and Zeh [MZ09]: one flippable DAG is replaced by two explicitly stored DAGs $G_{st}$ and $G_{ts}$, where $G_{st}$ ($G_{ts}$) is a DAG in which each edge is on a directed path from $s$ ($t$) to $t$ ($s$).

In the following, the implicit path decomposition data structure is described that is used in the algorithm of the next section. It is called $\text{DAG tuple data structure}$ and it only manages a set of edges and provides unique (graph) identifiers for new vertices. Each graph identifier $D_{st}$ is represented by a tuple of pairs of annotated vertices $\Psi_{st} = ((s,t),(t',s'))$. In the following it is shown how the operations $\text{new}$, $\text{serial}$, $\text{parallel}$, $\text{flip}$, and $\text{untangle}$ of the DAG tuple data structure $\mathcal{D}$ can be implemented such that they fulfill the requirements of an implicit path decomposition data structure.

- The operation $\text{new}(L,R,\rho)$ provides two new unique vertices $v$ and $v'$ which are both annotated by $L, R,$ and $\rho$ and returns the graph identifier $\Psi_{st} = ((v,v'),(v',v'))$, with $s = t = v$. A new unique vertex can be provided by an internal counter for each processor, which is then made unique by the identifier of the processor. Note that a call to this operation does not create a new edge in $\mathcal{D}$, but only creates the new graph identifier $\Psi_{st}$. 

105
Chapter 5 Computing tree decompositions in the PEM model

- The method \(\text{serial}(\Psi_{x}^{s_{x}t_{x}}, \Psi_{y}^{s_{y}t_{y}})\) returns the graph identifier \(\Psi_{x}^{s_{x}t_{x}}\), and is implemented as follows: the graph identifiers \(\Psi_{x}^{s_{x}t_{x}} = ((s_{x}, t_{x}), (t'_{x}, s'_{x}))\), and \(\Psi_{y}^{s_{y}t_{y}} = ((s_{y}, t_{y}), (t'_{y}, s'_{y}))\) are given as input. The resulting graph identifier \(\Psi_{x}^{s_{x}t_{x}} = ((s_{x}, t_{x}), (t'_{x}, s'_{x}))\) can be computed in \(O(1)\) I/Os by adding the edges \((t_{x}, s_{y})\), and \((s'_{y}, t'_{x})\) to \(D\).

- Similarly, the method \(\text{parallel}(\Psi_{x}^{s_{x}t_{x}}, \Psi_{y}^{s_{y}t_{y}})\) returns the graph identifier \(\Psi_{s_{x}t_{x}}\), and it is implemented as follows: let \(\Psi_{x}^{s_{x}t_{x}} = ((s_{x}, t_{x}), (t'_{x}, s'_{x}))\), and \(\Psi_{y}^{s_{y}t_{y}} = ((s_{y}, t_{y}), (t'_{y}, s'_{y}))\) be the graph identifiers that are given as input. The resulting graph identifier \(\Psi_{s_{x}t_{x}} = ((s, t), (t', s'))\) can be computed in \(O(1)\) I/Os for new unique vertices \(s, t, s', t'\) by adding the edges \((s, s_{x}), (s, s_{y}), (t_{x}, t), (t_{y}, t), (t', t'_{x}), (t', t'_{y}), (s'_{x}, s'), (s'_{y}, s')\) to \(D\).

- Let the graph identifier \(\Psi_{s_{x}t_{x}} = ((s, t), (t', s'))\) be the input of \(\text{flip}\). The resulting graph identifier is \(\Psi_{ts} = ((t', s'), (s, t))\), which clearly can be created in \(O(1)\) I/Os.

- The method \(\text{untangle}(\Psi_{s_{x}t_{x}})\), for the graph identifier \(\Psi_{s_{x}t_{x}} = ((s, t), (t', s'))\), applied on \(D\) should simply return the maximal subgraph \(D\) such that \(s\) is a predecessor of all vertices in \(D\). The subgraph \(D\) can be extracted by the following procedure: from every DAGs a 1-in-tree is computed. Thus, for each of them, an Euler tour can be computed. By a prefix sums computation, all vertices in \(D\) can be labeled. By a filtering step, all edges which are incident to a vertex in \(D\) can be obtained. The dominating step is the prefix sums computation task which takes \(O(\text{listRank}_{T}(N))\) parallel I/Os. Note that graphs consisting of a single vertex can be returned trivially.

Note that here flippable DAGs can be replaced by storing two explicit DAGs, since the algorithm of Maheshwari and Zeh uses only a limited subset of the features of flippable DAGs. This limited subset of features is defined by the implicit path decomposition data structure. It is not clear how the full (but unused) functionality of flippable DAGs can be implemented efficiently in the PEM model. However, replacing flippable DAGs by the DAG tuple data structure works for the EM model algorithm to compute a tree decomposition [MZ09] as well.
5.3 Dynamic programming for refining tree decompositions

5.3.2 Refining a tree decomposition

Algorithm 5.2 sketches how to compute from a given efficiently evaluable, nice tree decomposition of width $\ell$ a tree decomposition of width $k$. It was originally presented for the EM model [MZ09, Algorithm 5].

Algorithm 5.2: Pseudo code for treeDecomposition0fAtMost($D, k$).

**Input**: A graph $G = (V, E)$ of size $N$, an efficiently evaluable, nice tree decomposition $D = (X, T')$ of width at most $\ell$ and size $O(N)$ for $G$, and a constant $k \in \mathbb{N}$.

**Output**: A tree decomposition $E$ of width $k$ and size $O(N)$ for $G$ or the answer that $\text{tw}(G) > k$.

1. Check if $\text{tw}(G) < k$ by the testing algorithm of Bodlaender and Kloks
2. if $\text{tw}(G) > k$ then
3. \hspace{1em} return $\text{tw}(G) > k$.
4. Extract from the full sets of characteristics of Step 1 for each $x \in T'$ a characteristic corresponding to a certain set of tree decompositions for $G$ of width $k$ and thus build a “tree of characteristics” $C$.
5. Compute from $C$ an implicit tree decomposition $E = (Y, U)$.
6. Compute $E$ explicitly.
7. return $E$

In this section, among others, the terms of characteristics and full sets of characteristics are used. A characteristic for some node $i \in T'$ is a constant-sized description of a constant number of tree decompositions for the subgraph $G_i$, which is contained in the bags of the subtree rooted at $i$. However, here characteristics are used as identifiers for several constant time operations that are used as a black box. Thus, the exact definition of a characteristic is omitted, but can be found in [MZ09], based on the definitions of [BK96]. A full set of characteristics for some node $i \in T'$ is a constant-sized set of characteristics so that for every tree decomposition $D_i$ for $G_i$ there is a characteristic representing a tree decomposition which is at least as “good” as $D_i$ in a certain sense [BK96; MZ09]. The following lemmas state properties that are fundamental to the algorithms of Maheshwari and Zeh, and Bodlaender and Kloks. However, some of them have probably not been stated by a lemma on their own [MZ09; BK96].
Lemma 5.6
Let $\ell$ and $k$ fixed constants, and let $D = (X, T')$ be a nice tree decomposition of width $\ell$ for a graph $G$. The dynamic programming approach of Bodlaender and Kloks [BK96] for testing if $G$ has treewidth $k$, computes for each bag in $X$, a constant-sized full set of characteristics from the (full) sets of characteristics of its children, taking for each bag $O(1)$ time in the RAM model.

Furthermore, it is possible to compute and store $O(1)$ pointers for each characteristic, that have constant size itself [MZ09, Section 6.2.1]. Among other things they represent from which children’s characteristics a characteristic is obtained. If these pointers are computed for each set of characteristics, it is called a set of augmented characteristics.

Lemma 5.7 (Maheshwari and Zeh [MZ09, Section 6.2.2])
Let $x$ be a bag of $D$ and $y$ a child of $x$. Assume that the full sets of augmented characteristics for $x$ and $y$ are given. Then for every characteristic $C$ of the full set of characteristics of a bag $x$ the characteristic of $y$, from which $C$ has been computed, can be determined in $O(1)$ I/Os.

Lemma 5.8 (Maheshwari and Zeh [MZ09, Section 6.2.4])
Let a “tree of characteristics” $C$ be given. By spending $O(1)$ I/Os for each bag $x$ of $D$, a constant number of graph identifiers in an implicit path decomposition data structure $\mathcal{P}$ can be computed, such that the computations for $x$ depend only on the characteristics and on the graph identifiers in $\mathcal{P}$ of the children of $x$. Furthermore a link list $\mathcal{L}$, defining how these implicit path decompositions (graph identifiers) form a tree decomposition, can be computed in $O(1)$ parallel I/Os for each bag in a bottom-up fashion.

Lemma 5.9 (Maheshwari and Zeh [MZ09, Section 6.2.5])
Let the implicit path decomposition data structure $\mathcal{P}$ and the link list of the previous lemma be given. By only applying the untangle-operation of $\mathcal{P}$ to all valid identifiers $I$ simultaneously, an implicit tree decomposition of width $k$ can be computed.

The following lemma is the main result of this section, showing that Algorithm 5.2 can be implemented efficiently in the PEM model.
Lemma 5.10
Let the natural numbers $\ell$ and $k$ be constants, and let an efficiently evaluable, nice tree decomposition $D$ of width $\ell$ for a graph $G$ with $N$ vertices be given. Then it can be decided if $\text{tw}(G) \leq k$ and if so, such a tree decomposition can be constructed in $O(\text{listRank}_P^T(N))$ parallel I/Os in the CREW PEM model.

Proof. For proving the lemma, the different steps of Algorithm 5.2 are considered. First it is argued how the steps can be implemented in the PEM model, and finally the efficiency of the implementation is proven. The algorithmic steps are the same as in [MZ09, Algorithm 5], differing only but crucially in the way how they are implemented in the PEM model. Thus, this proof does not argue about the correctness of Algorithm 5.2, as it follows from the arguments given by Maheshwari and Zeh [MZ09].

A central lemma of Bodlaender and Kloks states that if the root of $T'$ has a non-empty full set of characteristics, then there exists a tree decomposition of width $k$ [BK96, Lemma 4.2]. Lemma 5.6 states that it is possible to compute for each bag in $O(1)$ I/Os a full set of characteristics, while depending only on the results of the bag’s children. Since $D$ is efficiently evaluable it follows by Lemma 4.19 that the full set of characteristics for the root of $T'$ can be computed in a bottom-up manner efficiently in the PEM model. Therefore the $O(N)$ steps that each take $O(1)$ I/Os of Step 1 can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os.

Thus, by Lemma 5.6 for each bag its full set of characteristics is known. By selecting for the root bag of $D$ an arbitrary characteristic $C$ of its full set of characteristics a tree of characteristics $C$ can be computed as follows. Lemma 5.7 implies that, by processing $T'$ top-down, for each bag one characteristic from which $C$ has been computed (transitively). Again, since $T'$ is efficiently evaluable, by Lemma 4.19, the tree of characteristics $C$ can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os.

Since $C$ and $T'$ have the same layout, $C$ is also efficiently evaluable. Therefore, by Lemma 4.19, the operation implied by Lemma 5.8 can be applied to all vertices of $C$ in $O(\text{listRank}_P^T(N))$ parallel I/Os. Thus, using the DAG tuple data structure in the operation of Lemma 5.8, an implicit path decomposition data structure $D$ and a link list $L$ can be obtained efficiently. Furthermore, by Lemma 5.9,
D and L can be transformed by applying the untangle operation into an implicit tree decomposition E in $O(\text{listRank}_p^T(N))$ parallel I/Os, completing Step 5.

Having an implicit tree decomposition, Step 6 can be implemented efficiently in the PEM model by using Algorithm 4.3. Lemma 4.23 states that this last step can be implemented in $O(\text{listRank}_p^T(N))$ parallel I/Os.

As the Steps 1 – 5 all use the bottom-up or top-down processing of constant-sized information on an efficiently evaluable tree, all these steps can be computed in $O(\text{listRank}_p^T(N))$ parallel I/Os. Since the last step needs asymptotically as many parallel I/Os, the lemma follows.

\[\square\]

5.4 Analysis

The analysis for the complexity of Algorithm 5.1 is split into three parts, depending on which algorithm for computing fractional independent sets is used.

5.4.1 Independent sets by delayed pointer processing

In the CREW PEM model with $M = B^{O(1)}$, the expression $\text{listRank}_p^T(N)$ can be bounded by $O(\text{sort}_p(N) + B \cdot (\log B) \cdot (\log^t N) \cdot \log M \frac{N}{B})$, if independent sets are computed by delayed pointer processing (Lemma 3.7). This term consists of the usual sorting term and the additive term $B \cdot (\log B) \cdot (\log^t N) \cdot \log M \frac{N}{B}$ covers the (recursive) cases in which the running time does not scale with the number of processors anymore. In the following analysis, this term is used for $\text{listRank}_p^T(N)$.

**Theorem 5.11**

Let $k$ and $t$ be a fixed natural number. Computing for a graph $G$ with $N$ vertices a tree decomposition of width $k$ or deciding that $\text{tw}(G) > k$ can be done in $O(\text{sort}_p(N))$ parallel I/Os in the CREW PEM model if $M = B^{O(1)}$ and $P \leq \frac{N}{B^2(\log B)(\log N) \log^t N}$. 

110
Proof  First, note that the restrictions on the number of processors and the size of the caches of the PEM model yield $\text{sort}_P(N) = \frac{N}{PB \log M \frac{N}{B}}$.

