Construcción Paralela de Estructuras de Datos Sucintas

Tesis para optar al grado de Doctor en Ciencias de la Computación

JOSÉ SEBASTIAN FUENTES SEPÚLVEDA
CONCEPCIÓN-CHILE
2016

Profesor Guía: Leo Ferres
Dpto. de Ingeniería Informática y Ciencias de la Computación
Facultad de Ingeniería
Universidad de Concepción
PARALLEL CONSTRUCTION OF SUCCINCT DATA STRUCTURES

By

José Fuentes Sepúlveda

Advisor: Leo Ferres, PhD
Co-advisor: Meng He, PhD

Submitted in partial fulfillment of the requirements for the degree of

PH.D. IN COMPUTER SCIENCE

Departamento de Ingeniería Informática y Ciencias de la Computación

UNIVERSIDAD DE CONCEPCIÓN

Concepción, Chile
September, 2016

This work was supported the doctoral scholarship of CONICYT (21120974) and in part by the Emerging Leaders in the Americas scholarship programme
Abstract

As of 31st December, 2013, the total number of accessible Web pages amounted to 14.3 trillions and the total size of accessible data was calculated to be approximately 672EB (exabytes), not including very large private databases. It is safe to assume that the same trend will persist in the coming years: the size of the data available on the Internet will keep growing exponentially. Thus, it now has become imperative to find ways to solve the problem of reading, processing and storing those enormous amounts of data. To date, two main approaches have been proposed to solve the problem: the traditional increase in the machines’ processing power (led by clock speed up until the beginning of the millenium, and superseded by adding processors as of 2004), and the more modern, algorithm-based minimization of the space required to store data. In this thesis, we will combine both approaches, constructing succinct data structures on multicore architectures.

In particular, three succinct data structures will be addressed: wavelet trees, succinct ordinal trees and triangulated plane graphs. For wavelet trees, we present two construction algorithms that achieves $O(n)$ and $O(lg n)$ time using $O(n \lg \sigma + \sigma \lg n)$ and $O(n \lg \sigma + p\sigma \lg n / \lg \sigma)$ bits of space, respectively, where $n$ is the size of the input, $\sigma$ is the alphabet size and $p$ is the number of available threads. For succinct trees, we introduce a practical construction algorithm that takes $O(lg n)$ time and $O(n \lg n)$ bits of space for a tree on $n$ nodes. For triangulated plane graphs, we present a parallel algorithm that computes the succinct representation of a triangulated plane graph, with $n$ vertices and a canonical ordering, in $O(lg n)$ time and $O(n \lg n)$ bits of space.
Table of Contents

Abstract ii

List of Tables vi

List of Figures vii

List of Algorithms 1

Chapter 1 Introduction 2

1.1 Hypothesis 3
1.2 Goals 3
1.2.1 Main Goal 3
1.2.2 Specific Goals 3
1.3 Methodology 4

Chapter 2 Background 6

2.1 Parallel Model for Hardware 6
2.2 Parallel Model for Algorithms Analysis 8
2.2.1 Dynamic Multithreading Model 9
2.2.2 Performance Measurement 13

Chapter 3 Related Work 14

3.1 Parallel Data Structures 14
3.2 Succinct Data Structures 14
3.2.1 Wavelet Tree 15
3.2.2 Succinct ordinal trees 17
3.2.3 Triangulated plane graphs 26
3.3 Parallel Succinct Data Structures 36
3.4 Libraries of Succinct Data Structures 38

Chapter 4 Parallel Construction of Wavelet Trees 39

4.1 Parallel Construction 39
4.1.1 Parallel recursive algorithm 39
4.1.2 Per-level parallel algorithm 41
4.1.3 Domain decomposition parallel algorithm 43
4.2 Parallel Querying 47
4.3 Experiments 50
4.3.1 Experimental setup 50
4.3.2 Construction Experiments 50
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Operations supported by the NS-representation</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Execution of the sequential canonical ordering algorithm</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Datasets used in the experiments of wavelet trees</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Running times of the sequential and parallel algorithms with 1 and 64 threads</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>Throughput, last level read misses and last level write misses of the (dd^{-}\text{IQA}) and (\text{parBQA}) parallel algorithms</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>Datasets used in the experiments of succinct trees</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Running times of the algorithms (\text{libcds}, \text{sds1}, \text{and PSTA})</td>
<td>78</td>
</tr>
<tr>
<td>5.3</td>
<td>Running times of (\text{PFEA}) algorithm</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>Datasets used in the experiments of succinct maximal plane graphs</td>
<td>100</td>
</tr>
<tr>
<td>6.2</td>
<td>Running times of the (\text{PGEA}) and (\text{PSTA}) algorithms on aggregate</td>
<td>102</td>
</tr>
<tr>
<td>A.1</td>
<td>Running times of the (\text{PGEA}) algorithm by artificially increasing the workload with 16 (\text{CAS}) operations per edge</td>
<td>110</td>
</tr>
<tr>
<td>A.2</td>
<td>Running times of the (\text{PGEA}) algorithm by artificially increasing the workload with 32 (\text{CAS}) operations per edge</td>
<td>111</td>
</tr>
<tr>
<td>A.3</td>
<td>Running times of the (\text{PGEA}) algorithm by artificially increasing the workload with 128 (\text{CAS}) operations per edge</td>
<td>111</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Example of a NUMA system and a grid topology .......................... 5  
2.1 MESI protocol transitions ......................................................... 7  
2.2 Symmetric Multiprocessor System (SMP) ........................................ 8  
2.3 Example of a multithreaded computation on the Dynamic Multithreading Model (1) ................................................................. 10  
2.4 Example of a multithreaded computation on the Dynamic Multithreading Model (2) ................................................................. 11  
2.5 Diagram of the work-stealing scheduler ........................................... 12  
3.1 A wavelet tree for the sequence \( S \text{= “once upon a time a PhD student”} \) ................................................................. 16  
3.2 Balanced parentheses representation of a tree ..................................... 18  
3.3 Example of a Range min-max tree .................................................. 21  
3.4 2d-Min-Heap of the sequence \((1, 4, 9, 5, 10, 7, 3, 2, 5, 4)\) .................. 23  
3.5 Triangulated plane graph with \( n = 12, m = 30 \) and \( f = 20 \) .......... 26  
3.6 Parentheses representation \( S_{co} \) of a maximal plane graph ................. 31  
3.7 Example of a realizer ................................................................... 33  
3.8 Parentheses representation \( S_{rz} \) of a maximal plane graph ................. 34  
4.1 Snapshot of an execution of the algorithm \textit{pwt} ................................ 43  
4.2 Snapshot of an execution of the algorithm \textit{dd} ................................ 46  
4.3 Speedups of the parallel algorithms to construct wavelet trees ............... 53  
4.4 Memory consumption for the parallel construction of wavelet trees ........... 55  
4.5 Experiments in a machine with limited resources for the parallel construction of wavelet trees ....................................................... 55  
4.6 Speedup of the dataset \textit{en.4.30} encoding each symbol with 4 bytes .... 55  
4.7 Time over \( n \) with \( \sigma = 2^{14} \), 64 threads and \textit{en.14} datasets ............. 55  
4.8 Time over \( \sigma \) for the best and worst cases with \( n = 2^{30} \) and \( p = \lg \sigma \) threads .......................................................... 55  
4.9 Throughput over the number of threads for 100,000 path queries ............. 57  
4.10 Branch queries experiments over the dataset \textit{en.14.29} .................. 59  
5.1 Tree representation used as input to the \textit{PFEA} algorithm ................. 64  
5.2 Example of the proof of Lemma 1 .................................................... 69  
5.3 Computation of the ladders of a tree \( T^B \), with embedding \( B \) .................. 71  
5.4 Speedup of \textit{PSTA} algorithm compared to its sequential version ........... 79  
5.5 Speedup of \textit{PSTA} algorithm compared to the \textit{sds1} algorithm ............ 79
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Memory consumption of the algorithms <code>psta</code>, <code>libcds</code> and <code>sdsl</code></td>
</tr>
<tr>
<td>5.7</td>
<td>Speedup of the PFEA algorithm with datasets <code>ctree25</code>, <code>dna</code> and <code>prot</code></td>
</tr>
<tr>
<td>5.8</td>
<td>Speedup of the PFEA algorithm with datasets <code>ctree25</code>, <code>dna</code> and <code>prot</code>, artificially increasing the workload with 16 CAS operations per edge</td>
</tr>
<tr>
<td>5.9</td>
<td>Speedup of the PFEA algorithm with datasets <code>ctree25</code>, <code>dna</code> and <code>prot</code>, artificially increasing the workload with 32 CAS operations per edge</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of graph decompositions that do not meet the properties</td>
</tr>
<tr>
<td>6.2</td>
<td>Decomposition of a triangulated plane graph</td>
</tr>
<tr>
<td>6.3</td>
<td>Parallel computation of canonical orderings based on dual graphs and BFS traversal</td>
</tr>
<tr>
<td>6.4</td>
<td>An example of a generated dataset to test the PGEA algorithm</td>
</tr>
<tr>
<td>6.5</td>
<td>Speedup of the PGEA and PSTA algorithms</td>
</tr>
<tr>
<td>6.6</td>
<td>Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 16 CAS operations per edge</td>
</tr>
<tr>
<td>6.7</td>
<td>Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 32 CAS operations per edge</td>
</tr>
<tr>
<td>6.8</td>
<td>Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 128 CAS operations per edge</td>
</tr>
<tr>
<td>6.9</td>
<td>Running times of the PGEA and PSTA algorithms over the number of vertices, with 64 threads</td>
</tr>
<tr>
<td>6.10</td>
<td>Memory consumption sorted by the number of vertices</td>
</tr>
<tr>
<td>B.1</td>
<td>Topology of machine A</td>
</tr>
<tr>
<td>B.2</td>
<td>Topology of machine B</td>
</tr>
</tbody>
</table>

viii
## List of Algorithms

1. Example of a parallel algorithm using the `parfor` keyword ........................................ 10
2. Example of a parallel recursive algorithm using the `spawn` and `sync` keywords .......................................................... 11
3. Sequential algorithm to compute the canonical ordering ........................................... 28
4. Adjacency operation of the succinct representation of maximal plane graphs based on realizers .......................................................... 37
5. Degree operation of the succinct representation of maximal plane graphs based on realizers .......................................................... 37
6. Parallel recursive algorithm to construct wavelet trees ............................................. 40
   - Function `createNode` ......................................................................................... 41
7. Per-level parallel algorithm to construct wavelet trees .............................................. 42
8. Domain decomposition parallel algorithm to construct wavelet trees .................................................. 44
   - Function `createPartialBA` ................................................................................ 45
   - Function `mergeBA` ............................................................................................ 47
9. Parallel batch querying of range report (parBQA) ....................................................... 48
   - Function `batchRangeCount` ............................................................................. 49
10. Parallel Folklore Encoding Algorithm (PFEA) ............................................................ 65
11. Parallel Succinct Tree Algorithm (PSTA), part I ......................................................... 66
12. Parallel Succinct Tree Algorithm (PSTA), part II ......................................................... 66
13. Parallel Succinct Tree Algorithm (PSTA), part III ....................................................... 66
   - Function `concat` .............................................................................................. 66
14. Parallel canonical spanning tree algorithm (PCoST) .................................................... 85
15. Parallel graph encoding algorithm (PGEA) ................................................................. 86
16. Parallel dual graph algorithm (PDGA) ........................................................................ 94
   - Function `ownership` ......................................................................................... 95
   - Function `newIndex` ......................................................................................... 95
17. Parallel graph encoding algorithm - realizers version (PGEA-rz) ................................ 96
18. Parallel algorithm `newOrder` .................................................................................. 97
19. Parallel computation of realizers (buildRealizers) ..................................................... 99
Chapter 1

Introduction

As of 31st December, 2013, the total number of accessible Web pages amounted to 14.3 trillions, of which only 48 billions were indexed by Google (∼ 0.0034% of the total) and 14 billions were indexed by Microsoft’s Bing (∼ 0.00098% of the total) [73]. Around the same time, the total size of accessible data was calculated to be approximately 672EB (exabytes), not including very large private databases such as Facebook, Twitter, the stock exchange, human genome, among others. It is safe to assume that, barring some unexpected catastrophe, the same trend will persist in the coming years: the size of the data available on the Internet will remain growing exponentially. Thus, it now has become imperative to find ways to solve the problem of reading, processing and storing those enormous amounts of data. To date, two main approaches have been proposed to solve the problem: the traditional increase in the machines’ processing power (led by clock speed up until the beginning of the millenium, and superseded by adding processors and cores as of 2004), and the more modern, algorithm-based minimization of the space required to store data.

After their introduction in the mid-2000s, multicore computers have become pervasive. In fact, it is hard nowadays to find a single-core desktop, let alone a high-end server. The argument for multicore systems is simple [105, 120]: thermodynamic and material considerations prevent chip manufacturers from increasing clock frequencies beyond 4GHz. Since 2005, clock frequencies have stagnated at around 3.75GHz for commodity computers, and even in 2013 4GHz computers are rare. With more processing power, we can speed up algorithms that process large data and, accordingly, process more data in less time. The first approach delineated in the previous paragraph aims at taking advantage of these new multi-core architectures.

The second approach, the minimization of the space needed by data, can be further sub-divided into two categories of algorithms: those reducing the space needed to store the data and those reducing space considering some operations of interest. The algorithms in the first category reduce space by exploiting regularities in the data. This approach is known as compression. Huffman code [72] and Lempel-Ziv [127] belong to this category. Operations on the compressed data are not always possible, requiring users to decompress the data either partially or totally. The second category of algorithms use the information-theoretic minimum number of bits to represent data while supporting operations in ideally optimal time, that are of interest to the problem at hand. This approach is known as succinct data structures [76]. In general, compression techniques use less space than succinct data structures, but succinct data structures support operations directly without requiring decompression. Succinct data structures have constant or logarithmic time complexity in most of their
primitive operations. Therefore, in context where the data will be queried constantly, succinct data structures are a better choice.

A weak point of succinct data structures is their construction time, which is generally quite slow compared to other operations of the data structure. We integrate both approaches by solving the problem of construction time, using the capabilities of multicore machines. This thesis will focus on improving the design of succinct data structures over multicore architectures, obtaining good theoretical results that are also practical. In this work, practical results means results that can be implemented in commodity architectures, results that scale on time over the number of cores on a machine and results where their implementations use memory space accordingly with the theoretical results. Improving the construction time using multicore architectures allows us to design succinct data structures with competitive querying time, efficient space usage and fast/scalable construction time.

This thesis is organized as follows: In Chapter 2 we talk about the background needed to understand the remaining chapters. In Chapter 3 we discuss works related to this project and explain the sequential versions of all the data structures that we will construct in parallel. In Chapters 4 to 6 we present our parallel algorithms: In Chapter 4 we present our parallel algorithms to construct wavelet trees. Chapter 5 discusses the parallel construction of succinct ordinal trees. The parallel construction of succinct triangulated plane graphs is shown in Chapter 6. In Chapter 7, we discuss open problems and future work. Finally, in Chapter 8 we present our conclusions.

1.1 Hypothesis

The thesis will be based on the following hypothesis: It is possible to design practical succinct data structures on multicore machines. As we said before, a succinct data structure is practical if it can be implemented in commodity architectures, can scale on time over the number of cores on a machine and its implementation uses memory space accordingly with the theoretical results.

1.2 Goals

1.2.1 Main Goal

The main goal of this thesis is to design and implement a subset of relevant succinct data structures, in particular, wavelet trees, succinct ordinal trees and triangulated plane graphs.

1.2.2 Specific Goals

1. To design construction and querying algorithms on multicore machines to wavelet trees, succinct ordinal trees and succinct planar graphs.
2. To analyze the bounds of the designed multicore algorithms in the dynamic multithreading model.

3. To make practical implementations of the algorithms on multicore machines.

4. To evaluate the performance of these multicore data structures on several dimensions including in time, memory usage, and memory transfers using both synthetic and real world data.

1.3 Methodology

- All succinct data structures will be designed to work on SMP systems. Thus, we will work with the restriction of a relatively small number of cores, compared to the input size (i.e., \( p \ll n \), where \( p \) is the number of cores and \( n \) is the input size). We will focus more in the construction algorithm, since previously, very few parallel algorithms have been designed to construct succinct data structures.

- Each succinct data structure will be analyzed using the DYM model and its metrics: work, span, speedup and parallelism. Furthermore, in this thesis, we will report two different speedups. The first speedup: \( T_1/T_p \), where \( T_1 \) is the time of the proposed construction algorithm running with one thread and \( T_p \) is the time of the proposed construction algorithm running with \( p \) threads. The second speedup: \( T_{best}/T_p \), where \( T_{best} \) is the best sequential time of the baselines and \( T_p \) is the time of the proposed construction algorithm running with \( p \) threads.

- All proposed data structures in this thesis will be implemented in C and compiled with the GCC Cilk branch. This branch was selected because it implements the complete DYM specification as of the time of writing.