By Lemma 5.10, Lemma 5.3 and Lemma 5.4, the methods reduce, balance, and treeDecompositionOfAtMost need $O(\text{listRank}_P(N))$ parallel I/Os. The number of parallel I/Os needed for the algorithm adds up over all passes of the $O(\log N)$ recursive steps each taking $O(\text{listRank}_P(N))$ parallel I/Os. Thus, the number of parallel I/Os needed for the second loop of Algorithm 5.1 can be bounded asymptotically by:

$$\log N \sum_{i=0}^{\log N} \text{listRank}_P \left( \frac{N}{c^i} \right) \leq \log N - (\log(PB^2 \log(t) N) - 1) + \sum_{i=0}^{\log N} \text{sort}_P \left( \frac{N}{c^i} \right) + \sum_{j=\log N - (\log(PB^2 \log(t) N))}^{\log N} B \cdot (\log B) \cdot (\log(t) N) \cdot \log M \frac{N}{B}.$$

Note that if $N \leq PB^2 \log(t) N$, the term $\text{sort}_P(N)$ is dominated by the term $B \cdot (\log B) \cdot (\log(t) N) \cdot \log M \frac{N}{B}$.

The first part of this sum can be bounded by $O(\text{sort}_P(N))$ parallel I/Os due to the geometrically decreasing input sizes. A rough estimation of the last sum yields:

$$O(\text{sort}_P(N)) + (\log N) \cdot B \cdot (\log B) \cdot (\log(t) N) \cdot \log M \frac{N}{B}.$$

The bound on the number of processors implies

$$B \cdot (\log B) \cdot (\log(t) N) \leq O \left( \frac{N}{PB} \right)$$

and thus the number of parallel I/Os is bounded by $O(\text{sort}_P(N))$. □

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 [BH98] 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.
5.4.2 Analysis for the full parameter range

If independent sets are computed by coin tossing or deterministic coin tossing, the expression $\text{listRank}_P^T(N)$ is bounded by $O(q \cdot \text{sort}_P(N) + (\log P) \cdot \log \frac{B}{\log P})$ with $q \in O(1)$ or $q = O(\log^* N)$.

**Theorem 5.12**

Let $k$ be a fixed natural number. Computing for a graph $G$ with $N$ vertices a tree decomposition of width $k$ or deciding that $\text{tw}(G) > k$, can be done in the CREW PEM model in

$$O\left(q \cdot \left(\text{sort}_P(N) + (\log N) \cdot (\log P) \cdot \log \frac{B}{\log P}\right)\right)$$

parallel I/Os, if an FIS algorithm is used which takes $O(q \cdot \text{sort}_P(N))$ parallel I/Os.

**Proof**  The number of parallel I/Os needed for Algorithm 5.1 adds up over all passes of the $O(\log N)$ recursive steps each taking $\text{listRank}_P^T(N)$ parallel I/Os.

By Lemma 5.10, Lemma 5.4 and Lemma 5.3, the methods reduce, balance, and $\text{treeDecompositionOfAtMost}$ can be computed in $O(\text{listRank}_P^T(N))$ parallel I/Os. Since the input size for each pass is geometrically decreasing, the number of parallel I/Os needed for the loops, can be bounded by:

$$\sum_{i=0}^{\log N} \text{listRank}_P^T\left(\frac{N}{c^i}\right) \leq \sum_{i=0}^{\log N} \left(\text{sort}_P\left(\frac{N}{c^i}\right) + (\log P) \cdot \log \frac{B}{\log P}\right) \leq t \cdot \left(\text{sort}_P(N) + (\log N) \cdot (\log P) \cdot \log \frac{B}{\log P}\right).$$

Using the randomized independent set construction of Lemma 3.5, this yields the following upper bound for a randomized PEM machine.
Corollary 5.13

Let $k$ be a fixed natural number. Computing for graph $G$ with $N$ vertices a tree decomposition of width $k$ or deciding that $\text{tw}(G) > k$, can be done in the CREW PEM model, in expectation, in

$$O\left(\text{sort}_P(N) + (\log N) \cdot (\log P) \cdot \frac{B}{\log P}\right)$$

parallel I/Os.

Furthermore, using the deterministic coin tossing technique, by Lemma 3.1 the following bound holds for a deterministic PEM machine.

Corollary 5.14

Let $k$ be a fixed natural number. Computing for a graph $G$ with $N$ vertices a tree decomposition of width $k$ or deciding that $\text{tw}(G) > k$ can be done in the CREW PEM model in

$$O\left((\log^* N) \cdot \left(\text{sort}_P(N) + (\log N) \cdot (\log P) \cdot \frac{B}{\log P}\right)\right)$$

parallel I/Os.

5.5 A permuting lower bound

The following theorem yields, based on the lower bound for the Proximate Neighbors problem, a permuting lower bound for deciding the Treewidth-$k$ problem in the PEM model. As in the previous lower bounds that are based on the Proximate Neighbors problem, a restriction on the actions of the PEM is needed. The presented reduction works on a graph in the edge list representation. Each edge $e$ consists of two atoms, that contain the vertices that are incident to $e$.

Definition 5.15

The equal PEM model is an extension of the atomic PEM model by the binary operation $\equiv$. Let $x$ and $y$ be two atoms in the cache of a processor that contain vertices $u$ and $v$ of a graph $G$. Then $x \equiv y$ is true if and only if $u = v$. 
The proof of the following theorem is based on the fact, that there is a hard instance for which it has to be tested, that certain vertices are not equal.

**Theorem 5.16**
Let \( k \) be a fixed natural number. There is for every \( k \) a connected graph \( G \) with \( N \) vertices and \( O(N) \) edges such that the following holds: every randomized algorithm that decides if \( G \) has treewidth \( k \) requires \( \Omega(\text{perm}_P(N)) \) parallel I/Os in the equal PEM.

**Proof** Let \( I_P \) be a hard instance of the Proximate Neighbors problem whose input is given by \( x_1, \ldots, x_{\frac{N}{2}}, y_{\pi(1)}, \ldots, y_{\pi(\frac{N}{2})} \) for some permutation \( \pi \) such that for all \( 1 \leq i \leq \frac{N}{2} \) it holds \( \lambda(i) = \{x_i, y_i\} \).

The following graph \( G \) on \( N + k + 2 \) vertices provides a hard instance for the Treewidth-\( k \) problem: there is a complete graph on \( k \) vertices \( v_1, \ldots, v_k \) and two vertices \( u \) and \( w \) that each are connected to all vertices of the complete graph. Observe that this is a complete graph on the vertices \( u, w, v_1, \ldots, v_k \) where the edge \( \{u, w\} \) is removed. Furthermore, there are \( N \) edges \( \{u, x_i\} \) and \( \{y_j, w\} \) for \( 1 \leq i \leq \frac{N}{2} \). The input for the Treewidth-\( k \) problem is given as a list of edges in the following order: \( \{u, x_1\}, \ldots, \{u, x_{\frac{N}{2}}\}, \{y_{\pi(1)}, w\}, \ldots, \{y_{\pi(\frac{N}{2})}, w\} \), followed by the edges of the clique subgraph and the edges that are incident to it.

Observe the following two facts on the input instance:

1. If there is some \( i \) such that \( x_i = y_i \), then \( G \) contains a clique of size \( k + 2 \) as a minor on the vertices \( v_1, \ldots, v_k, u, w, x_i = y_i \). Therefore, in this case, the input graph has treewidth at least \( k + 1 \).

2. If for all \( 1 \leq i \leq \frac{N}{2} \): \( x_i \neq y_i \), then \( G \) has treewidth \( k \) by the following tree decomposition. For each edge \( \{u, x_i\} \in G \) (respectively \( \{w, x_i\} \in G \)), there is a bag \( U_i = \{u, x_i\} \) (\( W_i = \{w, x_i\} \)). All the bags \( U_i \) (\( W_i \)) are connected to a bag \( U = \{u\} \) (\( W = \{w\} \)). The bags \( U \) and \( W \) are connected by the following path decomposition: \( \{u, v_1, \ldots, v_k\}, \{w, v_1, \ldots, v_k\} \). Since no bag contains more than \( k + 1 \) vertices, the described tree decomposition has width \( k \).

For any deterministic program its execution is considered on the input that has treewidth \( k \). This means that \( x_i \neq y_i \) for all \( 1 \leq i \leq \frac{N}{2} \). Because the program is correct, it has to move the input atoms in the following way: every pair \( x_i, y_i \), must be located, at least once, simultaneously in the memory of a processor for
checking their inequality. Otherwise, there would be an unchecked pair \( x_i, y_i \), and the input with \( x_i = y_i \) would lead the algorithm to an incorrect output. Hence, such a program solves the instance \( I^P_A \) of the **Proximate Neighbors** problem and must use \( \Omega(\perm_p(N)) \) parallel I/Os.

As argued in Section 3.4.3, a lower bound for a program solving a hard instance yields a lower bound for randomized algorithms. Thus, the theorem follows. □

### 5.6 Conclusion

This chapter states for every fixed \( k \in \mathbb{N} \) a PEM-efficient algorithm that computes for a graph \( G \) a tree decomposition of width \( k \) or decides that \( \text{tw}(G) > k \). As most efficient algorithms for computing a tree decomposition, this algorithm also uses the dynamic programming approach of Bodlaender and Kloks. This approach has been transfered to the EM model previously [MZ09] and is used here, too. Central to the algorithm is the load balancing approach as known from the PRAM algorithm of Bodlaender and Hagerup. Here this approach is implemented in a PEM-efficient way by using the building blocks of Chapter 4 [BH98]. The load balancing is based on the fact, that a contraction tree can be computed efficiently in the PEM model. Thus, computations on the tree, like the dynamic programming approach, can be implemented efficiently.

Another approach to implement the dynamic programming approach is to consider it as a type of a tree expression which is evaluated. However, this approach has not been chosen in order to omit technical details in the dynamic programming approach, which is used in the given implementation as a black box. Furthermore, due to the overall structure of the algorithm an approach via tree expression evaluation would not lead to a better I/O-complexity.

The problem of computing a path decomposition of width \( k \) has not been considered explicitly in this chapter. However, Bodlaender and Kloks state that a path decomposition can be computed from a tree decomposition of bounded width by a similar dynamic programming approach [BK96]. Furthermore, the tools adapted from the algorithm for the EM model can be used to compute a path decomposition of bounded width efficiently in the PEM model, too [MZ09].
By an easy permuting lower bound the given algorithm is shown to be asymptotically optimal for a large parameter range.
CHAPTER 6

Linear-sized kernelizations for the PEM model

A 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 fixed-parameter tractable problem $\Pi$ for a parameter $\kappa$ admits a kernelization, which is an algorithm that in polynomial time reduces any instance of $\Pi$ of size $N$ to 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 one. 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 $2\kappa$ vertices, where $\kappa$ denotes the size of a minimum vertex cover of $G$ [FG06].

The classical view on kernelization in the RAM model is to solve an instance $I$ of a hard problem $\Pi$ 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 $\kappa$. The second phase solves the problem $\Pi$ 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 to a total running time of $O(p(N) + f(g(\kappa)))$ 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 an efficient implementation of this approach. Recently, efficient kernelization algorithms for the Planar Dominating Set problem have been proposed [Hag11; Bev+11]. The algorithms of Hagerup [Hag11] and Bevern et al. [Bev+11] need linear time in the RAM model and therefore improve the previously known linear-size kernels that took cubic time.
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 $I'$ that can be handled. In this chapter are therefore algorithms for the first phase considered.

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 [Fom+10; Kim+13]. Their results are summarized by the following lemmas:

**Lemma 6.1 (Fomin et al. [Fom+10])**

On any class of graphs excluding a fixed apex graph as a minor, all separable contraction-bidimensional problems with finite integer index that are parameterized by solution size $\kappa$ admit kernels of size $O(\kappa)$.

Graph classes excluding a fixed graph $H$ as a minor can be generalized to classes excluding $H$ as a topological minor.

**Lemma 6.2 (Kim et al. [Kim+13])**

On any class of graphs excluding a fixed graph as a topological minor, all treewidth-bounding graph problems with finite integer index that are parameterized by solution size $\kappa$ admit kernels of size $O(\kappa)$.

Further results by Fomin et al. provide randomized linear-time kernelizations for these optimization problems [Fom+12b]. There are some reservations against the practicability of the algorithms implied by these meta-results, namely very large constant factors hidden in the $O$-notation and some difficulties in actually creating certain constant-sized structures.

In this chapter it is shown that the randomized kernelization algorithms of Fomin et al. can be efficiently implemented in the PEM model. On the one hand this algorithm does not improve the large constant factors in the running time. On the other hand it shows that there are no fundamental obstacles when implementing their algorithms in the PEM model.
6.1 Finding and replacing protrusions

In this section a PEM-efficient implementation of an algorithm of Fomin et al. is presented that replaces protrusions \([\text{Fom}+12b; \text{Fom}+12a]\). Their algorithm is applied to \(N\)-vertex graphs \(G\) from classes \(\mathcal{G}_H^\text{top}\) and \(\mathcal{G}_H^\text{top}\), where \(H\) is some fixed graph. Note that the graph classes imply that all handled graphs \(G\) have \(E(G) = O(|V(G)|) = O(N)\) edges.

The algorithm of Fomin et al. applies one reduction rule repeatedly on the input graph \(G\) to reduce its size. This rule was introduced by Fomin et al. [\text{Fom}+10]:

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

By [\text{Kim+13}, Lemma 1] for any parameterized problem \(\Pi\) 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 \(\Pi\) restricted to the graph class \(\mathcal{G}\). The safety of the reduction rule \((\star)\) is proven in [\text{Kim+13}, p. 620].

Since the set \(\mathcal{R}_{\Pi, \tau}\) depends on \(\tau\), our kernelization algorithms (as well as the known polynomial-time kernelizations [\text{Kim+13}; \text{Fom}+12b]) are non-uniform in \(\tau\) and thus \(\kappa\). On the other hand, in the following it is valid to assume that \(\mathcal{R}_{\Pi, \tau}\) is known explicitly to the algorithms.
Chapter 6 Linear-sized kernelizations for the PEM model

The set $R_{\Pi, \tau}$ is finite for every fixed $\tau$ and for every $\Pi$ that has FII. Therefore, the protrusion limit of $\Pi$ restricted to a graph class $\mathcal{G}$ is well-defined as $\rho_{\Pi, \mathcal{G}}(\tau) = \max_{G \in R_{\Pi, \tau}} |V(G)|$ [Kim+13].

The key result to be proven in this chapter is a PEM-efficient implementation of an algorithm that applies the reduction rule ($\star$) repeatedly. To this end, a randomized PEM-efficient protrusion finder yielding a set $P$ of protrusions, and a PEM-efficient protrusion replacer, replacing constant-sized protrusions, is presented. Since not all protrusions of $P$ are of constant size, in Lemma 6.5, a PEM-efficient algorithm for replacing protrusions of unbounded size by constant-sized protrusions is presented.

### 6.1.1 PEM-efficient protrusion finder

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

**Lemma 6.3**

Let $t \in \mathbb{N}$ be a constant. The randomized fast protrusion finder of Fomin et al. [Fom+12b; Fom+12a] can be implemented in the CREW PEM model with $P \leq N/(B^2(\log B)(\log N)\log(t)N)$ and $M = B^{O(1)}$, such that it takes for an input graph $G$ of size $N$ in expectation $O(sort_p(N))$ parallel I/Os. It computes a set $\mathcal{P}$ of $\tau$-protrusions such that for every $X \in \mathcal{P}$: $|X| > \rho_{\Pi, \mathcal{G}}(2\tau)$ holds.

**Proof** First, the original algorithm of Fomin et al. is described [Fom+12a], and afterwards it is argued that its steps can be implemented efficiently in the PEM model. Let $r = \rho_{\Pi, \mathcal{G}}(2\tau)$.

**Step 1:** Randomly partition $V(G)$ into $r + 1$ sets $X_1, \ldots, X_{r+1}$.

**Step 2:** For each $X_i$, where $i = 1, \ldots, r + 1$, compute the set of connected components of $G[X_i]$ and add them to a collection $\mathcal{C}'$.

**Step 3:** Compute from $\mathcal{C}'$ a collection $\mathcal{C}^*$ of “protrusion candidates” as follows: add a connected component $C' \in \mathcal{C}'$ to $\mathcal{C}^*$ if and only if $C'$ is an $r$-protrusion, this means if $|N(C')| \leq r$ and $tw(C' \cup N(C')) \leq r$.

**Step 4:** Store the “large” connected components of $\mathcal{C}^*$ in a set $\mathcal{P}$ of protrusions, this means those $C^* \in \mathcal{C}^*$ with $|V(C^*)| > 2r$. Let the remaining components
be $\mathcal{C} = \{C_1, \ldots, C_t\}$.

**Step 5:** Partition $\mathcal{C}$ into equivalence classes $\mathcal{Z}_1, \ldots, \mathcal{Z}_q$, such that for all $h \in \{1, \ldots, q\}$ it holds $N(C) = N(C')$ for all $C, C' \in \mathcal{Z}_h$.

**Step 6:** As long as there is a class $\mathcal{Z}_j$ with $4r \geq \sum_{J \in \mathcal{Z}_j} |J| > 2r$ select a subset $\mathcal{Z} \subseteq \mathcal{Z}_j$ with $4r \geq \sum_{J \in \mathcal{Z}} |J| > 2r$ and add $\mathcal{Z}$ to $\mathcal{P}$ while removing it from $\mathcal{Z}_j$.

This completes the description of the algorithm.

In the classical RAM model the only computationally hard step is computing the treewidth of a graph.

In the following, the expected number of parallel I/Os for their algorithm is analyzed.

- Step 1 can be implemented by scanning the vertices of $G$ and assigning randomly a number to each vertex. The sets $X_1, \ldots, X_{r+1}$ are then obtained by sorting the vertices by their color in $O(sort_p(N))$ I/Os.

- In Step 2 the connected components can be computed in $O(sort_p(N))$ parallel I/Os, by the algorithm of Arge, Goodrich, and Sitchinava for graphs that are sparse under contraction [AGS10].

- For Step 3 the algorithm of Chapter 5 can be used to check if the connected components have small treewidth. This takes $O(sort_p(N))$ parallel I/Os. The size of the neighborhoods can be checked by sorting the edges of each component by the index of the connected component of the source vertices and counting (by a prefix sum) the edges which occur only once.

- Step 4 can be implemented by a prefix sums computation on the input graph, as it is sorted by the connected components. The sets $\mathcal{C}^*$ and $\mathcal{C}$ can be obtained by sorting.

- In Step 5 the connected components $\mathcal{C}$ are sorted by their neighborhood as follows: each vertex name is interpreted as a number. Since each neighborhood consists of at most $r$ vertices, it can be encoded by a number with $r$ digits. By sorting the connected components by their neighborhood string, $\mathcal{Z}$ is obtained. Sorting needs $O(sort_p(N))$ parallel I/Os since $r$ is a constant.
• Step 6 is achieved by sorting all components of all $Z_i$ non-decreasingly by their size which can be achieved by one sorting step. Then, for each component its size is attached to an edge of a linked list, where each class $Z_i$ is represented by a list graph. By a list contraction algorithm [AGS10], every time when two edges are contracted that represent components with a total size of more than $2r$, the corresponding components are added to $P$. Note that a class that is added to $P$ has therefore a maximum size of $4r - 2$.

Since all steps can be executed in the CREW PEM model with $M = B^{O(1)}$ and $P \leq N/(B^2(\log B)(\log N)\log^{(t)} N)$ in $O(sort_P(N))$ parallel I/Os, the lemma follows. 

6.1.2 PEM-efficient protrusion replacer

In this section, based on the ideas of Bodlaender et al., a PEM-efficient algorithm is given for replacing all $\tau$-protrusions in parallel [Bod+09].

In the RAM model there is a constant time algorithm to replace constant-sized $\tau$-protrusions [Kim+12, Lemma 5]. A trivial simulation of this algorithm requires no more than $O(1)$ I/Os, which yields the following lemma:

Corollary 6.4

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

Recall that the protrusion finder of Fomin et al. finds a collection $P$ of $\tau$-protrusions $X$ of size $|X| > \rho_{\Pi,G}(2\tau)$. Since Corollary 6.4 can handle only protrusions of constant size, a PEM-efficient implementation of an algorithm [Fom+12b] is presented which replaces protrusions that are larger than $2\rho_{\Pi,G}(2\tau)$ by smaller $(2\tau + 1)$-protrusions.

Since $R_{\Pi,\tau}$ exists for a problem $\Pi$ for all $\tau$, the resulting protrusions can then be replaced by Corollary 6.4.
Lemma 6.5
Let \( t \in \mathbb{N} \). Let \( \tau \in \mathbb{N} \), and let \( \Pi \) be a graph problem with finite integer index on a graph class \( G \). Given a set of \( \tau \)-protrusions \( P \) for a graph \( G \in G \) of size \( N \), the number of parallel I/Os to find for all \( X \in P \) with \( |X| > \rho_{\Pi,G}(2\tau) \), a \((2\tau + 1)\)-protrusion \( Y \subseteq X \) satisfying \( \rho_{\Pi,G}(2\tau) < |Y| < 2\rho_{\Pi,G}(2\tau) \) is \( O(sort_P(N)) \) in the CREW PEM model with \( P \leq N/(B^2(\log B)(\log N)(\log^t N)) \) and \( M = B^{O(1)} \).

Proof By Fomin et al. (the proof of [Fom+12a, Lemma 12]), it follows that for every \( X \in P \) with \( |X| > \rho_{\Pi,G}(2\tau) = c \) there exists a replacing \((2\tau + 1)\)-protrusion \( Y \). Thus, it is left to provide a PEM-efficient algorithm for finding \( Y \). In the following a single \( X \in P \) is considered, but note that all operations can be performed simultaneously for all \( X \in P \).

The remainder of the proof follows the structure and the notation of the proof of [Fom+12a, Lemma 12]. Let \((T,B)\) be a nice tree decomposition for \( G[X] \) that is rooted at some node \( x_0 \). This can be obtained for all \( X \in P \) in \( O(sort_P(N)) \) parallel I/Os by using the algorithm introduced in Chapter 5.

For each node \( x \) of \( T \), let \( D_T(x) \) be the set of nodes of all descendants of \( x \) (including \( x \)) with respect to the root \( x_0 \) of \( T \). Let \( B_T \) be the set of nodes \( x \) of \( T \) such that the bags of \( D_T(x) \) contain more than \( c \) vertices in total (see Figure 6.1). Let \( b \in B_T \) be such that its children are not in \( B_T \); then \( Y \) is defined (by [Fom+12a, Proof of Lemma 12]) as \( \partial_G(X) \cup \bigcup_{y \in D_T(b)} X_y \), where \( X_y \) is the bag of node \( y \). Note that \( Y \) is a \((2\tau + 1)\)-protrusion since \( \partial_G(Y) \subseteq \partial_G(X) \cup X_b \).

While it is clear that such a node \( b \) can be found in the RAM model in linear time, the remainder of this proof argues that for all \( X \in P \) a vertex \( b_X \) can be found in \( O(sort_P(N)) \) parallel I/Os.

![Figure 6.1: Depicting the sets of the protrusion replacer.](image-url)
To this end, it is shown how to compute for all nodes $x \in T$ a number $c_x$, the number of nodes in the bags of $D_T(x)$, in a bottom-up manner by evaluating a tree expression. The tree expression is defined through the nice tree decomposition $(T, B)$: thus, node $x$ of $T$ is one of the following nodes:

- A start/leaf node $x$ contains exactly one vertex and so each $x$ is initialized with vertex count value $c_x := 1$.

- An introduce node $x$ of $T$ has one vertex less than its child $y$, the number of vertices in the subtree $D_T(x)$ is not decreased, thus $c_x$ is the same as $c_y$.

- A forget node $x$ contains one vertex more than the bag of its child $y$, therefore the vertex count of $c_y$ has to be increased by one.

- A join node $x$ of $T$ with children $y, z$ contains the same vertices as $y$ and $z$. Thus, the subtree $D_T(x)$ has vertex count $c_x = c_y + c_z - |X_y|$.

The algorithm of Arge, Goodrich, and Sitchinava [AGS10], based on their list ranking algorithms, evaluates the tree expression in $\text{sort}_P(n)$ parallel I/Os.

Thus, it is easy to chose a $b \in B_T$. With an in-order enumeration of the nodes of $T$ and an Euler tour all $y \in D_T(b)$ are obtained. Thus, with sorting and elimination of the duplicate vertices, appearing in several bags, $\bigcup_{y \in D_T(b)} X_y$ can be obtained in $O(\text{sort}_P(n))$ parallel I/Os.

For returning $Y$, the set $\partial_G(X)$ has to be computed. All vertices of $X$ are colored with one color $c_X$, and all vertices of $V(G) \setminus X$ obtain another color $c_X$. The color of each vertex $v \in V(G)$ is stored at $v$, as well as the source vertex of each edge. Next, all edges of $G$ are sorted by their target (or sink) vertices. Finally, by scanning the graph $G$, for each vertex $v$ a list $L(v)$ of vertices that are colored differently from $v$ is obtained. Every vertex $v \in G \setminus X$ with non-empty neighborhood list $L(v)$ is then added to $\partial_G(X)$. This procedure takes $O(\text{sort}_P(N))$ parallel I/Os.

This works as described above, by choosing for all protrusions $X \in \mathcal{P}$ pairwise disjoint colors, as well for all protrusions in parallel.
6.2 Applying the protrusion replacer

The randomized fast protrusion replacer of Fomin et al., respectively its PEM-efficient implementation, does not yield a kernelization yet [Fom+12b]. By Fomin et al. one application, transforming $G$ into $G'$, reduces with high probability the size of $G$ by at least a constant fraction $1 > r > 0$ [Fom+12a, Theorem 10]. Thus, for obtaining a linear kernel it has to be applied $O(\log N)$ times. The linear kernelization algorithm thus follows:

**Theorem 6.6**
Let $t$ be a natural number. The expected number of parallel I/Os to compute a linear kernel for each of the problems of Lemma 6.1 and Lemma 6.2 is $O(sort_p(N))$ in the CREW PEM model if $P \leq N/(B^2(\log B)(\log^2 N)\log^t N)$ and $M = B^{O(1)}$.

**Proof** The value $\tau$ of the randomized fast protrusion replacer is chosen depending on $\kappa$ as described by Fomin et al. [Fom+12b] and Kim et al. [Kim+13], respectively.

Using the I/O-complexities given by Lemma 6.3, Lemma 6.5, and Corollary 6.4 the complexity of the algorithm is split up into a term for the case where the number of processors is bigger than the processor bound of Lemma 6.3 and a term capturing the part in which the lemma does not provide optimal bounds for the remaining graphs of size at most $x = PB^2(\log B)(\log N)\log^t N$:

$$\log \frac{N}{x} - 1 \sum_{i=0}^{\log N} sort_p(\tau^i \cdot N) + \sum_{j=\log \frac{N}{x}}^{\log N} sort_p(x) \leq O(sort_p(N)) + sort_p(x) \cdot \log N$$

The second term is in $O(sort_p(N))$, since $\frac{X}{PR} \cdot \log N$ is in $O(\frac{N}{PR})$ 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 $O(sort_p(N))$, which yields the theorem. 
\[\square\]
6.3 A permuting lower bound

Theorem 5.16 yields for \( k = 1 \) a lower bound for the Treewidth-1 Vertex Deletion problem on a planar graph for the special case that there is no vertex deletion necessary. It states that any algorithm requires \( \Omega(\text{perm}_p(N)) \) parallel I/Os in the CREW PEM model to solve Treewidth-1 Vertex Deletion on a planar graph. This problem is covered by Lemma 6.1 and Lemma 6.2. Thus, the upper bounds presented in Theorem 6.6 match the lower bounds for a large parameter range of Theorem 6.6.