- In the evaluation of the data structures, mainly their construction time, we will sample input taken from real world corpora whenever possible. For each data structure, we will obtain two main metrics: time and memory usage. The ultimate goal is that construction time scales linearly with the number of threads, and that the memory usage is competitive with their sequential counterparts, even when increasing the number of threads. As a secondary metric, memory transfers (measured by the number of cache misses at the last level), will sometimes be used to explain some scalability effects of parallel construction time. Regarding queries, we will only consider the time, but not the memory usage.

- Reproducibility of the results obtained in this work is an integral part of this dissertation. As such, all implementations, corpora and results are available at http://thesis.josefuentes.cl

All implementations will be tested in two multicore machines. The description of each machine is below:
Machine A: This machine implements the Westmere microarchitecture. The machine has a dual-processor Intel® Xeon® CPU (E5645) with six cores per processor, for a total of 12 physical cores running at 2.40GHz. Hyperthreading is disabled. The computer runs Linux 3.5.0-17-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 32KB and 256KB, respectively and a per-processor shared L3 cache of 12MB, with a 5,958MB (~6GB) DDR3 RAM memory, clocked at 1333MHz. All caches levels are inclusive.

Machine B: This machine implements the Bulldozer microarchitecture. The machine has a quad-processor AMD Opteron™ Processor 6278 with 16 cores per processor\(^{1}\), for a total of 64 physical cores running at 2.40GHz. The computer runs Linux 3.11.0-26-generic, in 64-bit mode. This machine has a per-core L1 and L2 caches of sizes 64KB and 2048KB, respectively, and a per-processor shared L3 cache of 6MB, with a 189GB DDR3 RAM memory, clocked at 1333MHz. Caches L1 and L2 are inclusive. L3 is neither inclusive nor strictly exclusive of the L2 caches.

Both machines are Non Uniform Memory Access (NUMA) systems. In a NUMA system, each processor has a local memory and can access the local memory of other processors (remote memory) through a dedicated wiring\(^{2}\). To manage the memory access, each processor has an integrated memory controller. As usual, memory accesses to local memory are cheaper than memory accesses to remote memory. In a NUMA system, all the processors are connected in a grid topology [33], which increases the memory bandwidth, since all processors can access to memory independently. See Figure 1.1 for an example of a NUMA system with four processors.

For a performance comparison of Intel Sandy Bridge microarchitecture (the successor of Westmere microarchitecture) and AMD Bulldozer microarchitecture, please, see [94].

In Appendix B we show the memory hierarchy of each machine.

---

\(^{1}\)Each processor has 8 dual-cores. Each dual-core shares the instruction fetch and decode units, floating point unit, L1 instruction cache and the L2 cache. In this thesis, for our experiments, we consider that each processor has 16 cores.

\(^{2}\)In Intel processors, the dedicated wiring is called QuickPath Interconnect Technology. In AMD processors, the dedicated wiring is called HyperTransport Technology.
Chapter 2

Background

In this section, we discuss the parallel models that we will use in this thesis and how we will measure performance. With respect to models, it is necessary to distinguish between models for hardware and models for algorithm analyses. Models for hardware specify how the memory and the processing units are organized, while models for algorithms define how to analyze the complexities of the algorithms, based on running time, resource usage, among others. Both kind of models complement each other. They are described below.

2.1 Parallel Model for Hardware

This thesis assumes the Symmetric Multiprocessor System (SMP) [119]. An SMP system consists of a collection of homogeneous processing units which share a common physical memory called Main Memory. Each processing unit works independently with respect to the other processing units and all of them take equally long to access main memory. In order to access main memory, processing units use the Front Side Bus (FSB), shared by all of them. If two or more processing units try to use the FSB at the same time, a unit called the Bus Master randomly selects one of them to access the FSB. Generally, each processing unit has a high-speed memory called a cache to improve the locality of data. Among processors, caches and main memory, the unit of transfer is the cache line. A cache line can be in more than one cache, at the same time. Synchronization among processing units is done through Main Memory. SMP systems also assume that there exist just one operating system and all read/write operations to main memory are atomic. Figure 2.2 shows how a SMP system looks like as a diagram. A further assumption in SMP systems is that all processing units see the same memory content at any time. To ensure that uniform view of the memory, a cache coherency protocol is considered. In this thesis, we assume the Modified-Exclusive-Shared-Invalid (MESI) cache coherency protocol. The MESI protocol assumes the following four states for each cache line:

- **Modified**: The cache line has been modified for the local processing unit. This state implies that there is only one copy of the cache line.
- **Exclusive**: The cache line is not modified and there is only one copy of it.
- **Shared**: The cache line is not modified and might exist more than one copy of it in different caches.
- **Invalid**: The cache line is invalid or unused.
Figure 2.1 shows the transitions among states. Initially, all cache lines are empty and their state is Invalid. If a cache line is loaded to be written, then the cache line changes to Modified. If the cache line is loaded to be read, then its state changes to Exclusive or Shared, depending on if another processing unit has a copy of the cache line.

If a remote processing unit wants to read a Modified cache line, then the local processing unit sends the cache line to the remote processing unit and both processing units, the local and the remote, change its copy of the cache line to Shared. If a remote processing unit wants to write a Modified cache line, then the local unit sends the cache line and changes the state of its copy to Invalid.

If a Shared cache line is locally written, then its state changes to Modified and all other possible copies change their states to Invalid. If a remote processing unit wants to write a Shared cache line, then the state of the cache line changes to Invalid. The case of Exclusive cache lines is similar to Shared cache lines with only one difference: A local write does not generate changes in the states of other copies of the cache line. Any other transition does not change the state of the cache line.

The choice of the SMP system hardware model was based on two considerations: first, SMP systems reflect the parallel architectures that are implemented most commonly today in commodity computers; that is, computers that are readily available in the mass market. Currently, also, it is common to find large clusters that consist of several SMP-like computers—more than one core, shared memory and one operating system, making the findings effectively applicable to some distributed memory architectures as well. Second, the model of computation that will be used in this thesis (see Section 2.2) assumes an ideal parallel computer which consists of a set of cores and sequentially consistent shared memory [28]. SMP systems match that definition of the ideal parallel computer.
2.2 Parallel Model for Algorithms Analysis

The lack of a standard model for parallel computation is evident when reviewing the literature. Since the 1970’s until now the number of parallel models has been increasing, making it complicated to compare algorithmic solutions.

One of the most used parallel models is PRAM [41, 52, 56]. The amount of research over this model is impressive, even more so considering the absence of real machines implementing this model’s assumptions. Even though adaptations of solutions on PRAM can be considered practical today [6], the big picture is that PRAM model is not a practical model. PRAM model assumes an unbounded set of cores, an unbounded global memory and an unbounded local memory for each core which is not practical. For example, [63] exposed a parallel algorithm to construct canonical orderings on $O(\log n)$ time with $O(n^2)$ cores in CREW PRAM, where $n$ is the number of nodes of a triconnected planar graph and CREW PRAM is a version of the general model PRAM, considering concurrent reads and exclusive writes. Evidently, by today’s standards, the use of $n^2$ cores to compute a canonical ordering of a planar graph with $n$ nodes is not practical, especially when it comes to big graphs. Additionally, most PRAM algorithms assume SIMD machines (single instruction, multiple data), which do not take advantage of the MIMD architectures (multiple instruction, multiple data) that are prevalent today.

Since the approach of this thesis is to provide practical solutions, it is necessary to adopt models other than PRAM. Although the choice of models is wide-ranging [123, 10, 13], the Dynamic Multithreading Model (DYM, for short) proposed in [28] stands out for its influence on practical concurrency platforms, such as Intel’s CilkPlus\footnote{Intel’s CilkPlus: \url{www.cilkplus.org}}.
OpenMP\textsuperscript{2}, and Threading Building Blocks\textsuperscript{3}. A comparison of some of those platforms can be found in [107].

DYM specifies the \textit{logical parallelism} of an algorithm, using a reduced set of keywords: \texttt{spawn}, \texttt{sync}, and \texttt{parfor}. It is possible to obtain a serialization of a parallel algorithm in DYM by just deleting the keywords \texttt{spawn} and \texttt{sync}, and replacing the keyword \texttt{parfor} with the traditional keyword \texttt{for}. Thanks to that, DYM allows us to measure the parallelism of an algorithm in a clear and simple way, with the added characteristic that all correct parallel programs also imply correct sequential ones. Another feature of DYM is that it considers the usage of a work-stealing scheduler, which simplifies algorithm design, making the mapping of the parallel computation onto cores transparent to the programmer. DYM will be the model for parallel computation used in this thesis. The next section describes DYM in more detail.

\subsection*{2.2.1 Dynamic Multithreading Model}

As previously stated, DYM works by declaring the logical parallelism of an algorithm using the keywords \texttt{spawn}, \texttt{sync}, and \texttt{parfor}. In other words, \texttt{spawn}, \texttt{sync}, and \texttt{parfor} indicate which parts of the computation \textit{may} proceed in parallel. The \texttt{spawn} keyword signals that the procedure call that it precedes may be executed in parallel with the next instruction in the instance that executes the \texttt{spawn}. In turn, the \texttt{sync} keyword signals that all spawned procedures must finish before proceeding with the next instruction in the stream. Finally, \texttt{parfor} is “syntactic sugar” for \texttt{spawn}ing one thread per iteration in the for loop, thereby allowing these iterations to run in parallel, followed by a \texttt{sync} operation that waits for all iterations to complete. In practice, the \texttt{parfor} keyword is implemented by halving the range of loop iterations, \texttt{spawn}ing one half and using the current procedure to process the other half recursively until reaching one iteration per range. After that, the iterations are executed in parallel. This implementation adds an overhead to the parallel algorithm bounded above by the logarithm of the number of loop iterations. If a stream of instructions does not contain one of the above keywords, or a \texttt{return} (which implicitly \texttt{syncs}) from a procedure, they form what is called a \textit{strand}.

For all our parallel algorithms, their sequential versions can be obtained by replacing \texttt{parfor} instructions with sequential \texttt{for} instructions.

In DYM, a multithreaded computation can be seen as a directed acyclic graph (DAG) $G = (V,E)$, where the set of vertices $V$ are instructions and $(u,v) \in E$ are dependencies between instructions; in this case, $u$ must be executed before $v$. The possibility of seeing a multithreaded computation as a DAG allows us to obtain an estimation $T_p$, the time needed to execute the computation on $p$ cores. $T_p$ depends on two parameters of the computation: its \textit{work} $T_1$ and its \textit{span} $T_\infty$. For simplicity, let’s assume that each strand takes unit time (observe that we can decompose a strand that takes more than unit time into several strands that take unit time). The \textit{work}

\footnote{OpenMP: \url{www.openmp.org}} \footnote{Threading Building Blocks: \url{www.threadingbuildingblocks.org}}
Algorithm 1: Example of a parallel algorithm using the `parfor` keyword. In parallel, the algorithm initializes all the elements of the array `A` with 0.

```
A : array of 8 numbers
parfor i = 0 to 7 do
    A[i] = 0
return
```

Figure 2.3: Example of a multithreaded computation on the Dynamic Multithreading Model. It corresponds to the directed acyclic graph representation of Algorithm 1. Vertices represent strands and edges represent dependences.

is the total running time taken by all strands when executing on a single core. In other words, it is the number of vertices of the DAG\(^4\). Since \(p\) cores can execute only \(p\) instructions at a time, we have \(T_p = \Omega(T_1/p)\). The span, \(T_\infty\), is the critical path (the longest path) of the DAG. Since the instructions on this path need to be executed in order, we also have \(T_p = \Omega(T_\infty)\). Together, these two lower bounds give \(T_p = \Omega(T_\infty + T_1/p)\). In the study of parallel algorithms, the speedup is a metric that measures how a parallel algorithm scales with respect to a sequential one. In DYM, the speedup of a computation is defined as the ratio \(T_1/T_p\), where linear speedup, \(T_1/T_p = \Theta(p)\), is ideal. In DYM, we also compute the parallelism, which correspond to the ratio \(T_1/T_\infty\) and can be interpreted as the theoretical maximum number of cores for which it is possible to achieve linear speedup. Finally, the efficiency is defined as \(T_1/(T_p \times p)\), the ratio between the speedup and the number of cores, and can be interpreted in three ways: as the percentage of improvement achieved by using \(p\) cores, as the speedup per core, or how close we are of the linear speedup.

Algorithms 1 and 2, and Figures 2.3 and 2.4 give two examples of the usage of DYM. In the figures, each circle represents one strand and each rounded rectangle represents strands that belong to the same procedure call. Algorithm 1 represents a parallel algorithm using `parfor` and Figure 2.3 shows its multithreaded computation. The algorithm starts on the initial procedure call with the entire range \([0, 7]\). The first half of the range is spawned (black circle in the initial call) and the second half is processed by the same procedure (gray circle of the initial call). This divide-and-conquer strategy is repeated until reaching strands with one iteration of the loop (black circles on the bottom of the figure). Once an iteration is finished, the corresponding strand syncs to its calling procedure (white circles), until reaching the final strand (white circle of the initial call). Assuming that each strand takes unit

\(^4\)Note that analyzing the work amounts to finding the running time of the serial algorithm using the RAM model.
Input: $A, v, s, e$

1  $c = 0$
2  if $e - s = 1$ then
3   if $A[s] = v$ then return 1
4  return 0
5  $m = \lfloor (s + e)/2 \rfloor$
6  $a =$ spawn $pcount(A, v, s, m)$
7  $b = pcount(A, v, m + 1, e)$
8  sync
9  return $a + b$

**Algorithm 2: $pcount()$.** Example of a parallel recursive algorithm using the spawn and sync keywords. In parallel, the algorithm counts the occurrences of the element $v$ between the $s$-th and $e$-th elements of the subarray $A$.

**Figure 2.4:** Example of a multithreaded computation on the Dynamic Multithreading Model. It corresponds to the directed acyclic graph representation of the call $pcount(A, v, 0, 6)$ of Algorithm 2. Vertices represent strands and edges represent dependences.

time, the work is 29 time units and the span is 8 time units (this is represented in the figure by the nodes connected with shaded edges).

The case of Algorithm 2 is similar, but using spawn and sync. Figure 2.4 shows the corresponding multithreaded computation. Let $A[s, e]$ be the subarray with elements $A[s], A[s+1], \ldots, A[e]$. The algorithm starts on the initial procedure call with the subarray $A[0, 6]$. The first half of the subarray is spawned (black circle in the initial call) and the second half is processed by the same procedure (gray circle of the initial call). This divide-and-conquer strategy is repeated until reaching strands with one element of the array $A$ (black circles on the bottom of the figure, where $s$ is equal to $e$). Once a bottom strand is finished, it syncs to its calling procedure (white circles), until reaching the final strand (white circle of the initial call). Assuming that each strand takes unit time, the work is 25 time units and the span is 8 time units.

The DYM takes advantage of greedy schedulers [11, 28, 12] to schedule the strands efficiently onto cores of an ideal parallel computer. Greedy schedulers assign as many strands to cores as possible at each step; i.e., if the parallel computer has $p$ available cores and at least $p$ strands are ready to be executed, then $p$ strands are executed; otherwise, if less than $p$ strands are ready, all of them are executed. Such kind of schedulers have been proven to achieve at least half of the optimal performance. More precisely, for any multithreaded computation with work $T_1$ and span $T_\infty$, and for any number $p$ of cores, any greedy scheduler $\chi$ achieves $T_{\chi,p} \leq T_1/p + T_\infty$, where $T_{\chi,p}$ is the minimal execution time of the multithreaded computation using $p$ cores with the scheduler $\chi$. In particular, in this thesis we will be working with the work-stealing scheduler [2, 12, 86], which is the most used greedy scheduler in practical concurrency platforms. To simplify the notation, instead of $T_{\chi,p}$, we will
use $T_p$. The work-stealing scheduler uses the **work-stealing algorithm** to compute a schedule. Figure 2.5 shows a diagram of a work-stealing scheduler and illustrates the work-stealing algorithm. In the work-stealing algorithm, strands are also called **tasks** or **lightweight-threads**; strands are distributed on **threads** (also called **static threads**), and threads are mapped to physical cores. Each thread maintains a pool of **ready strands** from which it obtains work. The pool is maintained as a double-ended queue or deque. If the pool of a thread is empty, this thread (called the **thief**) tries to steal a strand from another thread (called the **victim**). If the steal is successful, the thief executes the stolen strand; otherwise, the thief chooses another victim and tries to steal again. To steal a strand, the thief has to try to steal from the top of the corresponding pool/deque. If the pool is not empty, the thread continues executing a strand from the bottom of its pool (working effectively as a stack). Each time that a thread creates a new strand (through **spawn** or **parfor**), such a strand is pushed in the bottom of the pool associated to that thread. This strategy balances the workload of the threads. The work-stealing scheduler simply schedules strands onto threads and assumes that the operating system schedules the threads on the physical cores. The incurred overhead by the operating system scheduler is not considered part of the work-stealing scheduler (and is in fact referred to as an “adversary” in parts of the literature). However, in this thesis we have considered the same amount of static threads and cores, reducing the probability of **context switching** among cores. In that way, the overhead of the operating system scheduler is negligible, or at least controlled. For more details about the work-stealing scheduler and its implementation, please review these works [12, 2, 86, 27].

**Figure 2.5:** Diagram of the **work-stealing scheduler**. The strands are symbolized by light-blue squares, pools by gray rectangles, cores by green circumferences and threads by orange arrows. The thread associated with the core 3 must steal a strand from another randomly selected thread, since this is the only option of this thread to obtain work on the work-stealing scheduler.

Finally, with respect to the memory usage of the work-stealing scheduler, Blumofe
and Leiserson [2, 12] show that the scheduler exhibits at most a linear expansion of space, that is, $O(S_1 p)$, where $S_1$ is the minimum amount of space used by the scheduler for any execution of a multithreaded computation using one core. This upper bound is optimal to within a constant factor [2]. In this thesis we will assume that the number of threads will be much lower than the size of the succinct data structures. Therefore, the space used by the scheduler can be considered negligible. In that way, in this thesis we will prove that our succinct data structures use the memory efficiently.

### 2.2.2 Performance Measurement

Considering practical implications of the succinct data structures studied in this thesis, it is necessary to consider additional tools besides the models. For each succinct data structure we obtain three measurements: time, memory usage, and cache misses. Time allows us to measure the impact of adding more processing units in the creation of the data structures. Memory usage is important to see if including more processing units involves more memory usage. Finally, cache misses is an additional measurement that allows us to detect if the use of more processing units improves/deteriorates the cache behavior. To measure the time we will use the high-resolution (nanosecond) C function `clock_gettime`. The function was set with the timer `CLOCK_THREAD_CPUTIME_ID` to measure the time of the main thread. Memory usage will be measured by counting the explicit memory allocations, through `malloc` in C or `new` in C++. To do that, we will use a tool called `malloc_count` [9]. On the other hand, `perf` will be used to obtain the cache misses.

---

5We used the C function `clock_gettime()` to ensure that the function `clock_gettime` has a nanosecond resolution.
Chapter 3

Related Work

3.1 Parallel Data Structures

Little work has been done to date on parallel succinct data structures, even in PRAM or SMP systems. With respect to non-succinct data structures, such as linked data structures, i.e., data structures which consist of a set of data nodes linked together and organized by references, the research has been more plenty on multicore architectures [1, 31, 35, 60, 69, 71, 79]. In general, the research on this kind of data structures is based on the synchronization of threads through synchronization primitives, allowing each thread to manipulate the data structure in a concurrent way. Such synchronization primitives may belong in two categories: Blocking or Non-blocking. Blocking primitives, such lock, lock all except one thread, ensuring that just one thread can manipulate the data structure. Once the thread finishes, the rest of the threads can continue using the data structure. Depending on the data structure, the primitives can be used to lock the entire structure or just a portion of it. As the counterpart, non-blocking primitives, such as compare-and-swap (CAS), do not use locks. Instead, these primitives make small changes that involve few machine instructions. Those machine instructions are applied atomically, preventing two threads from interfering each other. As these primitives make small changes, the overhead generated by the scheduling of the threads is low. Both kind of primitives are available in current architectures, being part of the main programming languages.

Solutions to parallel non-succinct data structures must be reviewed, evaluating the chance to apply them in parallel versions of succinct data structures. In particular, concepts such as linearizability [70], a correctness condition for data structures shared by concurrent processes, and wait-free/lock-free synchronization [68, 66] may be useful in the context of dynamic succinct data structures, where the succinct structure will be modified concurrently. Even though, dynamism in succinct structures is not the objective of this thesis, we will discuss how synchronization primitives may be used on dynamic versions of succinct data structures.

3.2 Succinct Data Structures

A succinct data structure is an asymptotically space-efficient and querying-time-efficient representation of a data structure [76]. Space-efficient means that the space used by the succinct data structure is close to the information-theoric lower bound of that data structure. In particular, let $lwr$ be the information-theoric lower bound, a succinct data structure uses $lwr + o(lwr)$ bits. Querying-time-efficient means that the
optimality reached by other non-succinct data structures on querying time is achieved by succinct representations. For example, consider the representation of a binary tree of \( n \) nodes. Let’s consider first a traditional linked representation, where each node has references to its left-child, right-child and parent. At \( \lg n \) bits per reference, the traditional representation uses \( n \lg n \) bits. Now, let’s consider a succinct representation. Since there exist only \( C = \binom{2n+1}{n}/2n+1 \) different binary trees, then \( \text{lwr} = \lg C \), which is fewer than \( 2n \) bits. Therefore, a succinct representation for a binary tree should use \( 2n + o(n) \) bits while still supporting operations in optimal time.

The research on succinct data structures has been broad, including succinct representation for text indexes\[49, 103, 25, 89, 58, 14, 55\], trees\[102, 75, 112, 96, 8, 50, 78, 88, 61, 37\], graphs\[75, 19, 7, 18\], among others\[118, 16, 92, 106, 99\]. Those structures also shows a good behavior in practice \[14, 3\].

In this thesis we will work on three succinct data structures: *wavelet trees* (henceforth \( \text{wtree} \)), one of the most used succinct data structures in text indexing, *unlabeled succinct ordinal trees* (henceforth \( \text{stree} \)), which is the base of the succinct representation of other structures such as labeled succinct trees and planar graphs \[7, 75\] and *succinct triangulated plane graphs* (henceforth \( \text{sgraph} \)), which have applications in representing geographic maps, microchip layouts, among others \[115, 101, 44, 80\].

### 3.2.1 Wavelet Tree

The \( \text{wtree} \) was introduced the first time in \[57\]. Although the \( \text{wtree} \) was original devised as a data structures for encoding a reordering of the elements of a sequence\[57, 38\], it has been successfully used in many applications. For example, it has been used to index documents \[124\], grids \[104\] and even sets of rectangles \[16, 17\], to name a few applications (w.r.t. \[103, 90\] for comprehensive surveys).

For the purpose of this thesis, a \( \text{wtree} \) is a data structure that maintains a sequence of \( n \) symbols \( S = s_1, s_2, \ldots, s_n \) over an alphabet \( \Sigma = [1..\sigma] \) under the following operations: \text{access}(S, i), which returns the symbol at position \( i \) in \( S \); \text{rank}_c(S, i), which counts the times symbol \( c \) appears up to position \( i \) in \( S \); and \text{select}_c(S, j), which returns the position in \( S \) of the \( j \)-th appearance of symbol \( c \). \( \text{wtrees} \) can be stored in space bounded by different measures of the entropy of the underlying data, thus enabling compression. In addition, they can be implemented efficiently \[24\] and perform well in practice.

The \( \text{wtree} \) is a balanced binary tree. We identify the two children of a node as left and right. Each node represents a range \( R \subseteq [1, \sigma] \) of the alphabet \( \Sigma \), its left child represents a subset \( R_l \), which corresponds to the first half of \( R \), and its right child a subset \( R_r \), which corresponds to the second half. Every node virtually represents a subsequence \( S' \) of \( S \) composed of symbols whose value lies in \( R \). This subsequence is stored as a bitmap in which a 0 bit means that position \( i \) belongs to \( R_l \) and a 1 bit means that it belongs to \( R_r \). In this work, we focus on \( \text{wtree} \) where the symbols of \( \Sigma \) are contiguous in \([1, \sigma]\). If they are not contiguous, a bitmap is used to remap the sequence to a contiguous alphabet \[24\]. Under these restrictions, the \( \text{wtree} \) is a
Figure 3.1: A wtree for the sequence \( S = \text{“once upon a time a PhD student”} \) and the contiguous alphabet \( \Sigma = \{o,n,c,e,’ ’,u,p,a,t,i,m,P,h,D,s,d\} \). We draw spaces using stars.

(a) Representation of a wtree using one pointer per node and its associated bitmap. The subsequences of \( S \) in the nodes (gray font) and the subsets of \( \Sigma \) in the edges are drawn for illustration purposes.

(b) Representation of a wtree using one pointer per level and its associated \( n \)-bit bitmap. It can simulate the navigation on the tree by using the rank operation over the bitmaps.

In its simplest form, this structure requires \( n \lceil \lg \sigma \rceil + o(n \lg \sigma) \) bits for the data, plus \( O(\sigma \lg n) \) bits to store the topology of the tree (considering one pointer per node), and supports aforementioned queries in \( O(\lg \sigma) \) time by traversing the tree using \( O(1) \)-time rank/select operations on bitmaps [109]. A simple recursive construction algorithm takes \( O(n \lg \sigma) \) time (we do not consider space-efficient construction algorithms [26, 121]). As mentioned before, the space required by the structure can be reduced: the data can be compressed and stored in space bounded by its entropy (via compressed encodings of bitmaps and modifications on the shape of the tree), and the \( O(\sigma \lg n) \) bits of the topology can be removed, effectively using one pointer per level of the tree, [24], which is important for large alphabets. Unless otherwise stated, we will focus on construction using a pointer per level because, even though it adds some running time costs, it is more suitable for big data. Notwithstanding this, it is trivial to apply the technique to the one-pointer-per-node construction case, and our results can be readily extended to other encodings and tree shapes.

Figure 3.1 shows an example of two wtree representations for the sequence \( S = \text{“once upon a time a PhD student”} \). Figure 3.1a shows the one-pointer-per-node
representation, while Figure 3.1b shows the one-pointer-per-level representation. In our algorithms, we implemented the one-pointer-per-level representation; however, for clarity, we use the one-pointer-per-node representation to exemplify. In both representations, we highlighted the traversal performed by the operation \text{access}(S, 24). To answer it, a top-down traversal of the wtree is performed: if a bit 0 is found, we visit the left branch; if a 1, the right branch is chosen. In the first representation, the query works as follows: Let curr be the root, \( B_{\text{curr}} \) be the bitmap of the current node, \( i = 24 \) be the index of interest, \( R \) be the range \([0, \sigma - 1] = [0, 15] \) and \( \text{rank}_c(B_{\text{curr}}, i) \) be the number of c-bits up to position \( i \) in \( B_{\text{curr}} \). At the beginning, we inspect the bit \( B_{\text{curr}}[i] \). Since the bit is 1, we recompute \( i = \text{rank}_1(B_{\text{curr}}, i) - 1 = 7 \), change curr to be the right child of curr and halve \( R = [8, 15] \). Then, we repeat the process. Since \( B_{\text{curr}}[i] = 0 \), \( i = \text{rank}_0(B_{\text{curr}}, i) - 1 = 4 \), curr is updated to be the left child of curr and \( R = [8, 11] \). Now, \( B_{\text{curr}}[i] = 0 \), \( i = \text{rank}_0(B_{\text{curr}}, i) - 1 = 2 \), curr is changed to be the left child of curr and \( R = [8, 9] \). Finally, in the last level, \( B_{\text{curr}}[i] = 0 \), so the range \( R = [8, 8] \) and the answer for \( \text{access}(S, 24) \) is \( \Sigma[8] = 't' \). \( \text{rank}_c(S, i) \) and \( \text{select}_c(S, i) \) perform similar traversals to \( \text{access}(S, i) \). For a more detailed explanation of wtree operations, see [103]. For the one-pointer-per-level representation, the procedure is similar, with the exception that the traversal of the tree must be simulated with rank operations over the bitmaps [24].

The wtree supports more complex queries than the primitives described above. For example, Mäkinen and Navarro [89] showed its connection with a classical two-dimensional range-search data structure. They showed how to solve range queries in a wtree and its applications in \textit{position-restricted searching}. In [82], the authors represent posting lists in a wtree and solve ranked AND queries by solving several range queries synchronously. Some work has been done in parallel processing of wtrees. In [4], the authors explore the use of wtrees in distributed web search engines. They assume a distributed memory model and propose partition techniques to balance the workload of processing wtrees. Note that our work is complementary to theirs, as each node in their distributed system can be viewed as a multicore computer that can benefit from our algorithms. In [84], the authors explore the use of SIMD instructions to improve the performance of wtrees (and other string algorithms, see, for example, [36]). This set of instructions can be considered as low-level parallelism. We can also benefit from their work as it may improve the performance of the sequential parts of our algorithms.

### 3.2.2 Succinct ordinal trees

Succinct ordinal trees were introduced in 1989 by Jacobson [76]. He showed how to represent an ordinal tree on \( n \) nodes using \( 2n + o(n) \) bits so that computing the first child, next sibling or parent of any node takes \( O(\lg n) \) time in the bit probe model. Clark and Munro [21] showed how to support the same operations in constant time in the word RAM model with word size \( \Theta(\lg n) \). Since then, much work has been done on succinct tree representations, to support more operations, to achieve compression,
to provide support for updates, and so on. See [111] for a thorough survey. Navarro and Sadakane [102] recently proposed a succinct tree representation, referred to as NS-representation throughout this thesis, which was the first to achieve a redundancy of $O(n/\lg^c n)$ bits for any positive constant $c$. The redundancy of a data structure is the additional space it uses above the information-theoretic lower bound. While all previous tree representations achieved a redundancy of $o(n)$ bits, their redundancy was $\Omega(n \lg \lg n/\lg n)$ bits, that is, just slightly sub-linear. The NS-representation also supports a large number of navigational operations in constant time (see Table 3.1); only the work in [61, 37] supports two additional operations. An experimental study of succinct trees [3] showed that a simplified version of this representation uses less space than other existing representations in most cases and performs most operations faster. In this thesis, we present a parallel algorithm for constructing the NS-representation.

Simplified NS-representation

The NS-representation is based on the balanced parenthesis sequence $P$ of the input tree $T$, which is obtained by performing a preorder traversal of $T$ and writing down an open parenthesis when visiting a node for the first time and a closed parenthesis after visiting all its descendents. Thus, the length of $P$ is $2n$. See Figure 3.2 as an example.

Figure 3.2: Balanced parentheses representation $P$ of a tree $T$. This representation, also known as folklore encoding, can be stored using a bit vector, writing a 1 for each open parenthesis and a 0 for each closed parenthesis.

![Example Tree and Folklore Encoding](image-url)

The NS-representation is not the first structure to use balanced parentheses to represent trees. Munro and Raman [96] used succinct representations of balanced parentheses to represent ordinal trees and reduced a set of navigational operations on trees to operations on their balanced parenthesis sequences. Their solution supports only a subset of the operations supported by the NS-representation. Additional operations can be supported using auxiliary data structures [88, 114, 100, 98], but supporting all operations in Table 3.1 requires many auxiliary structures, which increases the size of the final data structure and makes it complex in both theory and practice. The main novelty of the NS-representation lies in its reduction of a large set...
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 child(x, i)</td>
<td>Find the i\textsuperscript{th} child of node x</td>
</tr>
<tr>
<td>2 child_rank(x)</td>
<td>Report the number of left siblings of node x</td>
</tr>
<tr>
<td>3 degree(x)</td>
<td>Report the degree of node x</td>
</tr>
<tr>
<td>4 depth(x)</td>
<td>Report the depth of node x</td>
</tr>
<tr>
<td>5 level_anc(x, i)</td>
<td>Find the ancestor of node x that is i levels above node x</td>
</tr>
<tr>
<td>6 subtree_size(x)</td>
<td>Report the number of nodes in the subtree rooted at node x</td>
</tr>
<tr>
<td>7 height(x)</td>
<td>Report the height of the subtree rooted at x</td>
</tr>
<tr>
<td>8 deepest_node(x)</td>
<td>Find the deepest node in the subtree rooted at node x</td>
</tr>
<tr>
<td>9 LCA(x, y)</td>
<td>Find the lowest common ancestor of nodes x and y</td>
</tr>
<tr>
<td>10 lmost_leaf(x) / rmost_leaf(x)</td>
<td>Find the leftmost/rightmost leaf of the subtree rooted at node x</td>
</tr>
<tr>
<td>11 leaf_rank(x)</td>
<td>Report the number of leaves before node x in preorder</td>
</tr>
<tr>
<td>12 leaf_select(i)</td>
<td>Find the i\textsuperscript{th} leaf from left to right</td>
</tr>
<tr>
<td>13 pre_rank(x) / post_select(x)</td>
<td>Report the number of nodes preceding node x in preorder/postorder</td>
</tr>
<tr>
<td>14 pre_select/post_select(i)</td>
<td>Find the i\textsuperscript{th} node in preorder/postorder</td>
</tr>
<tr>
<td>15 level_lmost(i) / level_rmost(i)</td>
<td>Find the leftmost/rightmost node among all nodes at depth i</td>
</tr>
<tr>
<td>16 level_succ(x) / level_pred(x)</td>
<td>Find the node immediately to the left/right of node x among all nodes at depth i</td>
</tr>
<tr>
<td>17 access(i)</td>
<td>Report P[i]</td>
</tr>
<tr>
<td>18 find_open(i) / find_close(i)</td>
<td>Find The matching parenthesis of P[i]</td>
</tr>
<tr>
<td>19 enclose(i)</td>
<td>Find the closest enclosing matching parenthesis pair for P[i]</td>
</tr>
<tr>
<td>20 rank_open(i) / rank_close(i)</td>
<td>Report the number of open/closed parentheses in P[1..i]</td>
</tr>
<tr>
<td>21 select_open(i) / select_close(i)</td>
<td>Find the i\textsuperscript{th} open/closed parenthesis</td>
</tr>
</tbody>
</table>

Table 3.1: Operations supported by the NS-representation [102], including operations over the corresponding balanced parenthesis sequence.

of operations on trees and balanced parenthesis sequences to a small set of primitive operations. Representing $P$ as a bit vector storing a 1 for each open parenthesis and a 0 for each closed parenthesis (see Figure 3.2), these primitive operations are the following, where $g$ is an arbitrary function on \{0, 1\}:
\[
\sum(P, g, i, j) = \sum_{k=i}^j g(k)
\]

\[
fwd\text{_search}(P, g, i, d) = \min\{j \mid j \geq i, \sum(P, g, i, j) = d\}
\]

\[
bwd\text{_search}(P, g, i, d) = \max\{j \mid j \leq i, \sum(P, g, j, i) = d\}
\]

\[
rmq(P, g, i, j) = \min\{\sum(P, g, i, k) \mid i \leq k \leq j\}
\]

\[
RMQ(P, g, i, j) = \max\{\sum(P, g, i, k) \mid i \leq k \leq j\}
\]

\[
rmqi(P, g, i, j) = \arg\min_{k \in [i,j]} \{\sum(P, g, i, k)\}
\]

\[
RMQi(P, g, i, j) = \arg\max_{k \in [i,j]} \{\sum(P, g, i, k)\}
\]

Most operations supported by the NS-representation reduce to these primitives by choosing \(g\) to be one of the following three functions:

\[
\pi: 1 \mapsto 1 \quad \phi: 1 \mapsto 1 \quad \psi: 1 \mapsto 0
\]

\[
0 \mapsto -1 \quad 0 \mapsto 0 \quad 0 \mapsto 1
\]

For example, assuming the \(i\)th parenthesis in \(P\) is an open parenthesis, the matching closed parenthesis can be found using \(fwd\text{_search}(P, \pi, i, 0)\). Thus, it (almost)\(^1\) suffices to support the primitive operations above for \(g \in \{\pi, \phi, \psi\}\). To do so, Navarro and Sadakane designed a simple data structure called Range Min-Max Tree (RMMT), which supports the primitive operations above in logarithmic time when used to represent the entire sequence \(P\). To achieve constant-time operations, \(P\) is partitioned into chunks. Each chunk is represented using an RMMT, which supports primitive operations inside the chunk in constant time if the chunk is small enough. Additional data structures are used to support operations on the entire sequence \(P\) in constant time.