6.4 Conclusion

Hagerup [Hag11] and Bevern et al. [Bev+11] argued recently 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 [KN12].

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

The results in this chapter show that there are no general obstacles to implement PEM-efficient kernelization algorithms for a large number of problems on apex-minor free graphs and topological-minor free graphs. By Lemma 6.1, PEM-efficient kernelizations for 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 6.2 there exist PEM-efficient parallel randomized kernelizations on \( H \)-topological-minor free graphs for, among others, Chordal Vertex Deletion, Interval Vertex Deletion, Edge Dominating Set, and Treewidth-\( w \) Vertex Deletion. 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.
Since those results follow from meta-results, the implied algorithms suffer from large constants hidden in the asymptotic notation. Concrete algorithms for specific problems covered by the meta-results should be implementable more efficiently in the PEM model.
CHAPTER 7

On evaluating a bilinear form in the EM model

The asymptotic worst-case complexity for computing the matrix vector product $y = Ax$ for a sparse matrix $A$ and a dense vector $x$ in the EM model is settled [Ben+10]. For an $n \times n$ matrix that has at most $kn$ non-zero matrix elements, there are needed $\Theta(\min\{\frac{kn}{B}(\log \frac{n}{kM}), kn\})$ I/O operations to compute $y = Ax$ in the EM model. Furthermore, Bender et al. prove that a randomly chosen sparse matrix is hard with probability at least $\frac{1}{2}$, in the sense that the asymptotic worst case number of I/Os is actually needed [Ben+10].

Subsequently, Greiner generalized these results to the PEM model [Gre12]. One open question, also investigated by Greiner, is the question what such a hard matrix looks like, respectively which underlying structure induces the hardness of a sparse matrix. All classes of matrices considered so far could be solved in $O(\text{scan}_P(N))$ (parallel) I/Os [Gre12].

Greiner also showed that for a sparse matrix $A$ the number of I/Os needed to evaluate a product $y' = Ax'$ is, up to a constant factor, the same as for evaluating a bilinear form $z = x^T A y$ [Gre12]. Thus, it is representative to consider the problem of bilinear form evaluation instead of the problem of computing the sparse matrix dense vector product.

Previous research shows that it is NP-complete to write a “compiler” which computes for a matrix $A$ an I/O-optimal program for an EM machine, parameterized by $M$ and $B$ [Hof97; Lie09]. The NP-hardness result exploits a tight connection between the pathwidth of a certain graph which is related to the matrix $A$. This graph of a matrix is defined as follows:
Chapter 7 On evaluating a bilinear form in the EM model

Definition 7.1 (Corresponding Graph and Matrix)
For a matrix \( A \in \{0, 1\}^{a \times b} \) the corresponding graph \( G_A = (U \cup V, E) \) is defined as follows: \( U = \{s_1, ..., s_a\} \), \( V = \{t_1, ..., t_b\} \) and

\[
E = \{(s_i, t_j) \mid a_{ij} \neq 0\}.
\]

The relationship between a matrix and its corresponding graph is visualized in Figure 7.1.

More precisely it holds, that \( G_A \) has pathwidth \( M - 1 \) if and only if there exists a program for an EM machine with \( B = 1 \) and cache size \( M \) which scans the input vectors \( x \) and \( y \) only once [Lie09, Theorem 4.9]. Since the bipartite pathwidth problem is NP-complete [Klo94], this means that the problem of writing a compiler for matrices is, too. On the other hand, the pathwidth of a graph can be computed for fixed \( M \) in linear time [Bod96; BK96]. Thus, for \( B = 1 \), a fixed \( M \), for a matrix \( A \), a program for an EM machine which accesses every vector element once, can be computed by computing a path decomposition of \( G_A \). Note that for the considerations in this chapter only the access to the vector elements is accounted, which can be motivated by several reasons [Gre12].

However, this solution is not as satisfying as one likes it to be, because of the following two reasons. First, it works only for \( B = 1 \), which is not generalized in this thesis. Secondly, it can be used to check how large the cache of a computer has to be, such that the bilinear form \( x^T Ay \) can be computed – very efficiently – by only scanning the input vectors.

![Figure 7.1: Visualization of Definition 7.1.](image-url)
One is usually interested in is the following information: given a computer with memory size $M$ and a matrix $A$ of size $s \times t$, how many I/O operations are needed to compute the bilinear form $x^T Ay$? Since a natural lower bound for the number of I/Os is reading the input vectors at least once, in the following these compulsory I/Os are not considered. Instead, here the number $\ell = j - (s + t)$ of non-compulsory I/Os is considered, where $j$ are the total I/Os needed to compute a bilinear form.

First steps in this direction have already been done. Consider for $B = 1$, fixed $M$, and fixed $\ell = j - (s + t)$ a matrix $A$ of size $s \times t$. A program that evaluates a bilinear form $x^T Ay$ in at most $\ell$ non-compulsory I/Os on the EM model can be computed in polynomial time, or it can be decided that there are needed more than $\ell$ non-compulsory I/Os to evaluate $x^T Ay$ [Lie09, Theorem 5.33]. The result uses a modification in the dynamic programming approach of the algorithm of Bodlaender and Kloks to compute a path decomposition of bounded width [BK96]. An alternative, but non-constructive, proof of this result is given in Section 7.2.

For all results in this chapter the notion of splitting a vertex is crucial. Observe the following: if a vector element $x_i$ is loaded twice, during a computation of $x^T Ay$, the intermediate results (products) of $x_i$ which eventually end in the final result are partitioned into two sets that are computed in two disjoint time intervals. In the corresponding graph this means that the vertex corresponding to $x_i$ is replaced by two vertices $a$ and $b$, and the neighborhood of $x_i$ is partitioned into two neighborhoods of $a$ and $b$. A vertex split is defined formally as follows:

**Definition 7.2**

*Splitting* a vertex $v$ of a graph $G$ yields a graph $G'$ that contains, instead of $v$, two new vertices $v_1$ and $v_2$. The split partitions the incident edges $\overline{I}(v)$ of $v$ into two (nonempty) sets $\overline{I}(v_1)$ and $\overline{I}(v_2)$. Several splits, applied one after each other, are called a *split sequence*. In the following, a formal definition of a vertex split is given.

In this definition a graph $G = (V, E, I)$ is represented by a vertex set $V$, a set of edge identifiers $E$, and a mapping function $I : E \mapsto (V)$. Furthermore, denote for a vertex $v \in V$: $\overline{I}(v) = \{e \in E \mid v \in I(e)\}$. Then a split of a vertex $v \in G$ yields a graph $G' = (V \setminus \{v\} \cup \{v_1, v_2\}, E, I')$. Furthermore, for all edges $e$ that
are not in $\overline{I}(v)$ it is true that $I(e) = I'(e)$ and for all edges $e$ in $\overline{I}(v)$ it is true that $I(e) = \{x,v\} \rightarrow I'(e) = \{x,v_1\} \lor I'(e) = \{x,v_2\}$.

Let $G'$ be a graph that is derived by a split sequence from $G$. A vertex $v$ of a graph $G'$ is called *derived* from a vertex $w \in G$ if $I'(v) \subseteq I(w)$.

A split of a vertex $v$ in a graph $G$, yielding two vertices $v_1$ and $v_2$ in $G'$, is called *simple* if either $v_1$ or $v_2$ is incident to only one edge.

Furthermore, $\tau^k(G)$ ($\tau(G)$) denotes the graph after applying the first $k$ (all) split operations of $\tau$. A sequence of split operations $\tau$ is called simple if all its split operations are simple.

The concept of splitting vertices has been used in several occasions, but especially in the area graph planarization [Nas79; FFM01; Lie01].

By using the concept of vertex splits, the question for a compiler that computes for a matrix $A$ an I/O-efficient program for computing a bilinear form can be reformulated as the following problem.

**Definition 7.3**

For the decision problem Splits to Pathwidth-$k$ (S2PW-$k$) the input is a connected graph $G$ and an integer $\ell$. An input $(G, \ell)$ is a “yes”-instance if there is a sequence of $\ell$ splits yielding a graph $G'$ with $pw(G') \leq k$.

In Section 7.1 it is shown that the problem Splits to Pathwidth-$k$ is NP-complete for almost all $k$. Furthermore, in Section 7.3 some approximation results for S2PW-$k$ with $k = 1$ are presented.

### 7.1 Vertex splitting

In this chapter it is shown that the problem Splits to Pathwidth-$k$ is NP-complete for almost all $k \in \mathbb{N}$. To this end some structural insights on vertex splits are presented.
7.1 Vertex splitting

7.1.1 Structural properties of vertex splitting

**Definition 7.4 (Diestel [Die06])**
The cycle space $C_G$ of a graph $G$ is a vector space spanned by the edge sets of all cycles of $G$. Accordingly, the cycle basis for $G$ is a basis of $C_G$.

Let $T$ be a spanning tree of $G$. Then for $e \in E(G) \setminus E(T)$, the fundamental cycle $C_e^T$ with respect to $T$ is the unique cycle in the graph $T \cup \{e\}$.

The next lemma partially states the content of [Die06, Theorem 1.9.6].

**Lemma 7.5 (Diestel [Die06, Theorem 1.9.6])**
Let $G$ be a connected graph with $m$ edges and $n$ vertices, and let $T$ be an arbitrary spanning tree of $G$. The fundamental cycles of $G$ with respect to $T$ form a cycle basis and the dimension of the (induced) cyclespace is $m - n + 1$.

Therefore, the number of cycles in a connected graph $G$ is related to the number of its vertices and edges. Observe that by Definition 7.2, a vertex split increases the size of the vertex set $V$ of a graph $G$ by one, while the size of the edge set $E$ remains the same. The following lemma shows the implications of splitting one vertex to the dimension of a cycle basis.

**Lemma 7.6**
Let $G$ be a connected graph with $m$ edges and $n$ vertices, and let $G'$ be a connected graph that is derived from a vertex split from $G$. Then the dimension of a cycle basis of $G'$ is $m - n$.

**Proof** As observed, the vertex split increases the size of the vertex set of $G$ by one. Thus, by Lemma 7.5, the dimension of the cycle base of $G'$ is $m - (n + 1) + 1 = m - n$. □

**Corollary 7.7**
Let $G$ be a connected graph with $n$ vertices and $m$ edges. The sufficient and necessary number of vertex splits $k$ to transform $G$ into a connected acyclic graph (tree) is $m - n + 1$.

**Proof** By Lemma 7.5, any spanning tree $T$ of $G$ induces a cycle space of dimension $m - n + 1$. Each of the $m - n + 1$ non-tree edges is split off by a simple
split at one of the two incident vertices. Thus, at most \( m - n + 1 \) vertex splits are needed to derive an acyclic graph \( T \).

By the previous lemma, the dimension of the cycle space is reduced by any non-disconnecting split by one. Thus, at least \( m - n + 1 \) vertex splits are needed.

Note that for a graph with more than one connected component, the cycle basis of each of the connected components forms a subspace. Therefore, the dimension of the cyclespace of a graph with \( c \) connected components is \( m - n + c \).

<table>
<thead>
<tr>
<th>Corollary 7.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A vertex split on a graph that increases the number of connected components does not change the dimension of the cycle space.</td>
</tr>
</tbody>
</table>

**Proof** Let \( G \) be a graph with \( n \) vertices, \( m \) edges, and \( c \) connected components and \( G' \) be a graph with \( c + 1 \) connected components, derived from one vertex split applied to \( G \). The dimension of the cyclespace of \( G \) is \( m - n + c \). Since the node split increases the number of vertices by one, the dimension of the cyclespace of \( G' \) is \( m - (n + 1) + c + 1 \) which is just \( m - n + c \).

### 7.1.2 NP-hardness of splitting a graph to pathwidth 1

A connected graph that has pathwidth 1 is a graph of the class of *caterpillars*. One characterization of a caterpillar is that it excludes the graph \( C_3 \) and the graph \( T_2 \), which are depicted in Figure 7.2, as a minor. In the following, the edges of a caterpillar graph are distinguished as follows: edges which are incident to a leaf, are called *dangling* edges, and the other edges are called *spinal* edges. Note that in a connected caterpillar graph, all spinal edges form a path, the *spine*.

Therefore, if \( k = 1 \), the problem S2PW-\( k \) asks for the number of splits to derive a connected caterpillar from a graph. Thus, this problem is called *Splits to Caterpillar* (S2C).

In this section, a special version of the *Hamiltonian Path* problem is reduced to the problem S2C. It is well known that the problem *Hamiltonian Path* is
NP-complete. Garey, Johnson, and Tarjan showed that even the more restricted variant of Hamiltonian Path on cubic, 3-connected, and planar graphs is NP-complete [GJT76]. Here this variant of the problem Hamiltonian Path is abbreviated with C3CPHP.

**Lemma 7.9**
The decision problem Splits to Caterpillar is NP-complete.

**Proof** Let $G = (V, E)$ be a cubic 3-connected graph with $n = |V|$ vertices and $m = |E|$ edges. The graph $f(G)$ is defined on the vertex set $V(f(G)) = V \cup E$, and for each edge $e = \{u, v\}$ of $G$ there are two edges $\{u, e\}$ and $\{e, v\}$ in $E(f(G))$.

Here it is shown that (a) $G$ has a Hamiltonian path if and only if (b) $f(G)$ can be split with $m - n + 1$ splits to a caterpillar. Note that (a) $\Rightarrow$ (b) follows easily by splitting each of the $m - n + 1$ vertices in $f(G)$ which correspond to an edge that is not part of the Hamiltonian path of $G$.

For the backward direction (b) $\Rightarrow$ (a), let $G'$ be a graph that witnesses the membership of $f(G)$ in S2C. This means, $G'$ is a caterpillar graph that is derived from $f(G)$ by $m - n + 1$ vertex splits.

Note that $G'$ is connected since otherwise by Corollary 7.8 at least one split did not reduce the dimension of the cycle space. Thus, by Lemma 7.6 the remaining $m - n$ vertex splits could reduce the dimension of the cycle space to at most 1. Therefore, the graph $G'$ would not be cycle free, which contradicts the assumption that it is a caterpillar.

Let $P$ be a longest path in $G'$. Thus, $P$ contains the spine of $G'$ and at most two dangling edges. Then all vertices of $G$ (they are also part of $G'$) are connected by $P$. Otherwise such a vertex $v$ which is not on $P$ must be incident to a dangling
edge. Furthermore, the adjacent vertex $u$ of $v$ is a vertex of degree 2 and is on the spine. Therefore, the edge $\{u, v\}$ can be added to $P$ and a longer path is obtained. Since each vertex of $P$ corresponds to a vertex in $G$ and they are connected by edges in $G$ as well, the path $P$ corresponds to a Hamiltonian path in $G$.

Note that the problem Splits to Caterpillar is in NP since it is possible to check in polynomial time if a graph is a caterpillar. □

### 7.1.3 NP-hardness of splitting a graph to pathwidth $k$ for $k \geq 3$

In this section the result from the previous section is generalized to arbitrary $k \geq 3$. The reduction starts from the same cubic, 3-connected, planar Hamiltonian path problem C3CPHP.

To prove the NP-completeness of Splits to Pathwidth-$k$ for fixed $k$, a transformation $f_k$ is defined which maps inputs of the problem C3CPHP to inputs of S2PW-$k$. Before this transformation is defined, a certain gadget is defined.

**Definition 7.10**

Let $k$ be a natural number, $G$ a graph with $n$ vertices and $m$ edges. Define the variable $z$ as $(k + 1) \cdot \binom{m - n + 1 + k}{k} + 1 + (m - n + 1)$. The gadget $H_{n,m,k}^n$ for a graph $G$ is the complete bipartite graph $K_{z,k}$, with $z$ “left” vertices called repair vertices and $k$ “right” vertices called core vertices. The parameters $n, m$ and $k$ may be omitted if they are clear from the context.

**Definition 7.11**

The transformation $f_k$ maps a cubic, 3-connected graph $G$ with $n$ vertices and $m$ edges to a graph $G'$ and a natural number $\ell$, the bound on the number of vertex splits. For each vertex $v$ of $G$ there is a gadget graph $H_{v,n,m,k}^n$ in $G'$. For every edge $e = \{u, v\} \in E(G)$ there is a vertex $e$ in $G'$. Without loss of generality, let the edges incident to a vertex in $G$ be ordered in an arbitrary but fixed fashion. Hence, let edge $e = \{u, v\}$ be the $i$-th edge incident to $v$ and the $j$-th edge incident to $u$. For each edge $e \in G$ there are two edges $\{u_i, e\}$ and $\{e, v_j\}$ in $G'$, where $u_i$ is the $i$-th core vertex of $H_{u,n,m,k}^n$ and $v_j$ is the $j$-th core vertex of $H_{v,n,m,k}^n$. The number of allowed vertex splits $\ell$ is defined as $m - n + 1$. 136
It is easy to see that the graph $f_k(G)$ is bipartite. The graph can be colored by two colors: the core vertices with the first color and the edge, and repair vertices with the second color. Note that despite the fact that most core vertices of a gadget are only connected to repair vertices, they are needed to show that splits make sense only at certain vertices. This is shown in the following two sections.

Properties of the gadget

The following properties of the gadget are exploited in the proof of NP-hardness of Splits to Pathwidth-$k$.

**Lemma 7.12**
Consider the graph $H^{n,m,k}$. Let $G'$ be a graph which can be derived from at most $m - n + 1$ vertex splits from $H^{n,m,k}$. Then $G'$ contains at least one subgraph that is a $K_{k,k+2}$.

**Proof** In this proof, we consider a scenario in which $G'$ has been derived by $m - n + 1$ vertex splits that were applied to the core vertices, and another $m - n + 1$ vertex splits that were applied to the repair vertices of the gadget $H^{n,m,k}$. This is a worst case scenario since the lemma assumes that there are applied only $m - n + 1$ vertex splits to $H^{n,m,k}$ in total.

By construction, there are $(k + 1) \cdot \binom{m-n+1+k}{k} + 1 + (m-n+1)$ repair vertices in $H^{n,m,k}$. As every vertex split can modify at most one vertex, in graph $G'$ there are at least $(k + 1) \cdot \binom{m-n+1+k}{k} + 1$ repair vertices which are not split and hence have degree $k$. Those $(k + 1) \cdot \binom{m-n+1+k}{k} + 1$ repair vertices are considered in the following together with the vertices derived from core vertices.

The initial $k$ core vertices can be split by at most $m - n + 1$ vertex splits into at most $m - n + 1 + k$ vertices.

Thus, each of the $(k + 1) \cdot \binom{m-n+1+k}{k} + 1$ repair vertices is adjacent to $k$ vertices where no vertex is derived from the same core vertex. For the number of different sets of core vertices such a repair vertex could be incident to, $(m-n+1+k)$ is a generous upper bound. Actually this number is smaller since a repair vertex cannot be incident to two vertices that are derived from the same core vertex.
Since there are at least \((k + 1) \cdot \binom{m-n+1+k}{k} + 1\) repair vertices, by the generalized pigeonhole principle, there have to be at least \(k + 2\) repair vertices that are incident to the same \(k\) vertices \(L\) that are derived from core vertices. Let these \(k + 2\) repair vertices form a set \(R\). The induced subgraph \(G'[L \cup R]\) is thus the claimed \(K_{k,k+2}\).

The following lemma states essentially that if a graph contains a \(K_{k,k+2}\), then this subgraph must be covered by any path decomposition of width \(k\) as described in the following. The path decomposition has to have \(k + 2\) contiguous bags that all contain the \(k\) “left” vertices, and each of the bags contains one of the \(k + 2\) “right” vertices.

**Lemma 7.13**

Let \(G\) be a graph that contains at least one \(H_{n,m,k}\), and let \(G'\) be a graph that is derived from \(G\) by at most \(m - n + 1\) vertex splits. If \(G'\) has a path decomposition \(P\) of width at most \(k\), then there are for every subgraph \(H = H_{n,m,k}\) at least two bags in \(P\) containing the same set of \(k\) vertices that are derived from core vertices of \(H\), and each of those bags contains one of two different vertices that are derived from repair vertices of \(H\).

**Proof**  Assume \(H\) is a gadget for which there are no such two bags in \(P\) containing the same set of \(k\) vertices derived from the core vertices of \(H\). By Lemma 7.12, the graph \(G'\) contains a graph \(K = K_{k,k+2}\) derived from \(H\).

This proof is split into two arguments: the first states that if there is a bag that contains \(k\) vertices of \(K\) that are derived from core vertices then the lemma is true. The second argument states that there cannot be a path decomposition of width \(k\) for \(G'\) which has no bag that contains all those \(k\) vertices that are derived from core vertices.

There have to be some bags which cover the edges of \(K\). Let \(C\) be the \(k\) vertices in \(K\) derived from core vertices of \(H\), and \(R\) the other vertices of \(K\). Observe the following: let \(i\) be the first bag in \(P\) that contains \(C\) and a vertex \(u\) of \(R\). Since \(pw(P) \leq k\), in the next bag a vertex has to be replaced. However, there is another vertex \(v \in R\) which also is incident to all vertices in \(C\). Since \(P\) is a path decomposition, and thus all bags containing a vertex of \(G'\) have to form a path, it is not possible to remove a vertex of \(C\) from \(i\). Thus, \(u\) has to be evicted and

138
consequently at some time the vertex \( v \) has to be in a bag with the vertices of \( C \). Therefore, in this case the lemma follows.

In the remainder of this proof it is shown that it is not possible that all edges of \( K \) are covered if no bag of \( \mathcal{P} \) contains all vertices of \( C \). Consequently, the previous case is the only possible one. Assume, there is a path decomposition \( \mathcal{P}' \) in which the maximal number of vertices of \( C \) in a bag is \( k - 1 \). If a vertex is no longer in a bag, all its neighboring vertices have to be in a previous bag since all bags containing a vertex have to form a path. Thus, if any repair vertex in \( R \) gets eventually removed from a bag, all vertices of \( C \) have been in previous bags. Especially the vertex \( v \in R \) that is removed first in \( \mathcal{P}' \) from a bag, say, \( i \). Therefore, by assumption, one of the vertices of \( C \) has been removed from a bag \( j < i \). And thus, not all neighborhoods of \( R \) can be covered, since there are too many vertices in \( R \).

\[ \square \]

**Showing NP-completeness**

**Lemma 7.14**
The problem **Splits to Pathwidth-** \( k \) is NP-complete for every fixed \( k \geq 3 \).

**Proof**  The problem is in NP since the pathwidth of a graph can be tested for every fixed \( k \) in linear time by the algorithm of Bodlaender [Bod96; BK96]. The hardness follows from Lemma 7.15 and Lemma 7.17 below.

The first part of the proof shows that a Hamiltonian path implies a path decomposition of width \( k \). This is the easy part of the proof and is a generalized version of the first part of the proof of Lemma 7.9.

**Lemma 7.15**
If a graph \( G \) is a “yes”-instance for C3CPHP, then \( f_k(G) \) is a “yes”-instance for **Splits to Pathwidth-** \( k \).

**Proof**  Since \( G \in \text{C3CPHP} \), there is a Hamiltonian path \( P = t_1, \ldots, t_n \) in \( G \). Therefore, \( P \) is a spanning tree of \( G \), too. Thus, there are \( m - n + 1 \) edges which are not in \( P \).
The following rule yields a graph $G'$ that is derived from $f_k(G)$ by $m-n+1$ vertex splits: for every $e \in E(G) \setminus E(P)$, split the corresponding vertex $e \in f_k(G)$. Furthermore, the following decomposition is a path decomposition $P$ for $G'$ of width $k$: for each gadget $H_{t_i}$, a path decomposition $P_i$ is described. Glueing all $P_i$ in the order of $P$ together yields the path decomposition $P$ for $G'$.

If $i > 1$ the first bag of $P_i$ contains the vertex $e$ of $G'$ which corresponds to the edge $e = \{t_{i-1}, t_i\}$ and the core vertex of $H_{t_i}$ that is adjacent to $e$. Otherwise it is empty. The following bags cover the edges incident to the repair vertices of $H_{t_i}$. Each bag contains all the core vertices of $H_{t_i}$ and one of the repair vertices of $H_{t_i}$ that has not been in a bag yet. There are one or two “dangling edges” that are incident to the core vertices of $H_{t_i}$ and some edge vertices that were split. They are covered by the next one or two bags that also contain all the core vertices of $H_{t_i}$. If $i < n$, there is one last bag in $P_i$ that contains $e = \{t_i, t_{i+1}\}$ and the core vertex that is adjacent to $e$.

It is easy to observe that $P$ is a path decomposition: all vertices and edges of $G'$ are covered by a bag. Furthermore, for each (core) vertex $v$, the set of bags that contain $v$ form a path. Since each of the bags contains no more than $k+1$ vertices, the pathwidth of $P$ is $k$. $\square$

The remaining part of the proof is more involved, since it is not possible to exclude the case that $G'$ does not contain cycles. The following definition helps to identify an “easy” case in which a Hamiltonian path is implied easily from $G'$.

**Definition 7.16**

Let $G$ be a graph and $G'$ be a graph that is derived from $f_k(G)$ by $m-n+1$ vertex splits. The underlying graph $u(G')$ of $G'$ is obtained by several edge contractions as follows:

Let $e$ be an edge in $G$. If there are the two (incident) edges $\{u_i, e\}$ and $\{e, v_j\}$ in $G'$ (this means the vertex $e \in V(G')$ was not involved in a vertex split), then in $u(G')$ they are contracted to the edge $\{u_i, v_j\}$. Otherwise, if the vertex $e \in G'$ was split into vertices $e_1$ and $e_2$, the edge $\{u_i, e_1\}$ ($\{e_2, v_j\}$) is contracted to vertex $u_i$ ($v_j$), respectively, the edge $\{u_i, e_1\}$ ($\{e_2, v_j\}$) and the vertex $e_1$ ($e_2$) is removed.

Let $v$ be a vertex in $G$. All edges of $H_v$ in $G'$ are contracted. Thus, each connected component of $H_v$ yields a vertex corresponding to $v$. 
Observe that $G = u(f_k(G))$.

**Lemma 7.17**

Let $G$ be a cubic, 3-connected, planar graph. If a graph $f_k(G)$ is a “yes”-instance for S2PW-$k$, then $G$ is a “yes”-instance for C3CPHP.

**Proof** Let $G$ be a graph with $n$ vertices and $m$ edges. Furthermore, let $G'$ be a graph that is derived from $f_k(G)$ by at most $m - n + 1$ vertex splits and has a path decomposition $\mathcal{P}$ of pathwidth $k$. Since $f_k(G) \in S2PW-k$, such a graph $G'$ exists.

Considering the underlying graph $u(G')$, two cases are distinguished:

1. The underlying graph $u(G')$ has $n$ vertices.
2. There is at least one $H_v$ which is split into two components. Hence, $u(G')$ has at least $n + 1$ vertices.

It turns out, that the graphs of the first type imply a Hamiltonian path, which is shown in Lemma 7.18. For every graph $G'$ of the second type, there is a graph of the first type, which can be generated from $G'$ without using more vertex splits. We show this in Lemma 7.19. This implies a Hamiltonian path in $G$ for all graphs of the second type. $\square$

In the following “easy” case, the graph $G'$, similar to the proof for pathwidth 1, implies directly a Hamiltonian path if one reverts the blow up due to the transformation $f_k$.