Next we review the RMMT structure and how it supports the primitive operations for \(g = \pi\) (In Figure 3.3 we show an example of the function \(\pi\)). Navarro and Sadakane [102] discussed how to make it support these operations also for \(\phi\) and \(\psi\) while increasing its size by only \(O(n/\lg^c n)\). To define the variant of the RMMT we implemented, we partition \(P\) into chunks of size \(s = w \lg n\), where \(w\) is the machine word size. For simplicity, we assume that the length of \(P\) is a multiple of \(s\). The RMMT is a complete binary tree over the sequence of chunks (see Figure 3.3). (If the number of chunks is not a power of 2, we pad the sequence with chunks of zeroes to reach the closest power of 2. These chunks are not stored explicitly.) Each node \(u\) of the RMMT represents a subsequence \(P_u\) of \(P\) that is the concatenation of the chunks corresponding to the descendant leaves of \(u\). Since the RMMT is a complete tree, we

\(^1\)A few navigational operations cannot be expressed using these primitives. The NS-representation includes additional structures to support these operations.
need not store its structure explicitly. Instead, we index its nodes as in a binary heap and refer to each node by its index. The representation of the RMMT consists of four arrays \( e', m', M', \) and \( n' \), each of length equal to the number of nodes in the RMMT. The \( u \)th entry of each of these arrays stores some crucial information about \( P_u \): Let the **excess** at position \( i \) of \( P \) be defined as \( \text{sum}(P, \pi, 0, i) = \sum_{k=0}^{i} \pi(P[k]) \). \( e'[u] \) stores the excess at the last position in \( P_u \). \( m'[u] \) and \( M'[u] \) store the minimum and maximum excess, respectively, at any position in \( P_u \). \( n'[u] \) stores the number of positions in \( P_u \) that have the minimum excess value \( m'[u] \).

Combined with a standard technique called **table lookup**, a RMMT supports the primitive operations for \( \pi \) in \( O(\lg n) \) time. Consider \( \text{fwd_search}(P, \pi, i, d) \) for example. We first check the chunk containing \( P[i] \) to see if the answer is inside this chunk. This takes \( O(\lg n) \) time by dividing the chunk into portions of length \( w/2 \), where \( w \) is the machine word size, and testing for each portion in turn whether it contains the answer. Using a lookup table whose content does not depend on \( P \), the test for each portion of length \( w/2 \) takes constant time: For each possible bit vector of length \( w/2 \) and each of the \( w/2 \) positions in it, the table stores the answer of \( \text{fwd_search}(P, \pi, i, d) \) if it can be found inside this bit vector, or \(-1\) otherwise. As there are \( 2^{w/2} \) bit vectors of length \( w/2 \), this table uses \( 2^{w/2}\text{poly}(w) \) bits. If we find the answer inside the chunk containing \( P[i] \), we report it. Otherwise, let \( u \) be the leaf corresponding to this chunk. If \( u \) has a right sibling, we inspect the sibling’s \( m' \) and \( M' \) values to determine whether it contains the answer. If so, let \( v \) be this right sibling. Otherwise, we move up the tree from \( u \) until we find a right sibling \( v \) of an ancestor of \( u \) whose corresponding subsequence \( P_v \) contains the query answer. Then we use a similar procedure to descend down the tree starting from \( v \) to look for the leaf descendant of \( v \) containing the answer and spend another \( O(\lg n) \) time to determine the position of the answer inside its chunk. Since we spend \( O(\lg n) \) time for each of the two leaves we inspect and the tests for any other node in the tree take constant time, the cost is \( O(\lg n) \).

Figure 3.3 shows the \( m' \) and \( M' \) values involved in the answer of \( \text{fwd_search}(P, \pi, 5, 1) \). In this particular example, the objective is to find the closest...
position after $i = 5$ with excess value $d = 1$. Using lookup tables, we check if the answer is in the range $[5, 6]$ of the chunk $[5/7] = 0$. Since the answer is not there, we analyze the right sibling of the chunk 0. The $m'$ and $M'$ values of the right sibling are 2 and 4, so the answer is not there. We now move to the parent of the parent of the chunk 0. Let’s call $v$ to such node. The $m'$ and $M'$ values of $v$ are 0 and 5, and therefore, the answer exists and it is in the right child of $v$. Then, we check the $m'$ and $M'$ values of the child of $v$, 1 and 5, and move to the left child of the right child of $v$. Since the current node is a leaf, we use lookup tables to find the first value 1 in that chunk. In this case, such 1 value is at position 20.

Supporting operations on the leaves, such as finding the $i$th leaf from the left, reduces to \textbf{rank} and \textbf{select} operations over a bit vector $P_1[1..2n]$ where $P_1[i] = 1$ iff $P[i] = 1$ and $P[i + 1] = 0$. \textbf{rank} and \textbf{select} operations over $P_1$ in turn reduce to \textbf{sum} and \textbf{fwd_search} operations over $P_1$ and can thus be supported by an \textbf{RMMT} for $P_1$. $P_1$ does not need to be stored explicitly because any consecutive $O(w)$ bits of $P_1$ can be computed from the corresponding bits of $P$ using table lookup.

**Constant time queries**

To support constant time queries on arbitrary-sized trees, the balanced parentheses representation $P$ needs to be partitioned into blocks. We represent each block using a \textbf{RMMT} and then construct additional data structures considering the minimum, maximum and excess values of the \textbf{RMMT} of each block. The size of each block is $w^e$, so we have $\tau = \lceil 2n/w^e \rceil$ of such blocks. To support constant time queries inside each block, we construct a \textbf{RMMT}, similar as before, but with $s = w/2$ and arity $k = \Theta(w/c \lg w)$, instead of arity 2. Let $m_1, \ldots, m_\tau$, $M_1, \ldots, M_\tau$, $n_1, \ldots, n_\tau$ and $e_1, \ldots, e_\tau$ be the minima, maxima, number of minima and excess stored at the root of the $\tau$ \textbf{RMMTs}. Depending on the operations, the additional data structures differ.

To solve $\textbf{fwd_search}(P, \pi, i, d)$, we first try to solve it inside block $j = \lfloor i/w^e \rfloor$. The answer is returned if it is found in that block. If it is not, we must find the first excess $d' = d + e_{j-1} + \text{sum}(P, \pi, 0, i - 1 - w^e \cdot (j - 1))$ in the \textbf{RMMTs} of the following blocks. Applying Lemma 4.4 of [102], we must either find the first block $r > j$ such that $m_r \leq d'$, or such that $M_r \geq d'$. Once we find such a block, we complete the query inside of it with a local $\textbf{fwd_search}(P, \pi, 0, d' - e_{r-1})$.

To find the corresponding block $r$ in constant time, the authors propose additional data structures to represent the left-to-right minima and maxima values. For the case of left-to-right minima, it is necessary to build a tree called \textit{2d-Min-Heap} (The left-to-right maxima is similar):

**Definition 1.** [102] Let $m_1, \ldots, m_\tau$ be a sequence of integers. We define for each $1 \leq j \leq \tau$ the left-to-right minima starting at $j$ as $\text{lrm}(j) = (j_0, j_1, \ldots)$ where $j_0 = j$, $j_r < j_{r+1}$, $m_{j_{r+1}} < m_{j_r}$, and $m_{j_{r+1}}, \ldots, m_{j_{r+1}-1} \leq m_{j_r}$.

Once two $\text{lrm}$ sequences coincide, they do so until the end. Thus, a 2d-Min-Heap is defined as a trie of $\tau$ nodes, composed of the reversed $\text{lrm}$ sequences. Since
The resulting trie can be composed of disconnected paths, a dummy root is used to generate the tree. If we assign weight to the edges, where the weight of an upward edge \((j_i, j_{i+1})\) is defined as \(m_{j_i} - m_{j_{i+1}}\), we can reduce the problem of finding the first block \(r > j\) such that \(m_r \leq d'\) to the a weighted level ancestor query over the 2d-Min-Heap. More precisely, we need to find the first ancestor \(j_r\) of node \(j\) such that the sum of the weights between \(j\) and \(j_r\) is greater than \(d'' = m_j - d'\). Figure 3.4a shows an example of the 2d-Min-Heap for the sequence \((1, 4, 9, 5, 10, 7, 3, 2, 5, 4)\).

To answer the weighted level ancestor query, we need to decompose the 2d-Min-Heap. The 2d-Min-Heap is decomposed into paths by recursively extracting the longest path. Then, for each path of length \(l\), we store an extension of it by adding at most \(l\) nodes towards the root. These extended paths are called ladders. Figure 3.4b shows an example of ladders. This decomposition ensures that a node with height \(h\) will have its first \(h\) ancestors in its ladder. For each ladder, a sparse bitmap is stored, where the \(i\)-th 1 of the bitmap represents the \(i\)-th node upward in the ladder, and the distance between two 1’s is equal to the weight of the edge between them. All the bitmaps are concatenated into one of size \(O(n)\), which is represented by the sparse bitmap of Pațrașcu [106]. Additionally, for each node \(v\) of the 2d-Min-Heap, the \(\lg \tau\) ancestors at depths \(\text{depth}(v) - 2^i, i \geq 0\) are stored in an array. Similarly, for each node \(v\), the \(\lg \tau\) accumulated weights toward the ancestors at distance \(2^i\) are stored using fusion trees [45]. Fusion trees are used to store \(z\) keys of \(l\) bits each one in \(O(zl)\) bits, supporting predecessor queries in \(O(\lg z)\) time, by using a \(l^{1/6}\)-ary tree. The \(1/6\) factor can be reduced to achieve \(O(1/\epsilon)\) predecessor query, where \(0 < \epsilon \leq 1/2\) [102].
Observe that there is no guarantee that the weighted level ancestor \( j_r \) of the node \( j \) is in the ladder of \( j \). Therefore, to answer the weighted level ancestor query we need first to compute the ancestor \( j' \) of node \( j \) at distance \( 2^\lceil \lg (\text{depth}(j) - d^*) \rceil \). The answer is in the ladder of \( j' \). The ancestor \( j' \) can be founded in constant time by a predecessor query of fusion tree of the node \( j \) and the array with the \( \lg \tau \) ancestors of the node \( j \). If \( j' \) is at distance \( 2^i \), then the answer is at distance less than \( 2^{i+1} \). Applying \( \text{rank}/\text{select} \) queries over the bitmap of the ladder of node \( j' \), we find the node \( j_r \).

To solve \( \text{rmqi}(P,g,i,j) \) and \( \text{RMQi}(P,g,i,j) \) operations on the \( \tau \) blocks, we just need to build a data structure that supports range minimum and maximum queries in constant time, such as [39, 113].

To solve \( \text{degree}(i) \) operations, we need to consider pioneers. Let pioneers be the highest matching pair of parentheses \((i,j)\), with \( j = \text{find\_close}(i) \), such that \( i \) and \( j \) belong to different blocks. Let’s call a marked block is a block that has the opening parenthesis of a pioneer \((i,j)\) such that \((i,j)\) contains a whole block, i.e., \( i \) and \( j \) do not belong to consecutive blocks. Let \( a \) be a marked block with pioneer \((i,j)\) and let \( b \) be a block, we say that the block \( a \) contains the block \( b \) if the block \( b \) is between the blocks where \( i \) and \( j \) belong. There are \( O(\tau) \) of such marked blocks. The \( \text{degree}(i) \) operation, number of children of a node \( i \), can be solved as follows: If the operation involves at most two consecutive blocks, then the answer can be computed in constant time consulting the two corresponding RMMTs. Otherwise, it corresponds to the degree of a marked block. Since there are \( O(\tau) \) of such blocks, we can spend \( O(\tau \lg n) \) bits to store explicitly the degree of all the marked blocks and answer the operation in constant time.

The marked blocks are also used to solve \( \text{child}(i,q) \) and \( \text{child\_rank}(i) \) operations. Both for \( \text{child}(i,q) \) and \( \text{child\_rank}(i) \), if the block of \( i \) is not a marked block, then both can be solved in at most two in-block queries. For marked blocks, we store a bitmap to represent the information about the children of each of them. For each marked block \( j \), we store, in left-to-right order, information of marked blocks and blocks fully contained in \( j \). For each block \( j' \) contained in block \( j \), we store the number of children of \( j \) that starts within \( j' \) (the number of minima of block \( j' \)) and for each children-marked block, we store a 1, which represents a block containing one child of \( j \). All numbers are stored in a bitmap as gaps of 0’s between consecutive 1’s. For the \( \text{child}(i,q) \) query, we first check if \( \text{child}(i,q) \) lies in the block of \( i \) or in \( \text{find\_close}(i) \). If true, we solve it with an in-block query. If not, we compute \( p = \text{rank}_1(C_i, \text{select}_0(C_i,q)) \), where \( C_i \) is the bitmap associated to the block of \( i \). The value \( p \) represents the position of the block or marked block contained in \( i \), where the \( q \)-th child of \( i \) lies. If it is a marked block, then that is the answer. If it is a block \( j \), then the answer corresponds to the \( q' \)-th minimum within that block, where \( q' = q - \text{rank}_0(C_i, \text{select}_1(C_i,p)) \). \( \text{child\_rank}(i) \) can be solved similarly. Since the number of 1’s on each bitmap is less than the number of 0’s, the bitmap can be stored using the sparse bitmap of [106].

The remaining operations require \( \text{rank} \) and \( \text{select} \) on \( P \), or the virtual bit vectors \( P_1 \) and \( P_2 \). For \( \text{rank} \), it is necessary to store the answers at the end of each block,
finishing the query inside the corresponding block. For select₁ (and select₀), we build a sequence with the accumulated 1’s in each of the τ blocks. Such sequence is stored in a bitmap, representing each number in unary as gaps of 0’s between consecutive 1’s using the results of [106].

**Memory space**

To analyze the space used for the simplified NS-representation, observe that storing P requires 2n bits, while the space needed to store the vectors e’, m’, M’, and n’ is 2(n/s) lg n = 2n/w. The space needed to store the same vectors for the RMMT of P₁ is the same. Since we can assume that w = Ω(lg n), the total size of the simplified RMMT is thus 2n + O(n/lg n) bits.