**Lemma 7.18**

Let $G$ be a cubic, 3-connected, planar graph and $G'$ a graph, derived from $f_k(G)$ by $m - n + 1$ vertex splits, witnessing that $f_k(G) \in S2PW-k$. If $u(G')$ has exactly $n$ vertices, then the path decomposition $\mathcal{P}$ of width $k$ for $G'$ implies a Hamiltonian path in $G$.

**Proof** In $u(G')$ there is for every vertex $v$ of $G$ a corresponding vertex which is created by contracting the edges of $H_v$. Assume that $u(G')$ is a path. Since in $u(G')$ there can only be an edge between vertices $u$ and $v$ if there is some edge between the corresponding vertices in $G$, it follows that the corresponding
vertices of \( u(G') \) define a Hamiltonian path in \( G \). Thus, in the remainder it is shown that \( u(G') \) is a path.

Since \( \mathcal{P} \) is a path decomposition, each of its bags defines a separator with at most \( k + 1 \) vertices of \( G' \). Furthermore, by Lemma 7.12, for each vertex \( v \) of \( G \) there is at least one bag containing only vertices of \( H_v \). Thus, no other vertex can be in this bag. This defines a linear order among the vertices of \( G \) and implies that \( u(G') \) is cycle free.

However, to imply a Hamiltonian path, it must hold that this linear order is connected by edges. Since there are done at most \( m - n + 1 \) vertex splits to derive \( G' \), there are at least \( n - 1 \) edges that are not split. As \( u(G') \) is cycle free, those edges must connect the gadgets.

The following lemma states that any graph \( G' \) can be transformed into a graph that admits the nice property, that it directly implies a Hamiltonian path. However, in general \( G' \) has not this nice property, since \( u(G') \) might contain a cycle. The next lemma transforms a graph \( G' \) that is derived by \( m - n + 1 \) vertex splits from \( f_k(G) \) into a graph \( G'' \) which implies a Hamiltonian path.

This transformation \( g \) uses the fact that for each \( H_v \) there is a \( K = K_{k,k+2} \) in \( G' \). This subgraph \( K \) implies some place in a path decomposition where all vertices incident to \( H_v \) can be covered. Thus, the transformation reverts splits that created for a vertex \( v \in G \) two or more vertices in \( u(G') \), and introduces splits at edge vertices \( f_k(G) \) that correspond to edges in \( G \). The lemma shows that the number of splits needed to derive \( G'' \) from \( f_k(G) \) is at most \( m - n + 1 \).

**Lemma 7.19**

Let \( G \) be a cubic, planar, connected graph, and let \( G' \) be a graph that is derived from \( f_k(G) \) by \( m - n + 1 \) vertex splits. If \( f_k(G) \in S2PW-k \) and \( u(G') \) has more than \( n \) vertices, there is a transformation \( g \) such that the underlying graph of \( G'' = g(G') \) has only \( n \) vertices and \( G'' \) has pathwidth \( k \) and can be derived by \( m - n + 1 \) vertex splits from \( f_k(G) \).

**Proof** Since there are at most \( m - n + 1 \) vertex splits, there are for each vertex \( v \in G \) at most \( m - n + 1 + 1 \) vertices \( v_i \in u(G') \) derived from vertex \( v \). The transformation \( g \) is a transformation of \( G' \) to a graph \( G'' \) which has a path decomposition \( \mathcal{P}' \) of width \( k \) and in whose underlying graph \( u(G'') \) there is for
every vertex \( v \in G \) only one vertex \( v' \in u(G'') \). Furthermore, the transformation \( g \) applies splits only at edge vertices.

Note that the inverse operation of a vertex split that led to vertices \( u \) and \( v \) is identifying \( u \) and \( v \), such that afterwards \( u = v \).

Roughly, the transformation locates for each vertex \( v \in G \) one interval \( I_v \) in the path decomposition \( P \) for \( G' \) where the whole gadget \( H_v \subseteq f_k(G) \) can be covered by \( P \). It works by splitting the parts of \( H_v \) that are not in \( I_v \) at the edge vertices and reverting the splits on vertices of \( H_v \). After describing this transformation in detail, it is proven that the transformed graph \( G'' \) can be derived by at most \( m - n + 1 \) vertex splits from \( f_k(G) \) and has still pathwidth \( k \).

By Lemma 7.12, for every vertex \( v \in G \) there is at least one vertex \( v_1 \in u(G') \) such that the vertices of \( G' \) that are contracted to \( v_1 \) contain the bipartite graph \( K = K_{k,k+2} \) as a subgraph in \( G' \). Thus, let \( X^f_{v_1} \) \( (X^l_{v_1}) \) be the first (last) bag of the path decomposition \( P \) that contains the \( k \) vertices of \( K \) that are derived from core vertices of the corresponding gadget \( H_v \) and some repair vertex of \( K \).

Precisely, the transformation \( g \) splits for each vertex \( v \in G \) its edge vertices \( e_1 \), \( e_2 \), and \( e_3 \) if they are not adjacent to a vertex that is represented by \( v_1 \in u(G') \). Furthermore, it identifies (and thus reverts any split to these vertices) all vertices of \( H_v \) that are derived from the same vertex of \( f_k(G) \). We argue that (1) \( G'' \) has pathwidth \( k \) and (2) that the number of splits used to transform \( G \) into \( G'' \) is at most \( m - n + 1 \).

1. We show that the transformation \( g \) does not increase the pathwidth by modifying the path decomposition \( P \) to a path decomposition \( P' \). By definition, there are \( k \) vertices \( C \) in \( X^f_{v_1} \) that are derived from the core vertices of \( H_v \) of \( f_k(G) \). The same vertices \( C \) are in \( X^l_{v_1} \), too (Lemma 7.13). The two bags only differ in one vertex which is derived (by definition) from a repair vertex.

Between the bags \( X^f_{v_1} \) and \( X^l_{v_1} \), for each repair vertex that is not contained in a bag between \( X^f_{v_1} \) or \( X^l_{v_1} \), a new bag is created in \( P \). Each of those bags contains \( C \) and a repair vertex, that is removed from its original bag. Note that the new vertex splits by \( g \) allow to move parts of \( G' \) that are represented in \( u(G') \) by a vertex \( v_i \), \( 2 \leq i \leq m - n + 1 \), freely in \( P \).
identifies all vertices derived from core vertices, that we assume all to be in $C$, this modified path decomposition covers all edges of $H_v$.

For any dangling edge vertex $e_1$, $e_2$, and $e_3$, that was not incident to a vertex that is contracted in $u(G')$ to $v_1$, a similar bag can be created.

Since all edges of $H_v$ that are contracted in $u(G')$ to a vertex $v_i$, with $2 \leq i \leq m - n + 1$, are contained in a bag between $X_{v_1}$ and $X_{v_1}'$, still all edges of $G''$ are covered. Thus, also all vertices are covered. Furthermore, note that all bags form a path. Therefore, $P'$ is a path decomposition of width $k$ for $G''$.

2. In the remainder it is shown that there are needed at most $m - n + 1$ vertex splits to derive $G''$ from $f_k(G)$. To this end, for every vertex $v \in G$ the number of vertex splits that is necessary to derive $G'$ from $f_k(G)$ is compared to the number of vertex splits to derive $G''$ from $G'$.

The transformation $g$ uses only a split for a gadget $H_v$ if some edge vertex that is derived from $e_1$ is not adjacent to a vertex that is represented by $v_1 \in u(G')$. However, if this situation occurs, there was applied at least one split, to split vertices of $H_v$ such that they form in $u(G')$ a different vertex from $v_1$. Since these splits are reverted by $g$, the transformation needs at most as many splits to derive $G''$ as there are needed to derive $G'$.

Therefore, the total number of splits to derive $G''$ from the graph $f_k(G)$ is at most $m - n + 1$.

Thus, there is a path decomposition $P'$ for $G''$ which has pathwidth $k$. □

### 7.2 Bilinear form evaluation with vertex splits

In this section a fixed parameter tractable (FPT) procedure for the Splits to Pathwidth-$k$ problem with fixed parameters $B = 1$, $k$ and $\ell$ is given. This implies an FPT algorithm for the problem of checking if a bilinear form $x^TAy$ can be computed on an EM machine with memory $M = k$ and $B = 1$ with at most $\ell$ non-compulsory I/Os.

For checking if a bilinear form $x^TAy$ can be computed with $\ell$ non-compulsory I/Os, there are two challenges. There have to be selected up to $\ell$ vertices that
are split. For each of these vertices the edges have to be partitioned. There are at most \( N^\ell \) different vertex multisets of size \( \ell \) that represent the vertices to be split. Even worse, a vertex may have \( O(N) \) edges and thus there can be \( O(2^N) \) partitions of the edges for each vertex which is split. Those terms are even in the world of fixed-parameter tractable algorithms too large to obtain efficient algorithms.

It turns out that the number of partitions for each vertex can be handled with the following observation. If a bilinear form \( z = x^T Ay \) can be computed with cache size \( M \) with \( \ell \) non-compulsory I/Os, then \( z \) can be computed in the EM model with cache size \( M + \ell \) without non-compulsory I/Os. Conversely, if \( pw(G_A) \) is bigger than \( M + \ell - 1 \), the number of non-compulsory I/Os needed for the computation of \( z \) in the EM model with cache size \( M \) is more than \( \ell \). On the other hand, if \( pw(G_A) \) is at most \( M + \ell - 1 \) then either case is possible.

By presenting a monadic second order logic (MSOL) formula this chapter shows with the help of Courcelle’s Theorem [Cou90] that it is possible to decide the S2PW-\( k \) problem efficiently for fixed \( k = M \) and \( \ell \). Note that in this case Courcelle’s theorem is applicable since \( pw(G_A) \) is at most \( M + \ell - 1 \). Courcelle’s Theorem states that the task of recognizing a graph structure of bounded treewidth that satisfies a MSOL formula is fixed parameter tractable.

First, a formula that is true if a graph has constant pathwidth is presented. It uses the fact that a graph of constant pathwidth \( k \) is characterized by a constant number \( \text{ex}(k) \) of excluded minors [RS04]. Therefore it suffices to give a formula minor\(_H\) that tests if a graph \( G \) contains a graph \( H \) as a minor. Note that this makes the formula highly non-constructive from a mathematical point of view [FRS87]. Furthermore, it is non-constructive from a computational point of view since \( \text{ex}(k) \) is at least \( k!^2 \) [TUK94]. Even for \( k = 2 \) it is 110 [KL94].

Monadic second order logic does not allow to quantify over a set of a binary relation. Therefore, in contrast to the standard graph representation with a unary relation for the vertices and a binary relation for the edges, here the incidence representation is used [CE12]. The incidence representation of a graph structure consists of two unary relations \( V \) and \( E \), and the binary incidence relation \( I \). While \( V \) and \( E \) contain a representative for each vertex, respectively, each edge, the incidence relation \( I \) is a subset of the binary relation \( V \times E \) indicating if a vertex and an edge are incident.
A formula \( \text{minor}_H \) which is true if a graph \( H \) is a minor of a graph structure \( G \) is not too complicated and has already been presented by Courcelle and Engelfriet [CE12]. There, the formula is presented for the standard representation of a graph structure. For completeness, here it is given for the incidence representation. The following three abbreviations are defined for readability:

- For checking if for \( x, y \in V \) the vertex \( x \) is in the neighborhood of \( y \), let \( x \in N(y) \) be a shorthand for the term \( \exists e \in E : (x, e) \in I \land (y, e) \in I \).

- For checking if a vertex \( x \in V \) is in the neighborhood of \( S \subseteq V \), let \( x \in N(S) \) be a shorthand for the term \( \exists y \in S : x \in N(y) \).

- To check that a vertex set \( S \subseteq V \) is connected, let \( \text{connected}(S) \) be a shorthand for the term \( \forall S_1, S_2 \subseteq S : S_1 = S \setminus S_2 \Rightarrow \exists x \in S_1 : x \in N(S_2) \).

Thus, the following formula is fulfilled if the graph \( H \) of size \( h \) is a minor of a graph structure \( G \) and is abbreviated by \( \text{minor}_H \):

\[
\exists V_1, \ldots, V_h \subseteq V : \left( \bigwedge_{1 \leq i \leq h} (\text{connected}(V_i) \land \exists v \in V_i) \land \bigwedge_{1 \leq i < j \leq h} (x \in V_i \Rightarrow x \notin V_j) \land \bigwedge_{(i, j) \in E(H)} (\exists a \in V_i : \exists b \in V_j : \exists e \in E : (a, e) \in I \land (b, e) \in I) \right)
\]

Therefore the formula \( \bigwedge_{1 \leq m \leq \text{ex}(M)} \neg \text{minor}_{H_m} \) evaluates to true if a graph structure \( G \) has pathwidth \( M \).

In the remainder of this section, the formula is extended such that it is true if a graph structure can be split with \( \ell \) vertex splits into a graph structure with pathwidth \( M \). Since the formula for pathwidth is modified by replacing the test if a vertex is incident to an edge, for transparency, the graph structure for the next formula is the following: there is a unary set \( E \) for the edges, a unary set \( V \), which contains \( \ell \) additional vertices, and the binary incidence relation \( \text{Inc} \). The following formula is explained in the next paragraph:
In the first line, \( \ell \) vertices are “defined” which have no neighbors. Those vertices represent the concept of adding \( \ell \) new vertices to \( V \) and thus this line can be understood as a sanity check for the input. In the second line, for each of these vertices a set \( N_i \) of edges is selected that is a subset of the edges of a vertex \( c_i \), where \( c_i \) is a vertex which is split.

For using these partitions of the neighbors, in \( \text{minor}_H \) each expression \((x, e) \in I\), with \( x \in V, e \in E \), is replaced. To distinguish if one port \( x \in V \) of an edge \( e \in E \) has been moved to one of the new vertices, the shorthand \( \text{movedPort}(x, e) \) is defined as the term \( \bigvee_{1 \leq i \leq \ell} (x = c_i \land e \in N_i) \). Thus, the following incidence expression represents the correct incidence relationship between the vertices and the (partitioned) edges, and \((x, e) \in I\) can be replaced by it in the last part of the formula:

\[
\text{movedPort}(x, e) \lor \left( (x, e) \in Inc \land \neg \left( \bigvee_{1 \leq i \leq k} c_i = x \land \text{movedPort}(z_i, e) \right) \right)
\]

Therefore, by Courcelle’s theorem \([\text{Cou90}]\), the following theorem is true.

**Theorem 7.20**

For fixed \( M \) and \( \ell \), deciding the problem if \( x^T A y \) can be evaluated on an EM machine with \( B = 1 \), cache size \( M \) and \( \ell \) non-compulsory I/Os is in \( \text{FPT} \).

The vertices that are split and the partition of their edges can be obtained with linear overhead. This information can be used for an algorithm that computes actually a path decomposition and a thus a program for a specific bilinear form \( x^T A y \).
Chapter 7 On evaluating a bilinear form in the EM model

7.3 Algorithms for split to caterpillar

The previous approach is non-constructive and suffers in practice from large constants in the asymptotic notation. Therefore, in this chapter the approximability of the Splits to Caterpillar problem is considered.

Recall that every graph of caterpillars has pathwidth 1 and is thus characterized by the two excluded minors $C_3$ and $T_2$ as depicted in Figure 7.2.

Before an approximation algorithm for the problem of splitting a graph to a graph with pathwidth 1 is given, first an optimal algorithm for splitting trees to a set of caterpillars is presented. Note that trees already exclude the $C_3$ minor.

7.3.1 Splitting a tree to caterpillars

In the following, an algorithm to split a tree into caterpillars with the minimal number of vertex splits is presented.

The input to the algorithm is a tree $T$. The tree is rooted at an arbitrary chosen vertex $r \in V(T)$. A resulting set of caterpillars for this problem is a partition of the edge set of $T$. Considering the edge set of one such caterpillar $C$, there are three types of edges distinguished: the edges which are not incident to a leaf of $C$ are called spine edges. If there is a non-spine edge $e$ of $C$ on the path from the root of $T$ to a spine edge of $C$, then $e$ is called closing edge. All other edges of $C$ are called dangling edges. The edge types of a caterpillar are visualized in Figure 7.3.

For computing an optimal split sequence to derive from $T$ a set of caterpillars, for each edge of $T$ its type is determined in a bottom-up computation on $T$. For each vertex $v$ of $T$, three minimal split configurations are computed for the case that the parent edge $e = (v, p_v)$ might be a spinal, closing or dangling edge. The type of $e$ is determined by an edge-local decision which essentially takes the minimal split configuration. However, the rule to determine the edge type is explained later in detail.

The type of an edge $e = (v, p_v)$ induces some splitting costs at a vertex $v$. For example, if $e$ is considered to be a closing edge, then $p_v$ has to be split such that only $e$ is disconnected from $p_v$. 

148
In the following these (optimal) local split costs are listed for all parent edge types depending on the type of their children edges. To this end, assume a vertex \( v \) has (determined by a not yet specified selection rule) \( \delta_v \) children edges that are dangling, \( \sigma_v \) children edges that are spinal, and \( \chi_v \) children edges that are closing edges.

Observe the following: several spines of caterpillars may contain a vertex that is derived from one vertex of \( T \). It actually does not matter how spinal edges which are incident to a vertex are connected to each other for forming caterpillars. Furthermore, dangling and closing edges at a vertex \( v \) can be attached to any spinal edge incident to \( v \) of a caterpillar without violating the definition of a caterpillar.

1. Assume that the parent edge \( e = (v, p_v) \) of \( v \) is a closing edge. Clearly, one split for splitting \( e \) off at \( p_v \) has to be accounted. This split is accounted to \( v \). Thus, all children edges which are closing do not induce further costs and can be ignored in later steps. As mentioned, dangling children edges can be attached to any spine edge, and thus do not yield further cost. Note that this also holds for the “empty” spine edge, which is just the vertex \( v \).

The \( \sigma_v \) spinal children edges induce costs whenever there are more than
two of them. In this case all pairs of spinal children edges can be “glued” to a caterpillar, each at the cost of one split at \( v \). Note that each of those splits is necessary, since otherwise a \( T_2 \) would be created. Therefore the splitting costs for spinal edges at \( v \) are \( \max[\lceil \sigma_v/2 \rceil - 1, 0] \).

This yields in the total to local splitting costs \( s_e^c \) of \( \max[\lceil \sigma_v/2 \rceil, 1] \).

2. The exactly same accounting holds for the local splitting cost \( s_e^d \) to split \( e \) to a dangling edge, too. In this case the split does not occur at \( p_v \) but directly at \( v \). Observe that this split is not needed if all children edges of \( v \) are closing edges.

Therefore, \( s_e^d \) is 0 if all children edges are closing and \( \max[\lceil \sigma_v/2 \rceil, 1] \) otherwise.

3. If \( e \) is a spinal edge, one of the children edges of \( v \) has to be a dangling or a spinal edge. If this is not the case, the local splitting costs \( s_e^s \) for making \( e \) a spinal edge are \( \infty \). By the same arguments as in the previous two cases, children edges which are dangling or closing do not induce any splitting costs.

   In contrast to the previous cases, the splitting costs for spinal children edges are \( \lceil \sigma_v/2 - 1 \rceil \) since only one spinal child edge can be “glued” by the spine edge \( e \). All other pairs have to be glued and split, and thus accounted on their own.

   Therefore, \( s_e^s \) is \( \lceil \sigma_v/2 - 1 \rceil \) or \( \infty \) as stated.

Note that the local splitting costs at a root vertex may differ by at most 1. It can be considered having a virtual dangling edge that has not to be split off.

Furthermore, observe that the choice of an edge type for an edge \( e = (v, p_v) \) induces splits at \( v \) or its incident edges and thus a certain partitioning of the edges. Actually, due to several possibilities of gluing different spinal edges (or dangling edges) to each other, there are several implied partitions possible and correct. Here, no specific partition is assumed, besides that it witnesses the local split costs of \( e \).

The total split costs for an edge \( e = (v, p_v) \) are defined as the sum of the local split costs of \( e \) and the total split costs to split all children edges of \( v \) to their specific types.
Most importantly, the local splitting costs at $v$ imply that the number of spinal children edges for $v$ should be minimized to minimize local splitting costs. Therefore, the type of an edge is selected by the following rule: an edge $e$ is only spinal, if its total split costs are strictly smaller than the total split costs for making $e$ dangling or closing. Otherwise $e$ is chosen to be dangling only if its total splitting costs for making $e$ dangling are strictly smaller than the costs of making $e$ closing. If $e$ is chosen to be neither spinal nor dangling it is closing. Note that this selection can be computed easily.

In the following an edge $e = (v, p_v)$ and its parent edge $f = (p_v, u)$ are considered as visualized in Figure 7.4. It is shown that the just mentioned rule that selects $e$ to be a certain type minimizes the total splitting costs at $f$. Since this is applicable to all children edges of $p_v$, it follows by induction that also the total splitting costs for $f$, are minimized.

Note that the rule selects (correctly) for all edges that are incident to a leaf of $T$ to be dangling at local splitting costs 0, which is smaller than 1 (closing) and $\infty$ (spinal).

Therefore, by induction it is valid to assume that the algorithm achieves minimal total splitting costs for each children edge of $v$. In order to show the optimality of the selection rule it is assumed that the above introduced rule implies a splitting $S$ of $v$ such that $e$ is chosen to be a certain edge type. This is compared to the edge types for $e$ that an optimal splitting $S^*$ may have chosen in a different way. It is shown that the implied local costs for $f$ implied by $S$ are at most the costs of $S^*$.

1. Assume that $e = (v, p_v)$ is chosen to be spinal. This is only the case if the total split costs to make $e$ spinal are strictly smaller than making it closing or dangling. Note that the implied local splitting costs for $f$ may increase by at most 1 which is at least amortized since the total splitting costs for making $e$ spinal are strictly smaller than for making $e$ closing or dangling. Thus, any different selection cannot be better.

2. Assume that $e$ is chosen to be dangling at total costs $t^d_e$. A differing optimal solution could make $e$ spinal or closing at total costs $t^s_e$, respectively $t^c_e$. 

151
Figure 7.4: The edges $e$ and $f$ that are considered for proving the optimality of the total split costs for $f$ implied selection of the edge type of $e$.

Assume that an optimal solution selects $e$ to be closing. Then the choice of selecting $e$ to be dangling cannot be worse: by the selection in $S$ it holds $t^d_e < t^c_e$, and selecting $e$ to be dangling implies local splitting costs at $f$ that are at most one larger than selecting $e$ to be closing.

On the other hand, $e$ could be selected to be spinal in $S^*$. In this case by the selection rule it holds $t^d_e \leq t^s_e$. Since a dangling edge cannot imply further local splitting costs for $f$, selecting $e$ to be dangling is at least as good as selecting $e$ to be spinal.

3. Assume that $e$ is chosen to be closing at total costs $t^c_e$. Then a differing optimal solution can select $e$ to be spinal or dangling at total costs $t^s_e$, respectively $t^d_e$. And by the selection of $S$ it holds $t^c_e \leq t^s_e$ and $t^c_e \leq t^d_e$.

As in the previous case, selecting $e$ to be spinal can, in the best case, not increase the local splitting costs for $f$. Thus $S$, in this case is at least as good as $S^*$.

On the other hand, selecting $e$ to be dangling could increase the local splitting costs of $f$ in the case that all other children edges of $p_v$ are closing. In this case the local splitting costs for $p_v$ could be $0$. Therefore, selecting $e$ to be closing can only improve the local splitting costs for $f$.

Therefore, the children type selection of $S$ is optimal.
Theorem 7.21
A tree can be split with the optimal number of vertex splits into a collection of caterpillars.

The following algorithm is a very simple approximation algorithm for the same task.

Lemma 7.22
The number of vertex splits to transform a tree $T$ with $m$ edges into a set of caterpillars is at most $\left\lfloor \frac{m+1}{4} \right\rfloor$.

Proof There is a non-optimal greedy algorithm which achieves this goal by splitting $T$ in caterpillars with at least 4 vertices:

For a tree $T$ rooted at some vertex $r$, it selects repeatedly one vertex $x$ that has the biggest distance from $r$. Let $y$ be the parent of $x$ and $z$ the parent of $y$. Then there may occur 3 cases.

1. Assume that $x$ has a sibling $s$, then splitting the edge $(y,z)$ at $z$ off yields a caterpillar which contains at least 4 vertices.

2. If $x$ has no sibling and $y$ has no sibling as well, then splitting off the parent of $z$ yields a path of length 3, that contains 4 vertices.

3. If $x$ has no sibling, but $y$ has a sibling $s$, then splitting the vertex $z$ such that a connected component which contains $x, y, s$, and a vertex derived from $z$ contains also at least 4 vertices.

All those components can be created by one split and reduce the vertex set of $T$ by at least 4 vertices. Therefore, the number of split is $\left\lfloor \frac{m+1}{4} \right\rfloor$, since the last component, containing the root, may contain a smaller number of vertices. \[\square\]

7.3.2 Approximation for dense graphs

In this section, an easy approximation algorithm for splitting a graph $G$ into a set of caterpillars is given. It applies the following two steps:

1. Split $G$ into a tree $T$. 


2. Split $T$ into caterpillars in an optimal fashion.

**Theorem 7.23**

For a graph with $2n \leq m + 1$, there exists a $\frac{3}{2}$-approximation algorithm for the Splits to Caterpillar problem.

**Proof**  
For the second step, an algorithm was presented in Section 7.3.1. For the first step, there are needed exactly $m - n + 1$ vertex splits by Corollary 7.7. Note that both steps can be computed efficiently.

For the second step there are needed at most $\left\lfloor \frac{m+1}{4} \right\rfloor$ vertex splits by Lemma 7.22. Thus, in total, there are needed at most

$$m - n + 1 + \left\lfloor \frac{m+1}{4} \right\rfloor \leq m - n + 1 + \frac{m - n + 1}{4} + \frac{n}{4} \leq \frac{5}{4}(m - n + 1) + \frac{m - n + 1}{4} = 6\left(\frac{m - n + 1}{4}\right)$$

node splits.

Since $m - n + 1$ is a natural lower bound (Corollary 7.7) for the optimal solution, the approximation factor of the algorithm is at most $\frac{3}{2}$.

Note that the previous result is true, since the number of splits to derive a tree $T$ from $G$ dominates the number of splits to make $T$ a caterpillar if $G$ is dense.

**7.3.3 The general case**

In this section it is assumed that the input graph $G$ is connected. The idea for the algorithm is the same as for “dense” graphs: Split $G$ into a tree $T$, and then split $T$ into a collection of caterpillars with the algorithm of Section 7.3.1. Lemma 7.25 proves that this algorithm yields a 2-approximation for the problem Splits to Caterpillar.

The number of splits is dominated by either the number of splits $c$ of the first step or the number of splits $t$ of the second step. In each case, the dominating number of splits is a lower bound on the number of total splits. This is clear for the number of splits $c$ to remove all cycles by Corollary 7.7. Lemma 7.25 states
complementary that the number of splits \( t \), used to turn \( T \) into a caterpillar, is inevitable. Before this can be proven, in the next section it is shown that the order in which certain split sequences are applied does not matter.

**Lemma 7.24**

Let \( \sigma \) be a split sequence and \( \tau \) be a simple split sequence for a graph \( G \). Then \( \sigma(\tau(G)) = \tau(\sigma(G)) \).