The NS-representation that supports constant time queries requires the construction of τ = [2n/wc] RMMTs over sequences of w/c parentheses. Thus, the τ RMMTs require 2n + O(n/lg n) bits to be stored. The additional data structures needed to support constant time queries add some extra space: To support fwd_search, the ladders use O((lg² n)/w) bits, the arrays of ancestors use O((lg² n)/w) bits, the sparse bitmap uses O((lg n/wc + n’t/4 lg n + n/3√n) bits and the fusion trees use O((lg² n)/w) bits. Thus, the extra structures to support fwd_search uses O((lg² n/wc + n’t/4 lg n + n/3√n) bits, with t > 0. The rmqi and RMQ_i queries add O(n/wc) extra bits. Since there are O(n/wc) marked blocks, the degree operation uses O(n lg n/wc) extra bits. The bitmaps of the remaining operations, such as child and child_ranks, uses \(2\alpha_n \lg(w/c) + O(n’t/4 lg n + n^3/4)\) extra bits, since they correspond to the sparse bitmap of [106]. Therefore, the total space used by the additional data structures is \(O((\alpha_n lg(w + lg² n)) + n’t/4 lg n + n^3/4 + \sqrt{2^w})\) bits, where the term \(\sqrt{2^w}\) corresponds to the lookup tables. With w = Ω(lg n) and t = c, the extra space is \(O((\alpha_n lg n + lg^² n)/lg n + n^3/4 + \sqrt{n})\) bits. Combined with the 2n + O(n/lg n) bits of the RMMTs, the NS-representation requires 2n + O(n/lg n + \(n(c lg lg n + lg^2 n)/lg n\)) bits, with c > 0, supporting queries in O(c) time.

According to [102], the O(n/lg n) space of the RMMTs can be reduced by using aB-trees [106]. Given an array A of size N, with N a power of B, an aB-tree is a complete tree of arity B, that stores B consecutive elements of A on its leaves. Besides, each node of the aB-tree stores a value \(φ \in Φ\). For the leaves, \(φ\) must be a function of the elements of A that it stores; for internal nodes, \(φ\) must be a function of the \(φ\)-values of its children. An aB-tree can decode the B \(φ\)-values of the children of any internal node and the B values of A for the leaves in constant time, if they are packed in a machine word. An aB-tree can be stored in \(N + 2 + O(\sqrt{2^w})\) bits (See [106] for the details). Thus, with \(A = P, B = k = s = O(\frac{w}{lg w})\), \(φ\)-values encoding e’, m’, M’, n’ values and blocks of size \(N = B^c\), it is possible to store each iRMMT in \(N + 2 + O(\sqrt{2^w})\) bits. The sum of all the RMMTs is \(2n + O(\frac{n}{B^c} + \sqrt{2^w}) = 2n + O(\frac{n(c lg lg n)/lg n}{lg n} + \sqrt{2^w})\). Finally, using aB-trees to store the RMMTs, the space usage of the NS-representation is reduced to \(2n + O(\frac{n(c lg lg n + lg^2 n)/lg n}{lg n} + \sqrt{2^w})\) bits.
3.2.3 Triangulated plane graphs

The triangulated plane graphs have been used to represent polytopes, geographic maps, microchip layouts and design, software engineering diagrams, surface meshes in computer graphics, among others. In practice, triangulated plane graphs may be large, such as in VLSI (very large scale integrations) circuits or TIN (triangulated irregular network) surfaces. Thus, the design of space-efficient representations of triangulated plane graph becomes useful.

A graph $G = (V, E)$, with $|V| = n$ vertices, $|E| = m$ edges and $f$ faces, is a triangulated planar graph if $G$ is planar and the addition of any edge to $G$ results in a nonplanar graph. A triangulated planar graph with a particular drawing or embedding is a triangulated plane graph. Triangulated plane graphs are also known as maximal plane graphs. See Figure 3.5 as an example of a triangulated plane graph. Notice that a triangulated plane graph with $n$ vertices has $3n - 6$ edges, $2n - 4$ faces and all its faces are triangles.

A common approach to construct succinct representations of triangulated plane graphs is to decompose them into a set of trees and subgraphs, representing them as parentheses sequences to then apply some ideas of succinct ordinal trees to support operations in optimal time. Operations of interest are the computation of the degree of a vertex (degree) and the adjacency test of two vertices (adjacency). Usually, decomposition is achieved using either canonical ordering [20, 19, 64, 5], realizers [7, 115] and orderly spanning trees [18]. In [93], authors show that canonical orderings, realizers and orderly spanning trees are equivalent on triangulated plane graphs. We adopt the canonical ordering approach, but extend our results to realizers which
support more operations.

Succinct representation of triangulated plane graphs based on canonical orderings

Before explaining the succinct representation of triangulated plane graphs based on canonical ordering, we need to introduce the definition of canonical ordering. The canonical orderings of a maximal plane graph were introduced by de Fraysseix, Pach, and Pollack [30, 44] and later generalized by Kant [80]. Canonical orderings are the necessary input to several graph-drawing algorithms that work on plane graphs, such as [81, 80, 30, 44].

A canonical ordering is defined as follows: let $G = (V, E)$ be a triangulated plane graph, where $|V| = n \geq 3$, $|E| = m$ edges, vertices $u$, $v$ and $w$ are in the outer face of $G$, and where $\Pi = (v_1, \ldots, v_n)$ is an ordering of $V$ such that $v_1 = u$, $v_2 = v$ and $v_n = w$. Let $G_k$ be the sub-graph of $G$ induced by $v_1, \ldots, v_k$, and $C_k$ to be the contour of $G_k$. We say that $\Pi$ is a canonical ordering of $G$ if the following conditions are satisfied for each $k = (3, 4, \ldots, n-1, n)$:

- Each $G_k$ is 2-connected and internally triangulated.
- $C_k$ contains $(v_1, v_2)$.
- If $k < n$, then $v_{k+1}$ is in the outer face of $G_k$ and all neighbours of $v_{k+1}$ in $G_k$ appear on $C_k$ consecutively.

For example, the canonical ordering of the graph in Figure 3.5 is $\Pi = (a, b, c, h, i, j, d, e, k, f, l, g)$.

In [44, 59] a sequential algorithm to compute the canonical ordering in linear time was introduced. The algorithm labels the vertices with a $-1$ if the vertex has not been visited yet, a 0 if the vertex has been visited once or $i > 0$ if the vertex has been visited more than once. The meaning of $i > 0$ will be explained later.

The algorithm is detailed in Algorithm 3. The input consists of a triangulated plane graph with external vertices $u, v$ and $w$. All the vertices in the graph are labelled with $-1$, except the externals. Vertices $u$ and $v$ are initially labelled with 1. The vertex $w$ is not processed for the algorithm. Instead, we assign $w = v_n$, the last vertex in the canonical ordering.

The algorithm starts by choosing a vertex with label 1, $vv$ (line 3). The chosen vertex will take the next position in the canonical ordering of $G$ (line 4). Notice that, at the beginning, the external vertices $u$ and $v$ will be the first chosen vertices, assigning order $v_1$ and $v_2$ to them. After choosing the vertex $vv$, the algorithm visits all the neighbours of $vv$. Let $v'$ a neighbor of $vv$. The first time that $v'$ is visited, the algorithm changes its label to 0 (lines 7-8). If $v'$ has label 0, then $v'$ has another neighbor that has been processed. If such a neighbor and $vv$ are adjacent in the counterclockwise (ccw) order of the neighbors of $v'$, then, the algorithm relabels
**Input**: Triangulated plane graph $G$ with external vertices $u, v$ and $w$. All the vertices, except the externals, are labelled with $-1$. Vertices $u$ and $v$ are initially labelled with 1 and vertex $w$ will not be processed.

**Output**: The canonical ordering of the graph $G$.

```plaintext
1 $k = 1$
2 while There are unprocessed vertices do
3     $vv = a$ vertex of $G$ with label 1
4     Assign the order $k$ to $vv$ in the canonical ordering of $G$
5     $k = k + 1$
6     foreach neighbor $v'$ of $vv$ do
7         if $v'$ has label $-1$ then
8             relabel $v'$ with 0
9         else if $v'$ has label 0 then
10            $v'_1, v'_2 = AdjNeighbors(v', vv)$
11            if $v'_1$ or $v'_2$ has been processed then
12                relabel $v'$ with 1
13            else
14                relabel $v'$ with 2
15         else if $v'$ has label $i > 0$ then
16            $v'_1, v'_2 = AdjNeighbors(v', vv)$
17            if $v'_1$ and $v'_2$ have been processed then
18                relabel $v'$ with $i - 1$
19            else if neither $v'_1$ and $v'_2$ have been processed then
20                relabel $v'$ with $i + 1$
21     Mark $vv$ as processed
```

**Algorithm 3**: Sequential algorithm to compute the canonical ordering of a triangulated plane graph.

$v'$ with 1. Otherwise, the algorithm relabels it with 2. To check the condition in constant time, the algorithm uses the function $AdjNeighbors(v', vv)$ which returns the two adjacent vertices of $vv$, $v'_1$ and $v'_2$, in the ccw order of the neighbors of $v'$. It suffices to check if $v'_1$ or $v'_2$ has been processed (lines 9-14). In the third condition (lines 15-20), the label $i > 0$ of $v'$ means that there are $i$ intervals of processed neighbors in the list of neighbors of $v'$, in ccw order. Thus, if the two adjacent vertices of $vv$ in the ccw order of the neighbors of $v'$ have been processed, then the algorithm relabels $v'$ with $i - 1$ (two intervals are merged). If neither have been processed, then the algorithm relabels $v'$ with $i + 1$ (a new interval is added). Otherwise, the label of $v'$ does not change. After checking all the conditions, the algorithm marks the vertex $vv$ as processed, which means that it will not be chosen again. The algorithm finishes when all the vertices have been processed. Since each vertex is processed once and the number of edges of $G$ is $3n - 6$, the algorithm takes $O(n \lg n)$ time, by using a heap structure to retrieve the vertices with label 1.

Table 3.2 shows an execution example of this algorithm taking the graph of Figure 3.5 as input. The output corresponds to the canonical ordering in Figure 3.5. The
iterations of the algorithm are shown in the rows of the table. In the first row, the external vertex \( g \) is designated as the last vertex of the canonical ordering, \( v_{12} \). In each iteration, the chosen vertex is circled. Observe that the canonical ordering is not unique. For example, in the iteration three of Table 3.2, instead of the vertex \( h \), the algorithm could have chosen the vertex \( d \). No matter which vertex is chosen, the output canonical ordering will be correct.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>( \bullet )</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>v_{12}</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>Processing ( v_1/\alpha )</td>
<td>( v_1 )</td>
<td>( \bullet )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>v_{12}</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>Processing ( v_2/\beta )</td>
<td>( v_1 ) v_{2}</td>
<td>( \bullet )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>v_{12}</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Processing ( v_3/\gamma )</td>
<td>( v_1 ) v_{2} v_{3}</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>( \bullet )</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Processing ( v_4/\delta )</td>
<td>( v_1 ) v_{2} v_{3}</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>( \bullet )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Processing ( v_5/\epsilon )</td>
<td>( v_1 ) v_{2} v_{3}</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>( \bullet )</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Processing ( v_6/\zeta )</td>
<td>( v_1 ) v_{2} v_{3}</td>
<td>( \bullet )</td>
<td>0</td>
<td>2</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Processing ( v_7/\eta )</td>
<td>( v_1 ) v_{2} v_{3} v_{7}</td>
<td>( \bullet )</td>
<td>2</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Processing ( v_8/\iota )</td>
<td>( v_1 ) v_{2} v_{3} v_{7} v_{8}</td>
<td>1</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>( \bullet )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Processing ( v_9/\kappa )</td>
<td>( v_1 ) v_{2} v_{3} v_{7} v_{8}</td>
<td>( \bullet )</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>v_{9}</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Processing ( v_{10}/\lambda )</td>
<td>( v_1 ) v_{2} v_{3} v_{7} v_{8} v_{9} v_{10}</td>
<td>v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>v_{9}</td>
<td>v_{10}</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Processing ( v_{11}/\mu )</td>
<td>( v_1 ) v_{2} v_{3} v_{7} v_{8} v_{9} v_{10} v_{12}</td>
<td>v_{4}</td>
<td>v_{5}</td>
<td>v_{6}</td>
<td>v_{7}</td>
<td>v_{8}</td>
<td>v_{9}</td>
<td>v_{10}</td>
<td>v_{11}</td>
<td>( \bullet )</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: An execution of Algorithm 3. Each row is an iteration of the algorithm. Iterations are ordered top-down according to their execution. In each iteration, the vertex that will be processed next is circled. The input is the graph in Figure 3.5. “Processing \( v_k/x \)” represents the iteration where the order \( k \) is designated to the vertex \( x \).

He et al. [63] introduced the only parallel algorithm that computes the canonical ordering of a triangulated plane graph in \( O(\log^4 n) \) time using \( O(n^2) \) processors in the CREW PRAM model. The authors defined the extended graph \( \hat{G} \) of \( G \) as the graph obtained after adding a new vertex \( v_f \) to \( G \) for each internal face \( f \), connecting \( v_f \) to each vertex in the boundary of \( f \). The algorithm is based on computing independent sets and a realizer of \( \hat{G} \). As far as we know, there is not an implementation of this algorithm.

Given the canonical ordering introduced in the preceding paragraphs, a succinct representation of \( G \) can be constructed using its canonical spanning tree, \( T_{co} \) [20, 19, 64]. The canonical spanning tree is a tree rooted at \( v_1 \), including the edge \((v_1, v_2)\) and edges \((v_i, v_j)\), where \( i < j \) in the canonical ordering and the node \( v_i \) is the leftmost neighbor of node \( v_j \) in ccw order (see Figure 3.5). Using parentheses to encode \( T_{co} \) and brackets to encode the edges in \( G \setminus T_{co} \), a string \( S_{co} \) is built as follows:

1. \( S_{co} = FE(T_{co}) \), where \( FE(T_{co}) \) is the folklore encoding of \( T_{co} \). In the folklore encoding, each node \( v_i \) of \( T_{co} \) is represented by a parenthesis pair (\( i \) and \( i \)).

2. For each edge \((v_i, v_j)\) of \( G \setminus T_{co} \), with \( i < j \), write a “[” right after \( i \) and a “]” right after \( j \). For us, \( G \setminus T_{co} \) represents the set of edges that belong to \( G \), but not to \( T_{co} \).
See Figure 3.6 as an example of the string $S_{co}$. Observe that, as was pointed in [19], an equivalent definition of the construction of $S_{co}$ is:

- $S_{co} = FE(T_{co})$.
- For each vertex $v_i$ of $T_{co}$, count the number of lower-numbered neighbors, $l_i$, and higher-numbered neighbors, $h_i$, of $v_i$ in $G \setminus T_{co}$.
- For each vertex $v_i$ of $T_{co}$, write $l_i$ “]”s right after ($i$ and $h_i$ “[”s right after $i$.

We will use this equivalent definition in Section 6.1.1 to present our parallel algorithm to compute $S_{co}$.

Let $n()$ and $n[]$ the number of parentheses and brackets of $S_{co}$, respectively. $S_{co}$ can be encoded by bit-vectors, $S_1$ and $S_2$, with size $2n() + n[] + o(n() + n[])$ bits. Such bit-vectors are defined as follows:

- If $S_{co}[i]$ is a parenthesis, with $1 \leq i \leq n() + n[]$, then $S_1[i] = 1$. Otherwise, $S_1[i] = 0$.
- If the $j$-th parenthesis of $S_{co}$ is open, with $1 \leq j \leq n()$, then $S_2[j] = 1$. Otherwise, $S_2[j] = 0$.

Applying standard techniques of succinct representations of bit-vectors and balanced parentheses sequences [99, 102] over $S_1$ and $S_2$, we can retrieve $S_{co}[i]$ in constant time as follows.

- If $S_1[i] = 1$, $S_{co}[i]$ is a parenthesis. If $S_2[\text{rank}_1(S_1,i)] = 1$, then $S_{co}[i]$ is an open parenthesis. Otherwise it is a closed parenthesis.
- If $S_1[i] = 0$, $S_{co}[i]$ is a bracket. If $S_2[\text{select}_1(S_1,\text{rank}_1(S_1,i))] = 1$, then $S_{co}[i]$ is a closed bracket. Otherwise it is an open bracket.

In order to support degree and adjacency queries in constant time, the authors of [19, 20, 64] propose the construction of two strings of length $2|S_{co}|$, $S()$ and $S[]$. $S()$ and $S[]$ are defined in a similar way. For each $0 \leq i < |S_{co}|$:

- If $S_{co}[i]$ is an open parenthesis, then $S()[2i]$ and $S()[2i + 1]$ are defined as open parentheses.
- If $S_{co}[i]$ is a closed parenthesis, then $S()[2i]$ and $S()[2i + 1]$ are defined as closed parentheses.
- Otherwise, $S()[2i]$ is defined as an open parenthesis and $S()[2i + 1]$ as a closed parenthesis.
Figure 3.6: Parentheses representation $S_{co}$ of the graph in Figure 3.5. In the string $S_{co}$, parentheses represent the edges of $T_{co}$ and brackets represent the edges of $G \setminus T_{co}$. The bit-vectors $S_1$, $S_2$ and $S_3$ obtained from $S_{co}$ are also shown.

Again, applying standard techniques of succinct representations of bit-vectors and balanced parentheses sequences over $S()$ and $S[]$, degree and adjacency queries can be answered in constant time. Note that the strings $S()$ and $S[]$ are not explicitly stored, because each symbol of $S()[i]$ and $S[][i]$ can be determined from $S_{co}[\lfloor i/2 \rfloor]$ in constant time.

Observe that strings $S()$ and $S[]$ may be replaced by a bit-vector $S_3$ of size $n[]$ and still maintain constant-time support for degree and adjacency queries. The bit-vector $S_3$ is defined as follows:

- If the $i$-th bracket of $S_{co}$ is an open bracket, with $1 \leq i \leq n[]$, then $S_3[i] = 1$. Otherwise $S_3[i] = 0$.

An example of bit-vectors $S_1$, $S_2$ and $S_3$ are shown in Figure 3.6.

Before defining the degree and adjacency queries using $S_3$, we need some extra definitions:

- $\text{first}_1(S_{co}, i)$: The position of the closest 1 after $S_{co}[i]$. It can be computed in $O(1)$ with $\text{select}_1(S_1, \text{rank}_1(S_1, i) + 1)$.

- $\text{last}_1(S_{co}, i)$: The position of the closest 1 before $S_{co}[i]$. It can be computed in $O(1)$ with $\text{select}_1(S_1, \text{rank}_1(S_1, i) - 1)$.

- $\text{enclose}_1(S_3, i, j)$: The position $(p, q)$ of the closest matching bracket pair that encloses $S_3[i]$ and $S_3[j]$. Since $S_3$ can be seen as a parentheses sequence, it can be computed in $O(1)$ using the results of [102] (see Section 3.2.2).

- $\text{op}(S_{co}, i)$: The position in $S_{co}$ of the open parenthesis associated to the vertex $v_i$.

- $\text{cp}(S_{co}, i)$: The position in $S_{co}$ of the closed parenthesis associated to the vertex $v_i$.

Two vertices $v_i$ and $v_j$, with $i < j$, are adjacent in $G$ if they are adjacent in $T_{co}$ or in $G \setminus T_{co}$. Otherwise, they are not adjacent. If $v_i$ is the parent of $v_j$ in
Let \( T_{co} \), then \( \text{level}_\text{anc}(v_j, 1) = v_i \), which means that \( v_i \) and \( v_j \) are adjacent in \( T_{co} \) (see Table 3.1, operation 5). In \( G \setminus T_{co} \), if \( cp(S_{co}, i) < p < q < \text{first}_1(S_{co}, op(S_{co}, j)) \), where \( (p, q) = \text{enclose}(S_3, \text{rank}_0(S_1, \text{first}_1(S_{co}, cp(S_{co}, j))), op(S_{co}, j)) \), then they are adjacent.

The degree of a vertex \( v_i \) in \( G \) is the degree of \( v_i \) in \( T_{co} \) plus the degree of \( v_i \) in \( G \setminus T_{co} \). The degree of a vertex in \( T_{co} \) can be computed in \( O(1) \) using the results of \([102]\). The degree of a vertex in \( G \setminus T_{co} \) can be computed in \( O(1) \) with \((\text{first}_1(S_{co}, op(S_{co}, j)) - op(S_{co}, j) + (\text{first}_1(S_{co}, cp(S_{co}, j)) - cp(S_{co}, j))\).

The succinct representation described above uses \( 2m + 2n + o(m + n) \) bits, while still supporting degree and adjacency queries in \( O(1) \) time.

**Succinct representation of triangulated plane graphs based on realizers**

Schnyder introduced realizers, also called Schnyder woods, in \([115]\). A realizer of a triangulated plane graph \( G = (V, E) \), with \( |V| = n \geq 3 \) vertices, \( |E| = m \) edges, and external vertices \( u, v, w \), is a partition of the interior edges of \( G \) in three sets \( T_1, T_2, T_3 \) of directed edges such that for each interior vertex \( v' \) it holds that:

- \( v' \) has outdegree one in each of \( T_1, T_2, T_3 \).
- The counterclockwise ordering of the edges incident on \( v' \) is: leaving in \( T_1 \), entering in \( T_3 \), leaving in \( T_2 \), entering in \( T_1 \), leaving in \( T_3 \), entering in \( T_2 \). This is also known as local condition.

With this partition, each set \( T_1, T_2 \) and \( T_3 \) forms a directed tree, which includes all the internal vertices and one of the external vertices. See Figure 3.7 as an example.

Schnyder \([115]\) also describes a linear time algorithm to compute realizers based on edge contraction and in an ordering of the vertices of \( G \). This ordering is equivalent to a canonical ordering as defined previously. Here, we describe a variation of the original algorithm, based on a canonical ordering \( \Pi = (v_1, v_2, \ldots, v_n) \) of \( G \). This variation was presented in \([19, 101]\). A realizer of \( G \) can be computed as follows:

- First we choose \( v_1 \) and \( v_2 \) as the root of \( T_1 \) and \( T_2 \), respectively.
- Let \( C_k \) be the contour of the sub-graph of \( G \) induced by \( v_1, \ldots, v_k \). For each \( v_k \), \( 3 \leq k \leq n - 1 \), let \( C_{k-1} \) be \((w_0 = v_1, v_1, \ldots, w_l = v_2)\), and the neighbours of \( v_k \) on \( C_{k-1} \) are \( w_p, w_{p+1}, \ldots, w_q \). Orient \((w_p, v_k) \in T_1 \) towards \( w_p \), \((w_q, v_k) \in T_2 \) towards \( w_q \), and \((w_{p+1}, v_k), (w_{p+2}, v_k), \ldots, (w_{q-1}, v_k) \in T_3 \) towards \( v_k \).
- Finally, choose \( v_n \) as the root of \( T_3 \), and set all inner incident edges to \( v_n \) in \( T_3 \).

The derived partition of inner edges \( T_1, T_2 \) and \( T_3 \) is a realizer. For instance, considering the canonical ordering given in Figure 3.5, we obtain the realizer shown in Figure 3.7. Nakano \([101]\) shows that it is possible to obtain a canonical ordering from a realizer. Observe that \( T_1 \) corresponds to the canonical spanning tree of \( G \).
Figure 3.7: Realizer of the graph in Figure 3.5. $T_1$ is represented by thick edges, $T_2$ by dashed edges and $T_3$ by dotted edges. This realizer is also induced by the canonical ordering of Figure 3.5.

In [47], Fürer et al. present a parallel algorithm to construct realizers and straight-line embeddings in $O(\log n \log \log n)$ time using $O(n/\log n \log \log n)$ processors under the CRCW PRAM model.

Barbay et al. [7] introduced a succinct representation of plane graphs based on realizers. With $T_1$, $T_2$ and $T_3$, the authors define three new orders of the vertices that are used later to encode the graph using a string of three kinds of parentheses. The first order is called zeroth order and correspond to the ccw pre-order of $T'_1 = T_1 \cup (v_1, v_2) \cup (v_1, v_n)$. The other two orders, first order and second order, use the same ccw pre-order to compute their own order of vertices, but based on trees $T'_2 = T_2 \cup (v_2, v_n)$ and $T_3$, respectively. Authors explain that by assigning a different kind of parenthesis to each pre-order and merging them into a parentheses sequence $S'_{rz}$, it is possible to obtain a succinct representation of $G$ that uses $2m \lg 6 + o(m)$ bits, which supports in constant time adjacency and degree operations and two extra operations: Find the $i$th neighbor of a vertex in ccw order and find the number of neighbors between two other vertices.

The parenthesis sequence $S'_{rz}$ is computed as follows [7]:

1. $S'_{rz} = FE(T'_1)$, where $FE(T'_1)$ is the folklore encoding of $T'_1$. In the folklore encoding, each node $v_i$ of $T'_1$ is represented by a parenthesis pair $(i \text{ and } i)$.

2. Let $v'_1, v'_2, \ldots, v'_n$ be the ccw preorder of the nodes of $T'_1$. For each vertex $v'_i$, visit all the neighbors of $v'_i$ in ccw order and insert:

   - A "[" for each edge $(v'_i, v'_j)$ in $T'_2$, where $i < j$, right before $)$. 


Figure 3.8: Parentheses representation $S'_{rz}$ of the graph in Figure 3.7. In the string $S'_{rz}$, ( and ) represent the edges of $T_1'$, [ and ] represent the edges of $T_2'$, and { and } represent the edges of $T_3$. The parenthesis sequences $S_1'$, $S_2'$, $S_3'$ and the bit-vectors $B_1$ and $B_2$ obtained from $S'_{rz}$ are also shown.

- A “]” for each edge $(v'_i, v'_j)$ in $T_2'$, where $i < j$, right after $(j$.
- A “}” for each edge $(v'_i, v'_j)$ in $T_3$, where $i > j$, right after $(i$.
- A “{” for each edge $(v'_i, v'_j)$ in $T_3$, where $i > j$, right before $)$.

The relative order of the parentheses “[”, “]”, “{” and “}” inserted between two consecutive parentheses “(” and “)” does not matter. Figure 3.8 shows an example of the parenthesis sequence $S'_{rz}$ of the graph in Figure 3.7. Notice that the ccw preorder of $T_1'$, $v'_1, v'_2 \ldots, v'_n$, corresponds to another canonical ordering of $G$, which can be different to the canonical ordering used in the computation of the realizers.

In Section 6.2.1 we show how to compute the sequence $S'_{rz}$ in parallel, but using an equivalent definition of the construction of $S'_{rz}$. With the equivalent definition, the sequence $S'_{rz}$ is constructed as follows:

1. Classify all the edges of $G$ as part of $T_1$, $T_2$ or $T_3$. At the end of this stage, we will have the three spanning trees.
2. Perform an Euler tour over $T_1$ to define a new order among the vertices of $G$.
3. For each vertex $v_i$ of $G$, count its number of neighbors in $T_2$ that are lower, $l_{i}^{T_2}$, and higher, $h_{i}^{T_2}$, numbered. The same is done for neighbors in $T_3$.
4. Perform a new Euler tour over $T_1$. Each time that we visit a forward edge, write “(” followed by $l_{i}^{T_2}$ “]”s and $h_{i}^{T_3}$ “}”s. Each time that we visit a backward edge, write $h_{j}^{T_3}$ “{”s followed by $h_{i}^{T_2}$ “[”s and a “)”. The resulting parenthesis sequence is $S'_{rz}$.

The number of edges of $T_1'$ is $n$, of $T_2'$ is $n - 2$ and of $T_3$ is $n - 3$, therefore, the length of $S'_{rz}$ is $2(n + n - 2 + n - 3) = 2(3n - 5) = 2(m + 1)$.

To store $S'_{rz}$ succinctly, the authors in [7] propose to decompose $S'_{rz}$ into two bit-vectors, $B_1$ and $B_2$, and three parenthesis sequences, $S'_1$, $S'_2$ and $S'_3$. Let $n[]$, $n\|$ and
$n_{(1)}$ be the number of “(”, “[” and “{” respectively. The succinct representation of $S'_{rz}$ is constructed as follows:

- If $S'_{rz}[i]$ is a “(” or “)”, with $1 \leq i \leq n_{(0)} + n_{[0]} + n_{(1)}$, then $B_1[i] = 1$. Otherwise $B_1[i] = 0$.

- If the parenthesis associated to the $i$th 0 in $B_1$ is either a “[” or “]”, with $1 \leq i \leq n_{[0]} + n_{(1)}$, then $B_2[i] = 1$. Otherwise $B_2[i] = 0$.

- $S'_1$ corresponds to the subsequence of $S'_{rz}$ containing the parentheses “(” and “)”.

- $S'_2$ corresponds to the subsequence of $S'_{rz}$ containing the parentheses “[” and “]”.

- $S'_3$ corresponds to the subsequence of $S'_{rz}$ containing the parentheses “{” and “}”.

The bit-vectors $B_1$ and $B_2$ are stored as a rank/select structure in $2m \log 3 + o(m)$ bits, using the results of [110]. Parenthesis sequences $S'_1, S'_2$ and $S'_3$ are stored in $2m + o(m)$ bits, using the results of [97] and [102]. Thus, the total space used by the succinct representation of Barbay et al. is $2m \log 6 + o(m)$ bits.

This succinct representation supports the following operations:

- **adjacency**($v_i, v_j$): Whether vertices $v_i$ and $v_j$ are adjacent, with $i < j$. Vertices $v_i$ and $v_j$ are adjacent if and only if one is the parent of the other in one of the trees $T'_1, T'_2$ and $T'_3$. To check if $v_i$ is parent of $v_j$ in $T'_1$, we just need to check if $\text{level}_\text{anc}(v_j, 1)$ is equal to $v_i$, where $\text{level}_\text{anc}$ is defined in [102]. To test if $v_i$ is the parent of $v_j$ in $T'_2$, we need to check if the only outgoing edge of $v_j$, denoted by a “[”], is an incoming edge of $v_i$, denoted by a “[”. The case of $T_3$ is similar. Algorithm 4 shows how to answer the adjacency operations using the parenthesis sequences $S'_1, S'_2, S'_3$ and the bit-vectors $B_1$ and $B_2$. In Algorithm 4, $\text{first}_0(S'_k, i)$ is the position of the closest 0 after $S'_k[i]$, with $k \in \{1, 2, 3\}$, which can be computed in $O(1)$ with $\text{select}_0(S'_k, \text{rank}_0(S'_k, i) + 1)$.

- **degree**($v_i$): The degree of the vertex $v_i$. The answer is the sum of the degree of $v_i$ in $T'_1, T'_2$ and $T'_3$. The degree of $v_i$ in $T'_1$ can be obtained in constant time with the results of [102]. The degree in $T'_2$ and $T'_3$ can be computed with the formula $(o_{\text{first}} - o_{\text{left}} - 1) + (c_{\text{right}} - c_{\text{last}} - 1)$, where $o_{\text{left}}$ and $c_{\text{right}}$ correspond to the open and closed parentheses of the vertex $v_i$ in $S'_{rz}$, $o_{\text{first}}$ is the position of the open parenthesis in $S'_{rz}$ of the first child of $v_i$ in $T'_1$ and $c_{\text{last}}$ is the position of the closed parenthesis in $S'_{rz}$ of the last child of $v_i$ in $T'_1$. Algorithm 5 shows how to answer degree operations using parenthesis sequences $S'_1, S'_2, S'_3$ and bit-vectors $B_1$ and $B_2$. In the algorithm, $\text{last}_0(S'_k, i)$ is the position of the closest 0 before $S'_k[i]$, with $k \in \{1, 2, 3\}$, which can be computed in $O(1)$ with $\text{select}_0(S'_k, \text{rank}_0(S'_k, i) - 1)$. 
• **select_neighbor_ccw**(vᵢ, vⱼ, r): The r-th neighbor of vertex vᵢ starting from vertex vⱼ in ccw order if vᵢ and vⱼ are adjacent, and ∞ otherwise. To support this operation, it is necessary to use the local condition of realizers. The local condition indicates that, given a vertex vᵢ, its neighbors listed in ccw order form the following six types of vertices: vᵢ’s parent in T₁, vᵢ’s children in T₂, vᵢ’s parent in T₂', vᵢ’s children in T₂', vᵢ’s parent in T₃, and vᵢ’s children in T₃'. The constant time operations child(vᵢ, i) and child_rank(vⱼ) of [102] allow us to obtain the i-th child of vᵢ and the number of siblings before vⱼ in ccw order (where vⱼ is a neighbor of vᵢ) in T₁, respectively. Besides, the number of children of vᵢ in T₂ can be computed by counting the "["s right before the closed parenthesis of vᵢ and the number of children of vᵢ in T₃ can be computed by counting the "]"s right after the open parenthesis of vᵢ. Additionally, we can compute the number of each type of neighbors of vᵢ in constant time. Since all the previous operations are supported by rank/select operations over the parenthesis sequences S₁', S₂', S₃' and the bit-vectors B₁ and B₂, the operation select_neighbor_ccw(vᵢ, vⱼ, r) is supported in constant time.

• **rank_neighbor_ccw**(vᵢ, vⱼ, vₖ): The number of neighbors of vertex vᵢ between (and including) the vertices vⱼ and vₖ in ccw order if vⱼ and vₖ are both neighbors of vᵢ, and ∞ otherwise. This operation is supported in the same way as the select_neighbor_ccw(vᵢ, vⱼ, r) operation.

In this thesis, we will focus our efforts on the construction of succinct representations based on canonical orderings. Using the equivalence of canonical orderings and realizers, we will extend our results to succinct representation based on realizers.

### 3.3 Parallel Succinct Data Structures

At the time of this thesis, three succinct data structures have been studied for multicore machines: wtrees and rank/select structures. Based on work in [46], in [116], the author introduced two new algorithms to construct wtrees in parallel. The first algorithm, called levelWT, constructs the wtree level-by-level. In each level of the [lg σ] levels, the algorithm constructs the node and their bitmaps in parallel with O(n) work and O(lg n) span, which gives an algorithm of O(n lg σ) work and O(lg n lg σ) span, for an input sequence of size n and an alphabet of size σ. The second algorithm, called sortWT, constructs all levels in parallel, instead of one-by-one. For a level l, the sortWT algorithm applies a parallel stable integer sorting using the l most significant bits of each symbol as the key. With the sorted input sequence, the algorithm fills the corresponding bitarrays in parallel, using parallel prefix sum and filter algorithms to compute the position of the bits. The total work of the sortWT algorithm is O(W_sort lg σ), where W_sort is the work of the sorting algorithm, and the span is O(S_sort + lg n), where S_sort corresponds to the span of the sorting algorithm and the lg n component is the span of the prefix sum and filter algorithms. The author also discusses a variation of the sortWT algorithm, reaching O(n lg σ) work
Algorithm 4: Adjacency operation of the succinct representation of maximal plane graphs based on realizers.