**Proof**  After the split sequence \( \sigma \) was applied to \( G \), the graph may have several vertices \( v_i \) derived from \( v \). The split sequence \( \tau \) can still be applied, since each simple split \( \tau_j \) is identified by the edge and the vertex to be split, it is clear which of the \( v_i \) has to be split by \( \tau_j \).

On the other hand, if \( \tau \) is applied to \( G \) first, each \( \tau_i \), splitting off \( e_i \) yields a vertex \( v_j \) incident to \( e_i \) and a main vertex \( v_m \) which is incident to all the other edges of \( v \) (which are not split off by other \( \tau_k \)). Thus, a split \( \sigma_{\ell} \), splitting \( v \in G \) is applied in \( \tau(G) \) to \( v_m \) respectively to \( v_{m_1} \) if \( v_m \) was split by a previous \( \sigma_x \).

We consider the edge incidence relation of the vertices \( v_i \) and \( v_{m_j} \) that are derived from \( v \) after applying \( \tau(\sigma^k(G)) \) respectively \( \sigma^k(\tau(G)) \). The lemma follows by induction over \( k \).

By induction the graphs \( \tau(\sigma^k(G)) \) and \( \sigma^k(\tau(G)) \) are equal. The following observation shows that \( \tau(\sigma^{k+1}(G)) \) and \( \sigma^{k+1}(\tau(G)) \) are equal, too. The split \( \sigma_{k+1} \) applied to \( \sigma^k(G) \) splits a vertex \( v \) into \( v_1 \) and \( v_2 \). Consider \( \sigma^k(\tau(G)) \): the split \( \sigma_{k+1} \) is applied to \( v_m \), which is incident to all the edges of \( v \) that are not split off by \( \tau \), and results in \( v_{m_1} \) and \( v_{m_2} \).

Since the edges that are split off from \( v_m \) by \( \tau \) are split off from \( v_1 \) and \( v_2 \) later on by \( \tau \), and the split \( \sigma_{k+1} \) is applied to \( v \) respectively \( v_m \) in the same way, the resulting edge incidence relation is the same.

\( \square \)

**Analysis of the 2-approximation**

**Lemma 7.25**

Consider a graph \( G \) and a tree \( T \) that is derived from a simple split sequence
τ from $G$. The optimal split sequence $\sigma$, containing $opt$ splits to turn $G$ into a collection of caterpillars, turns $T$ into a collection of caterpillars, too.

**Proof**  By applying an optimal split sequence to $G$ we derive a graph $G'$ which is already a collection of caterpillars. Since all splits in $\tau$ are simple, the splits are welldefined and thus applicable on $G'$ and yield a collection of refined caterpillars $\tau(\sigma(G))$.

By Lemma 7.24 the graphs $\tau(\sigma(G))$ and $\sigma(\tau(G))$ are equal. Thus, the split sequence $\sigma \circ \tau$ leads to the same a collection of refined caterpillars.  \( \Box \)

**Theorem 7.26**

There is a 2-approximation algorithm for the problem Splits to Pathwidth-$k$ on a connected graph $G$.

**Proof**  The algorithm consists of two steps:

1. Split $G$ optimally into a tree $T$ with $c$ splits.

2. Split $T$ with an optimal algorithm into a set of caterpillars with $t$ splits.

Since caterpillars are cycle free, by Corollary 7.7, it follows that the optimal number $opt$ of splits to derive caterpillars from $G$ is at least $c$. Thus, if $c \geq t$ the number of splits performed by the two steps is at most $2 \cdot opt$.

The following argument bounds the number of $t$ by $opt$ to make the argument work the other way round. Let $\sigma$ be an optimal split sequence to turn $G$ into caterpillars. If $t \geq opt$, by Lemma 7.25, the sequence $\sigma$ could be used for turning $T$ into a caterpillar with a smaller number of splits.

Thus, the optimal number of splits is at least the maximum of the number of splits in the first step and the number of splits in the second step:

$$\max\{c, t\} \leq opt.$$  

Assume that the number of splits to turn a tree into caterpillar graphs dominates the number of splits to turn $G$ into caterpillars. Therefore, $c \leq t \leq opt \leq t + c \leq 2t \leq 2 \cdot opt$. Thus the algorithm yields a 2-approximation.  \( \Box \)
7.4 Conclusion

Motivated by the connection between sparse matrix vector multiplication and bilinear form evaluation [Gre12], this chapter considered the complexity of splitting a graph into a graph that has pathwidth $k$. A similar problem has been considered by Hoffmann. He considered matrices which were defined by processes which had to be scheduled in an external memory setting. In his case, the corresponding graph was of bounded degree which makes the results difficult to compare [Hof97].

In the first part of this chapter it is shown that it is $\text{NP}$-complete to decide how many splits there are needed to derive a graph of pathwidth $k$. It seems plausible, that by using for the reduction a slightly different problem this holds also for the case $k = 2$.

Therefore, the remainder dealt with algorithms for the corresponding optimization problem. Besides the optimal algorithms for splitting a graph into a tree and splitting a tree into caterpillar graphs, two constant approximation algorithms were obtained to split a graph into caterpillar graphs.

However there are several open questions which are listed in the following:

- Is there, as for treewidth 1, an optimal polynomial-time algorithm to split a graph to a graph that has treewidth 2, or is this $\text{NP}$-hard?
- Are there polynomial-time approximation algorithms for the number of splits needed to split a graph into a graph that has pathwidth $k$ with $k \geq 2$ or is this $\text{NP}$-hard?
- Can non-trivial hardness results for splitting a graph into a caterpillar be achieved?
- The number $\ell$ of splits needed to create a graph that has pathwidth $k$, implies a program for the EM model with $B = 1$ that needs $\ell$ non-compulsory I/Os to compute a bilinear form $x^T A y$. It is unclear how similar results can be achieved for a larger block size $B$. 
CHAPTER 8

Conclusion

This thesis investigated the complexity of three computational tasks in the context of the parallel external memory (PEM) model. The PEM model is a theoretical model that captures the running time of many computational problems on large instances on parallel computers. It models processors that each have a fast cache of limited size in which arbitrary computations can be performed. Processors can move cache lines between their cache and a conceptually infinite shared memory by I/O operations. The complexity measure of the PEM model is the number of parallel I/O operations.

The first problem that was considered is the fundamental problem of list ranking. Despite the fact that an algorithm for list ranking in the PEM model has been introduced previously, there were no non-trivial lower bounds known for list ranking that showed its optimality in the PEM model. Furthermore, the complexity of this algorithm was proven only for certain parameter settings of the PEM model, missing especially the massively parallel case.

The second problem that was considered in this thesis is the problem of computing a tree decomposition of bounded width for a given graph. An explicitly known tree decomposition for a graph makes many (NP-)hard problems efficiently tractable. However, in contrast to many other models, for the PEM model there was previously no efficient algorithm known to compute such a decomposition.

The third central problem considered, is modifying a graph by splitting vertices such that it has bounded pathwidth. This problem is related to the problem of analyzing how many I/O operations are needed to compute a sparse matrix vector product for a certain matrix.
Chapter 8 Conclusion

8.1 Results

The previously known algorithm for list ranking has a sorting-like complexity for many parameter settings of the PEM model. This thesis showed a complementing lower bound. This lower bound exploits that the list ranking problem solves, like several other problems in the (P)EM model, a permuting problem. Since the complexity of permuting equals the complexity of sorting for most parameter settings of the PEM model, this shows optimality of the previously known algorithm in many settings.

Subsequently, an algorithm was designed that improves the parameter range of the previously known algorithm. However, it became apparent that permuting is only one part of the nature of the list ranking problem. For many parameter settings the algorithm can be shown to have essentially sorting complexity. On the other hand, this seems to be hardly possible for a parameter setting in which the complete input can be distributed into the caches of $\sqrt{N}$ processors. In this setting, the complexity seems to be quadratic in the complexity of sorting.

It turns out that an algorithm that knows the structure of the list, still can rank the list in this parameter setting in sorting complexity. Therefore the hardness of list ranking in this setting seems to stem from discovering the list structure. This was modeled in the novel Guided Interval Fusion (GIF) problem, which led to a matching lower bound for this setting. Furthermore, this lower bound implies the first non-trivial lower bound for list ranking in the MapReduce model and the bulk-synchronous-parallel model.

Considering the algorithms for computing a tree decomposition in the PRAM and the EM model, it appears that many steps of subroutines can be implemented efficiently in the PEM model by using corresponding PEM-efficient algorithms. This can be justified by the fact that many building blocks for the PRAM and the EM model are based on efficient implementations of sorting or list ranking. Besides the improved list ranking algorithm, an efficient, and for many parameter settings optimal tree decomposition algorithm for the PEM model was presented based on the following two contributions.

First, a new parallel data structure that represents “flippable DAGs” was introduced. It can also be used in the algorithm for the EM model, for which
flippable DAGs have been introduced initially. Actually, it simplifies this algorithm. Secondly, the tree contraction technique, which is well-known from the PRAM model, was shown to be implementable with slight modifications in the PEM model. This yields an alternative and sometimes simpler view on the problem of evaluating a tree expression.

Additionally, it was shown that recently established meta-results from the area of kernelization can be implemented efficiently in the PEM model with the help of the new tree decomposition algorithm. This yields asymptotically optimal kernelization algorithms for large classes of NP-hard problems for many parameter settings in the PEM model.

The results for splitting a graph such that it has bounded pathwidth exploit that the effect of a vertex split on the dimension of a cycle space is very limited. This connection shows that certain splits are inevitable to create a graph with pathwidth 1. These inevitable vertex splits are exactly the splits to turn a graph into a tree, that is a graph of treewidth 1.

Since graphs of pathwidth 1 are cycle-free, by a reduction from the Hamiltonian path problem this showed NP-completeness for the splitting problem. Furthermore, the connection to the dimension of a cycle space led to simple, yet fast approximation algorithms for this problem.

8.2 Implications and open problems

Both lower bounds for the list ranking problem come with some restrictions. The permuting lower bound uses atomicity of input elements, which is a standard assumption that is used for many lower bounds in the (P)EM model. The assumption states that an input element cannot be divided, respectively it is not possible to assemble output elements from such partial information.

On the one hand, this is not a problem since all known algorithms essentially treat input elements as atoms. On the other hand, it is possible that a novel algorithmic technique could violate the atomicity of input elements and thus a faster algorithm could break this lower bound. Therefore, a challenging open problem is to give an unconditional lower bound that implies a permuting lower bound.
A similar discussion is applicable to the lower bound that uses the GIF problem. In this case, it is not even clear which classes of algorithms are generally covered by this problem. In addition, this is a central question in further understanding the algorithmic consequences of the lower bounds implied by the GIF problem.

Another challenging part, which is probably connected to the previous question, is based on the following observation: list ranking is one central building block for solving many computational problems in the PEM model. In this thesis, computing a tree decomposition of bounded width has been shown to be one of these problems. Therefore, an open question is whether the lower bound implied by the GIF problem is applicable to these problems.

The introduced algorithm to compute a tree decomposition of width $k$ in the PEM model is, as the algorithm for the PRAM model and the algorithm for the EM model, asymptotically efficient. The asymptotic considerations hide that these algorithms are doubly exponential in $k$. This leads to intractable large constants, even for small values of $k$. However, it is a general question, which is not specific to the PEM model, whether these constants can be lowered.

A similar question considers the structure of the algorithm to compute a tree decomposition. Due to its recursive scheme, in certain parameter settings, it is by a factor $O(\log N)$ slower than list ranking. This scheme is also used in the optimal algorithms for the RAM, the PRAM, and the EM model. Therefore, it would be interesting in its own right whether there are efficient alternatives to this recursive scheme.

Moreover, the algorithms implied by the meta-results suffer from additional huge constants due to the generality of the approach. In contrast to the treewidth problem, in the RAM model there exist for several of the problems covered by the meta-results efficient kernelization algorithms. A future task is to give for the PEM model such, practically efficient, algorithms for specific problems that are covered by the meta-results. Here, this thesis showed that, in general, PEM-efficient kernelization is possible for many computational hard problems for a large range of the parameters of the PEM model.

Regarding the problem of splitting a graph such that it has bounded pathwidth, there are several open questions. Most fundamentally, all upper bounds on the complexity of this problem only imply fast algorithms for sparse matrix
multiplication if the cache line size is 1. Here, an open question asks for a more general approach.

Furthermore, the approximation algorithms currently only work for pathwidth 1. It is unclear if the constant-factor approximation can be generalized to larger values of the pathwidth, or if there also exists a hardness in approximating these problems.

Furthermore, the problem of splitting a graph such that it has treewidth 1 is easily polynomial time solvable. For larger values of treewidth neither an algorithm nor a hardness result is known. In either case, this would complement the hardness of splitting a graph such that it has pathwidth $k$.

8.3 Summary

This thesis considered three computational problems in the context of the PEM model. For the list ranking problem, the parameter range of previously known algorithms has been extended such that they cover the complete parameter range. Most interestingly, in contrast to many other problems in the context of external memory models, it does not seem to be possible to achieve a complexity that is similar to sorting for the complete parameter range. This finding has been complemented with a lower bound for the novel Guided Interval Fusion problem.

The new list ranking algorithm is used as a fundamental building block for a PEM-efficient algorithm to compute a tree decomposition of bounded width for a given graph. To this end, several techniques known for the PRAM model and for the EM model, which led to efficient algorithms in these models, were transferred to the PEM model and combined for a PEM-efficient algorithm. Subsequently, this treewidth algorithm was applied to a large class of NP-hard problems for which it has been shown that there exist asymptotically PEM-efficient kernelization algorithms.

In the context of splitting a graph such that it has bounded pathwidth, its NP-hardness has been shown. This result exploits a connection between vertex splits and the dimension of the cycle space of a graph, which implies further approximation results for the special case of pathwidth 1.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


Bibliography


Bibliography