Algorithm 5: Degree operation of the succinct representation of maximal plane graphs based on realizers.

and $O(\lg n \lg \sigma)$ span. In practice, the levelWT algorithm shows better performance. Compared to our previous algorithms in [46], the levelWT and sortWT algorithms can scale beyond $O(\lg \sigma)$ cores. However, both also need to duplicate and modify the input sequence, resulting in an increase in memory usage, requiring $O(n \lg n)$ bits of extra space. Most recently, in [83], two new algorithms to construct wtree s were proposed. The first algorithm, called recursiveWT, constructs the wtree recursively, performing parallel recursive calls to construct the two children of each node in the wtree. This algorithm is an optimization of the prwt algorithm in Section 4.1.1. The main problem with this algorithm is its dependency on the frequency of symbols. When few symbols have a high frequency, its scalability will be diminished. The second algorithm, ddWT, construct the wtree in a domain-decomposition fashion, constructing partial wtree s and then merge them into a final wtree. This algorithm is based on our domain-decomposition algorithm in [46]. Coincidentally, the ddWT algorithm improves our previous algorithm by improving the merge of the partial wavelet trees, in the same way that our new dd algorithm does (See Section 4.1.3).

---

Published after we sent our new wtree algorithms to revision.
In [116], the author introduces a technique to parallelize rank and select structures. The parallelised rank structure is based on that by Jacobson in [76]. The parallelization consists of three steps: first, the first and second level directories of Jacobson’s structure are built by performing a parallel prefix sum over the input bit-vector. Then the entries of the second level directory are packed into machine words. Finally the corresponding lookup tables are constructed in parallel. With respect to the select structure, the author shows how to construct the select structure of Clark [22]. The parallelization has three steps: First, the position of all the 1-bits are computed using a parallel prefix sum together with parallel filter algorithms [77]. Then, the second level directory is also computed using a prefix sum algorithm over the results of the first step to finally construct the lookup tables in parallel. In [83], the implementations of these parallel algorithms have speedups up to 38 for the rank structure and up to 14 for the select structure, with respect to themselves.

3.4 Libraries of Succinct Data Structures

Currently, there are only two libraries that implement succinct data structures: Libcds [23] and Sdsl [51]. Libcds has implementations for bit vectors supporting rank and select operations, trees and, in its second version, succinct trees. Sdsl supports the same structures that Libcds and additionally implements compressed suffix trees, compressed suffix arrays, longest common prefix arrays and range minimum/maximum query structures. Both libraries implement the data structures in C++ and are available in the Web. Although both libraries are the best available implementations of succinct data structures, none of them have implementations of such data structures for parallel machines, in particular, for SMP systems.

In this thesis, both Libcds and Sdsl will be used has baseline. In this manner, the speed up for each succinct data structures implemented in this thesis will be calculated considering the fastest current implementation of such data structures, either Libcds or Sdsl.
Chapter 4

Parallel Construction of Wavelet Trees

4.1 Parallel Construction

As was mentioned in Section 3.2.1, we focus on binary \textit{wtrees}, where the symbols in \( \Sigma \) are contiguous in \([1, \sigma]\). Under these restrictions, the \textit{wtree} is a balanced binary tree with \( \lg \sigma \) levels. We will build the representation of \textit{wtrees} that removes the \( O(\sigma \lg n) \) bits of the topology. Hence, when we refer to a \textit{node}, this is a conceptual node that does not exist in the actual implementation of the data structure.

In what follows, two iterative construction algorithms are introduced that capitalize on the idea that any level of the \textit{wtree} can be built independently from the others. Unlike in classical \textit{wtree} construction, when building a level we cannot assume that any previous step is providing us with the correct permutation of the elements of \( S \). Instead, we compute the node at level \( i \) for each symbol of the original sequence. More formally,

\textbf{Proposition 1.} Given a symbol \( s \in S \) and a level \( i, 0 \leq i < l = \lceil \lg \sigma \rceil \), of a \textit{wtree}, the node at which \( s \) is represented at level \( i \) can be computed as \( s \gg l - i \).

In other words, if the symbols of \( \Sigma \) are contiguous, then the \( i \) most significant bits of the symbol \( s \) gives us its corresponding node at level \( i \). In the word-RAM model with word size \( \Omega(\lg n) \), this computation takes \( O(1) \) time, and thus the following corollary holds:

\textbf{Corollary 1.} The node at which a symbol \( s \) is represented at level \( i \) can be computed in \( O(1) \) time.

4.1.1 Parallel recursive algorithm

Before introduce our iterative algorithms, we present a naïve parallel recursive algorithm, based on the simplest sequential algorithm (see Section 3.2.1). On its sequential version, the recursive algorithm works by halving \( \Sigma \) recursively into binary sub-trees whose left child are all 0s and the right all 1s, until 1s and 0s mean only one symbol in \( \Sigma \). We parallelized it by the technique of spawning one task for each recursive call except the last, while doing the latter on the calling thread [86]. In our case, we spawn the left sub-tree to continue working on the right sub-tree.

The algorithm, called \texttt{prwt}, is shown in Algorithm 6. The algorithm takes as input a sequence of symbols \( S \), the length \( n \) of \( S \), and the length of the alphabet, \( \sigma \). The output is a \textit{wtree} \( WT \) that represents \( S \). We denote the \( i \)th level of \( WT \) as \( WT[i], \forall i, 0 \leq i < \lceil \lg \sigma \rceil \).
Input: $S, n, \sigma$

Output: A wavelet tree representation $WT$ of $S$

1. $WT$ is a new wavelet tree with $\lceil \lg \sigma \rceil$ levels
2. $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
3. $\text{createNode}(S, n, B, 0, 0, \lceil \lg \sigma \rceil)$

```
parfor i = 0 to $\lceil \lg \sigma \rceil - 1$
do

| $WT[i] = \text{createRankSelect}(B[i])$

return $WT$
```

Algorithm 6: Parallel recursive algorithm ($\text{prwt}$)

The first step of $\text{prwt}$ (lines 1 and 2) allocates memory for the output $\text{wtree}$ and its bitarrays, $B$. After that, the algorithm calls the Function $\text{createNode}$ whom sets, recursively, the bitarrays of the $\text{wtree}$ (line 3). Finally, in lines 4-5, the algorithm creates the rank/select structures for each level of the $\text{wtree}$.

The Function $\text{createNode}$ performs the major part of the work. Its input corresponds to a sequence of symbols $S$, the length $n$ of $S$, the array of bitarrays of the $\text{wtree}$, $B$, the current level, $lvl$, the offset from which the function should start to write on the bitarray $B[lvl]$, and the number of levels of the $\text{wtree}$, $\text{levels}$. The initial call of the function creates the bitarray of the first level of the $\text{wtree}$ (see line 3 of Algorithm 6), therefore, $S$ corresponds to the original input sequence and $lvl$ and $offset$ are 0. On each call, the function generate two new sequences, $S_{\text{left}}$ and $S_{\text{right}}$, from $S$. The size of $S_{\text{left}}$ and $S_{\text{right}}$ is computed by scanning $S$ (lines 4 to 8). In lines 9 and 10, the function allocates memory for the new sequences. Then, the function sets the bits of $B[lvl]$ and the symbols of $S_{\text{left}}$ and $S_{\text{right}}$, sequentially scanning $S$ (lines 12 to 20). For each symbol in $S$, the function computes if the symbol belongs to either the first or second half of $\Sigma$ for the current node. Notice that each call of the function represents the computation of one (virtual) node of the $\text{wtree}$. If the symbol belongs to the first half of $\Sigma$ assigned to the node, the bit at position $\text{offset} + i$ of $B[lvl]$ is set to 0 using $\text{bitmapSetBit}$, and the symbol $S[i]$ is copied into $S_{\text{left}}$ at position $\text{llen}$ (lines 17 to 20). Similarly, if the symbol belongs to the second half, the bit at position $\text{offset} + i$ of $B[lvl]$ is set to 1 and the symbol $S[i]$ is copied into $S_{\text{right}}$ at position $\text{rlen}$ (lines 12 to 16). Once the new sequences and the bitarray at level $lvl$ are computed, the memory allocated to store $S$ can be released in order to reduce the working space (line 21). Finally, we recursively call the function with the new sequences. For example, at line 22, the function creates a parallel recursive task to compute a new node, with $S_{\text{left}}$ as input.

This new parallel task is pushed into the bottom of the deque of the calling thread. Meanwhile, the calling thread performs a recursive call with $S_{\text{right}}$ as input (line 23). All parallel recursive calls are synchronized at line 24.

Under the DYM model, the DAG of the $\text{prwt}$ is weighted. Not all strands in this DAG have the same weight: the frequency of symbols is not the same. All paths are the same length; that is, $O(\lg \sigma)$, the critical path will be given by the weight of
Input: $S$, $n$, $B$, $lvl$, $offset$, $levels$
Output: Setting of the array of bitarrays $B$

1. if $lvl == levels$ then
2.  return // last level
3. llen = 0, rlen = 0
4. for $i = 0$ to $n$ do
5.  if $(S[j] \& 2^{\lceil \log_2 \sigma \rceil - i - 1}) == 1$ then
6.    increment(rlen)
7.  else
8.    increment(llen)
9. $S_{left}$ is an array of $llen$ symbols
10. $S_{right}$ is an array of $rlen$ symbols
11. llen = 0, rlen = 0
12. for $i = 0$ to $n$ do
13.  if $(S[j] \& 2^{\lceil \log_2 \sigma \rceil - i - 1}) == 1$ then
14.    bitmapSetBit($B[lvl]$, $offset+i$, 1)
15.    $S_{right}[rlen] = S[i]$
16.    increment(rlen)
17.  else
18.    bitmapSetBit($B[lvl]$, $offset+i$, 0)
19.    $S_{left}[llen] = S[i]$
20.    increment(llen)
21. release $S$
22. spawn $createNode(S_{left}, llen, B, lvl+1, offset, levels)$
23. $createNode(S_{right}, rlen, B, lvl+1, llen+offset, levels)$
24. sync

Function $createNode$

the heaviest path in the DAG. In the worst case, where one branch always contains most of $S$, $T_\infty = O(n \log \sigma)$. This is the case, for example, when $\Sigma$ is ordered by frequency. In the best case, when all symbols in $\Sigma$ have exactly the same frequency, then $T_\infty = O(n)$. Finally, the parallelism for the worst case of $prwt$ is $T_1/T_\infty = O(1)$, which is no parallelism at all. In turn, in the best case, we have that the parallelism is $O(\log \sigma)$, which means that the algorithm scales on $\sigma$.

The working space needed by $prwt$ is limited by the space needed for the $wtree$ and the new sequences $S_{left}$ and $S_{right}$. Since on each call to the Function $createNode$ the input $S$ is released, the working space is $O(n \log \sigma)$ bits.

4.1.2 Per-level parallel algorithm

Our second algorithm, called $pwt$, is shown in Algorithm 7. The algorithm takes as input a sequence of symbols $S$, the length $n$ of $S$, and the length of the alphabet, $\sigma$. The output is a $wtree$ $WT$ that represents $S$. We denote the $i$th level of $WT$ as $WT[i]$, $\forall i, 0 \leq i < \lceil \log \sigma \rceil$. 
Input: \( S, n, \sigma \)

Output: A wavelet tree representation \( WT \) of \( S \)

\[ WT \text{ is a new wavelet tree with } \lceil \lg \sigma \rceil \text{ levels} \]

\begin{verbatim}
parfor i = 0 to \( \lceil \lg \sigma \rceil - 1 \) do
    B is a bitarray of size \( n \)
    C is an integer array of size \( 2^i \)
    for j = 0 to \( n - 1 \) do
        increment(\( C[S[j]/2^\lceil \lg \sigma \rceil - i] \))
        parPrefixSum(\( C \))
    for j = 0 to \( n - 1 \) do
        if \((S[j] & 2^{\lceil \lg \sigma \rceil - i - 1}) == 1\) then
            bitmapSetBit(B, \( C[S[j]/2^\lceil \lg \sigma \rceil - i] \), 1)
        else
            bitmapSetBit(B, \( C[S[j]/2^\lceil \lg \sigma \rceil - i] \), 0)
        increment(\( C[S[j]/2^\lceil \lg \sigma \rceil - i] \))
    \( WT[i] = \) createRankSelect(B)
\end{verbatim}

Algorithm 7: Per-level parallel algorithm (\texttt{pwt})

The outer loop (line 2) iterates in parallel over \( \lceil \lg \sigma \rceil \) levels. Lines 3 to 14 scan each level performing the following tasks: the first step (lines 3 and 4) initializes the bitmap \( B \) of the \( i \)th level and initializes an array of integers \( C \). The array \( C \) will be used to count the number of bits in each node of the \( wtree \) at level \( i \), using counting sort. The second step (lines 5 and 6) computes the size of each node in the \( i \)th level performing a linear-time sweep over \( S \). For each symbol in \( S \), the algorithm computes the corresponding node for alphabet range at the current level. The expression \( S[j]/2^{\lceil \lg \sigma \rceil - i} \) in line 6 shows an equivalent representation of the idea in Proposition 1. The third step performs a parallel prefix sum algorithm [65] over the array \( C \), obtaining the offset of each node. Once the offset of the nodes is known, the algorithm constructs the corresponding bitarray \( B \), sequentially scanning \( S \) (lines 8 to 13). For each symbol in \( S \), the algorithm computes the corresponding node and whether the symbol belongs to either the first or second half of \( \Sigma \) for that node. The corresponding bit is set using \texttt{bitmapSetBit} at position \( C[S[j]/2^{\lceil \lg \sigma \rceil - i}] \). Line 14 creates the rank/select structures of the bitmap \( B \) of the \( i \)th level.

Figure 4.1 shows a snapshot of the execution of the \texttt{pwt} for the input sequence of Figure 3.1: the levels of the \( wtree \) can be constructed in different threads asynchronously.

The work \( T_1 \) of this algorithm takes \( O(n \lg \sigma) \) time. This matches the time for construction found in the literature. Each of the \( \lg \sigma \) tasks that create the \texttt{pwt} algorithm has a complexity of \( O(n + \sigma/p + \lg p) \), due to the scans over the input sequence and the parallel prefix sum over the array \( C \). The work of \texttt{pwt} is still \( T_1 = O(n \lg \sigma) \). Since all tasks have the same complexity, assuming constant access to any position in memory, the critical path is given by the construction of one level of
Figure 4.1: Snapshot of an execution of the algorithm \texttt{pwt} for the sequence introduced in Figure 3.1. In the snapshot, thread \texttt{t1} is writing the first bit of the symbol \( S[10] = \text{'a'} \) at level 0, thread \texttt{t2} is writing the second bit of \( S[15] = \text{'e'} \) at level 1, thread \texttt{t3} is writing the third bit of \( S[19] = \text{'P'} \) at level 2 and thread \texttt{t4} is writing the fourth bit of \( S[26] = \text{'d'} \) at level 3. Black areas represent bits associated to unprocessed symbols.

The main drawback of the \texttt{pwt} algorithm is that it only scales until the number of cores equals the number of levels in the wavelet tree. So, even if we have more cores available, the algorithm will only use up to \( \lg \sigma \) cores. Nevertheless, this algorithm is simple to implement, suitable in domains where it is not possible to use all available resources to the construction of \texttt{wtrees}.

4.1.3 Domain decomposition parallel algorithm

The third algorithm that we propose makes efficient use of all available cores. The main idea of the algorithm is to divide the input sequence \( S \) in \( k = O(p/\lg(\sigma)) \) segments of size \( O(n/k) \) and then apply the \texttt{pwt} algorithm on each segment, generating \( O(\lg \sigma) \) tasks per segment and creating \( k \) partial \texttt{wtrees}. After that, the algorithm merges all the partial \texttt{wtrees} into a single one that represents the entire input text. We call this algorithm \texttt{dd} because of its domain decomposition nature.

The \texttt{dd} algorithm is shown in Algorithm 8. It takes the same input as \texttt{pwt} with the addition of the number of segments, \( k \). The output is a \texttt{wtree} \( WT \), which represents the input data \( S \).

The first step of \texttt{dd} (lines 1 to 4) allocates memory for the output \texttt{wtree}, its
Input : \( S, n, \sigma, k \)
Output: A wavelet tree representation \( WT \) of \( S \)

1. \( WT \) is a new tree with \( \lceil \lg \sigma \rceil \) levels
2. \( B \) is an array of \( \lceil \lg \sigma \rceil \) bitarrays of size \( n \)
3. \( pB \) is a bidimensional array of bitarrays of dimensions \( k \times \lceil \lg \sigma \rceil \)
4. \( G, L \) are tridimensional arrays of integers of dimensions \( k \times \lceil \lg \sigma \rceil \times 2^{\text{level}} \)

5. \textbf{parfor} \( i = 0 \) to \( k - 1 \) do
6. \hspace{1em} \( pB[i] = \text{createPartialBA}(S, \sigma, i, n/k) \)
7. \textbf{parfor} \( i = 0 \) to \( \lfloor \lg \sigma \rfloor - 1 \) do
8. \hspace{1em} \text{parPrefixSum}(i, k)
9. \hspace{1em} \( B = \text{mergeBA}(n, \sigma, k, pB) \)
10. \hspace{1em} \textbf{parfor} \( i = 0 \) to \( \lfloor \lg \sigma \rfloor - 1 \) do
11. \hspace{2em} \( WT[i] = \text{createRankSelect}(B[i]) \)
12. return \( WT \)

Algorithm 8: Domain decomposition parallel algorithm (dd)

bitarrays, \( B \), the bitarrays of the partial \( \text{wtrees} \), \( pB \), and two 3-dimensional arrays of numbers, \( L \) and \( G \), where the third dimension changes according to the number of nodes in each level. Arrays \( L \) and \( G \) store local and global offsets, respectively. The local offsets store the offsets of all the nodes of the partial \( \text{wtrees} \) with respect to the partial \( \text{wtree} \) containing them. Similarly, \( G \) stores the offsets of all the nodes of the partial \( \text{wtrees} \) with respect to the final \( \text{wtree} \). In other words, each entry \( L[a][b][c] \) stores the position of node \( c \) at level \( b \) whose parent is partial \( \text{wtree} \) \( a \). Each entry \( G[a][b][c] \) stores the position of node \( c \) at level \( b \) in the partial \( \text{wtree} \) \( a \) inside the final \( \text{wtree} \). We will treat the arrays \( L \) and \( G \) as global variables to simplify the pseudocode.

The second step (lines 5 and 6) computes the partial \( \text{wtrees} \) of the \( k \) segments in parallel. For each segment, \( \text{createPartialBA} \) is called to create the partial \( \text{wtree} \). This function is similar to the one in the \( \text{pwt} \) algorithm, performing a prefix sum (line 5 in Function \( \text{createPartialBA} \)) to compute the local offsets and store them both in \( G \) and \( L \). We reuse the array \( G \) to save memory in the next step. Notice that the output of the function is a partial \( \text{wtree} \) composed of \( \lceil \lg \sigma \rceil \) bitarrays, without rank/select structures over such bitarrays.

The third step of the dd algorithm uses the local offsets stored in \( L \) to compute the global ones (lines 7 and 8). To do that, at each level \( i \), the algorithm applies a parallel prefix sum algorithm using the \( k \) local offsets of that level. The prefix sum algorithm uses the implicit total order within the local offsets. Since each level in the offsets is independent of the others, we can apply the \( \lceil \lg \sigma \rceil \) calls of the parallel prefix sum algorithm in parallel.

Once we have the global offsets computed, the fourth step merges all partial \( \text{wtrees} \), in parallel. Function \( \text{mergeBA} \) creates one parallel task for each node in the partial \( \text{wtrees} \). In each parallel task (lines 5 to 10) the function concatenates the bitarray of the node \( m/k \) of the \( i \)th level of the \( m/k \) partial \( \text{wtrees} \) into the corresponding bitarray, \( B[i] \), of the final \( \text{wtree} \). Using the local and the global offsets,
Input: $S, \sigma, k', n$
Output: A bitarray representation $B$ of the $k'$th segment of $S$

1. $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
2. parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$
   3. for $j = n \times k'$ to $n \times (k' + 1) - 1$
      4. increment($G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}])$
      5. prefixSum($G, L$)
   6. for $j = n \times k'$ to $n \times (k' + 1) - 1$
      7. if $(S[j] \& 2^{\lceil \lg \sigma \rceil - i}) == 1$
         8. bitmapSetBit($B, G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}], 1$)
      else
         9. bitmapSetBit($B, G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}], 0$)
      10. increment($G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}]$)
11. return $B$

Function createPartialBA

The function parallelBitarrayConcat copies $nb$ bits of $pB[i]$, starting at position $L[m\%k][i][m/k]$ into the bitarray $B[i]$ at position $G[m\%k][i][m/k]$. The function parallelBitarrayConcat is thread-safe: the first and last machine words that compose each bitarray are copied using atomic operations. Thus, the concatenated bitarrays are correct regardless of multiple concurrent concatenations. The last step, lines 10-11, creates the rank/select structures for each level of the wtree.

For an example of the algorithm, see Figure 4.2. Figure 4.2a shows a snapshot of the function createPartialBA and Figure 4.2b shows a snapshot of mergeBA.

The dd algorithm has the same asymptotic complexity as pwt, with work $T_1 = O(n \lg \sigma)$. When running on $p$ cores and dividing $S$ in $k = O(p/\lg \sigma)$ segments, the construction of the partial wtrees takes $O(n \lg \sigma/p)$ time. The prefix sum takes $O(\lg \sigma/p \times (\sigma/\lg \sigma + \lg p))$ time [65]. Merge takes $O(n \lg \sigma/pw)$, where $w$ is the word size of that architecture. The overhead of the parfor is $O(\lg p + \lg \sigma \lg \lg \sigma)$. For $p = \infty$, the span of the construction of the partial wtrees is $O(1)$, $O(\lg (k\sigma))$ for the prefix sum section and $O(1)$ for the merge function. In the case of the merge function, the offsets of the bitarrays have been previously computed and each bit can be copied in parallel. Thus, considering $w$ as a constant and $k = O(p/\lg \sigma)$, the span is $T_\infty = O(\lg n)$ in all cases.

The working space needed by dd is limited by the space needed for the wtree, the partial wtrees, and local and global offsets, totalling $O(n \lg \sigma + k\sigma \lg n)$ bits. By manipulating the value of $k$, however, we can reduce the needed space or improve the performance of dd algorithm. If $k = 1$, then space is reduced to $O(n \lg \sigma + \sigma \lg n)$ bits, but this limits scalability to $p < \lg \sigma$. If $k = p$, we improve the time complexity, at the cost of $O(n \lg \sigma + p\sigma \lg n)$ bits.
(a) Snapshot of Function `createPartialBA`. The figure shows the construction of the partial `wtrees` after the split of the input sequence introduced in Figure 3.1 into three subsequences. To create each partial `wtree`, the algorithm uses the `pwt` algorithm. These partial `wtrees` are the input of Function `mergeBA`.

(b) Snapshot of the Function `mergeBA`. White, light gray and dark gray bitarrays represent the bitarrays of first, second and third partial `wtrees`, respectively. The positions of the partial `wtrees` bitarrays are computed in advance. Therefore such bitarrays can be copied to the final `wtree` in parallel. Black areas represent uncopied bits.

**Figure 4.2:** Snapshot of an execution of the algorithm `dd`. Figures 4.2a and 4.2b represent snapshots of Functions `createPartialBA` and `mergeBA`, respectively. The result of this example is the `wtree` of Figure 3.1a.
Input: $n$, $\sigma$, $k$, $pB$

Output: A bitarray representation $B$ of the input sequence $S$

1. $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
2. parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
3.   parfor $j = 0$ to $k - 1$ do
4.      parfor $m = j \times 2^i$ to $(j + 1) \times 2^i$ do
5.         $dst = B[i]$ // Destination of the bits to be copied
6.         $src = pB[m \% k][i]$ // Source of the bits to be copied
7.         $go = G[m \% k][i][m/k]$ // Offset in $dst$
8.         $lo = L[m \% k][i][m/k]$ // Offset in $src$
9.         $nb = L[m \% k][i][m/k + 1] - L[m \% k][i][m/k]$ // Number of bits
10.    parallelBitarrayConcat($dest, src, go, lo, nb$)
11. return $B$

Function mergeBA

4.2 Parallel Querying

We also consider the problem of answering, in parallel, a batch of queries. We distinguish between two kinds of queries on $w$trees: path and branch queries. Path queries are characterized by following just a single path from the root to a leaf and the value in level $i - 1$ has to be computed before the value in level $i$. Examples of this type of queries are select, rank, and access. On the other hand, branch queries may follow more than one path root-to-leaf (indeed they may reach more than one leaf). Each path has the same characteristics as path queries and each path is independent from others paths. Examples of this type of queries are range count and range report [49].

Given $1 \leq i \leq i' \leq n$ and $1 \leq j \leq j' \leq \sigma$, a range report query $rq(S, i, i', j, j')$ reports all the symbols $s_x$ such that $x \in [i, i']$ and $s_x \in [j, j']$. The counting version of the problem can be defined analogously.

In a parallel setting, a single path query cannot be parallelized because only one level of the query can be computed at a time. The common alternative is parallelizing several path queries using domain decomposition over queries (i.e., dividing queries over $p$). For this naïve approach, we obtained near-optimal throughput, defined as the number of cores times sequential throughput (see Section 4.3.3).

For branch queries, we implemented two techniques: individual-query-answering (IQA) and batch-query-answering (BQA). The IQA technique is the obvious query by query processing. The BQA technique involves grouping sets of queries to take advantage of spatial and temporal locality in hierarchical memory architectures. For instance, at each node in the $w$tree, we can evaluate all the queries in a batch reusing the node’s bitarray, thus increasing locality.

With little effort, we can parallelize sequential IQA in a domain decomposition fashion (denoted as dd-IQA), achieving near-optimal throughput (more than 10 times the throughput for $p = 12$ compared to the sequential IQA).

The parallelization of the BQA technique is shown in Algorithm 9 (denoted as
**Input**: \( WT, n, \sigma, \text{queries}, \text{num\_queries}, \text{batch\_size} \)

**Output**: \( \text{results} \), an array containing the results for each query

1. \( num\_batches = \text{num\_queries} / \text{batch\_size} \)

2. \( results \) is an array of size \( num\_batches \). It will store the result of each query.

3. \( nd \) is a structure that store information about the node that is being processed.

4. \textbf{parfor} \( i = 0 \) to \( num\_batches - 1 \) do

5. \( states \) is an array of size \( batch\_size \). It will mark if a query is already finished. Initially, all queries are marked as unfinished.

6. \( nd.S_l = 0, \ nd.S_r = n - 1 \) \hspace{1cm} // The root of \( WT \)

7. \( nd.\sigma_l = 0, \ nd.\sigma_r = \sigma - 1, \ nd.lvl = 0 \)

8. \( \text{batchRangeCount}(WT, \sigma, n, \text{queries}, nd, \text{results}, states, batch\_size \times i, batch\_size) \)

9. return \( results \)

**Algorithm 9**: Parallel batch querying of range report (parBQA)

parBQA). The input of the algorithm corresponds to a \textit{wtree}, the size of the sequence \( S \) represented by the \textit{wtree}, \( n \), the number of symbols of such sequence, \( \sigma \), an array with the branch queries, \textit{queries}, the number of queries, \textit{num\_queries}, and the number of queries on each batch, \textit{batch\_size}. Each branch query is represented by a structure with four fields. For a branch query \( q \), the range of interest in \( S \) is given by \([q.S_l, q.S_r]\) and the range of interest in \( \Sigma \) is given by \([q.\sigma_l, q.\sigma_r]\). The algorithm iterates over the batches of queries (lines 4 to 8). For each batch, it starts on the root of the \textit{wtree} (lines 6 and 7) and then calls the recursive Function \textit{batchRangeCount} (line 8). Each recursive call of the function corresponds to the processing of a virtual node of the \textit{wtree}, whose limits are defined on \( nd \). On each node, the function iterates over all the queries in the batch, reusing the bitarray associated to the current node (lines 6 to 26 of Function \textit{batchRangeCount}). Since we implement the \textit{wtree} using one bitarray per level, we need to compute an \textit{offset} to obtain the correct answers. After iterating over all the queries, the function computes the limits of the two children of the current node (lines 27 to 30) and makes the recursive calls (lines 31 and 32). To mark when a query is finished on a branch, the function takes as argument an array called \textit{states} which stores the state of each query. If a query \( q \) was already finished in a previous call, then \textit{stated}[q] is 1; otherwise it is 0 (lines 7 to 9). If the query is finished on the current call, then the state of the query \( q \) is changed to 1 (lines 10 to 20). Once the states of all queries are changed to 1, the function halts (lines 25 and 26) and the result of each query is store in the array \textit{results}.

Observe that the \textbf{parfor} in line 4 of the Algorithm 9 exploits the spatial and temporal locality of the node’s bitarrays and the \textbf{spawn} and \textbf{sync} in lines 31 and 33 of the Function \textit{batchRangeCount} exploit the independent paths of the branch query. With this, we obtain a parallel querying algorithm with \( O(q \lg \sigma) \) work and \( O(\lg \sigma) \) span, where \( q \) is the number of branch queries. This algorithm can be applied directly both to \textit{range count} and \textit{range report}. Finally, note that the algorithm can be modified to apply batch processing to path queries.
Input: $WT, \sigma, n, queries, nd, results, states, init, num\_queries$
Output: A bitarray representation $B$ of the input sequence $S$

1. $queries_l$ is an array of $num\_queries$ queries
2. $local\_states$ is a copy of the array $states$
3. $finished = 0$
4. $offset = rank_0(WT[nd.lvl], nd.S_l - 1)$
5. $zeros = rank_0(WT[nd.lvl], nd.S_2) - offset$
6. for $q = init$ to $init + num\_queries$ do
   7. if $local\_states[q] == 1$ then
      8. increment($finished$) // The query is already finished
      9. continue
   10. if $queries[q].\sigma_l > queries[q].\sigma_r$ then
        11. $local\_states[q] = 1$, increment($finished$)
        12. continue
   13. if $queries[q].\sigma_r < nd.\sigma_l \lor queries[q].\sigma_l > nd.\sigma_r$ then
        14. $local\_states[q] = 1$, increment($finished$)
        15. continue
   16. // The current node contains completely the range of interest
   17. // The whole range $[nd.lim_l, nd.lim_r]$ is part of the answer
   18. if $queries[q].\sigma_l \leq nd.\sigma_l \land queries[q].\sigma_r \geq nd.\sigma_r$ then
        19. $local\_states[q] = 1$
        20. increment($finished$)
        21. $results[q] += queries[q].S_r - queries[q].S_l + 1$
        22. continue
   23. $queries_l[q].S_l = rank_0(WT[nd.lvl], queries[q].S_l - 1) - offset + nd.lim_l$
   24. $queries_l[q].S_r = rank_0(WT[nd.lvl], queries[q].S_r) - offset + nd.lim_r$
   25. $queries[q].S_l = queries[q].S_l - queries[q].S_l + zeros + nd.S_l$
   26. $queries[q].S_r = queries[q].S_r - queries[q].S_r + 1 + zeros + nd.S_r$
   27. if $finished == num\_queries$ then
      28. return
   29. $nd_l$ is a copy of $nd$
   30. $h_\sigma = (nd.\sigma_l + nd.\sigma_l)/2$
   31. $nd_l.S_r = nd.S_r + zeros - 1$, $nd_l.\sigma_r = half_\sigma$, $nd_l.lvl ++$
   32. $nd.S_l = nd.S_l + zeros$, $nd.\sigma_l = half_\sigma + 1$, $nd.lvl ++$
   33. spawn $batchRangeCount(WT, \sigma, n, queries_l, nd_l, results, local\_states, init, batch\_size)$
   34. $batchRangeCount(WT, \sigma, n, queries, nd, results, local\_states, init, batch\_size)$
   35. sync
   36. release $local\_states$
   37. release $queries_l$

   Function $batchRangeCount$
4.3 Experiments

Construction experiments were carried out in the machine B and querying experiments were carried out in the machine A.

4.3.1 Experimental setup

The experimental trials consisted of running the algorithms on datasets of different alphabet sizes, input sizes $n$ and number of cores. The datasets are shown in Table 4.1. We distinguish between two types of datasets: those in which each symbol is encoded using 1 byte, and those in which each symbol is encoded using 4 bytes. Datasets 1-10 in Table 4.1 with $\sigma \leq 256$ were encoded using 1 byte per symbol. Datasets 11-14 were encoded using 4 bytes. Datasets 15-18 were encoded as follows: for $x = \{4, 6, 8\}$, each symbol was encoded with a single byte. For $x = \{10, 12, 14\}$, each symbol was encoded in four bytes. The dataset rna.13GB is the GenBank mRNAs of the University of California, Santa Cruz\(^1\). The rest of the rna datasets were generated by splitting the previous one. We also tested datasets of protein sequences, prot\(^2\) and source code, src.200MB\(^3\). We also built a version of the source code dataset using words as symbols, src.98MB. The rest of the src datasets were generated by concatenating the previous one up to a maximum of 2GB. To measure the impact of varying the alphabet size, we took the English corpus of the Pizza & Chili website\(^4\) as a sequence of words and filtered the number of different symbols in the dataset. The dataset had an initial alphabet $\Sigma$ of $\sigma = 633,816$ symbols. For experimentation, we generated an alphabet $\Sigma'$ of size $2^x$, taking the top $2^x$ most frequent words in the original $\Sigma$, and then assigning a random index to each symbol using a Marsenne Twister \([91]\), with $x \in \{4, 6, 8, 10, 12, 14\}$. To create an input sequence $S$ of $n$ symbols for the English dataset (en), we searched for each symbol in $\Sigma'$ in the original English text and, when found, appended it to $S$ until it reached the maximum possible size given $\sigma'$ (\(~1.5\)GB, in the case of $\sigma' = 2^{18}$), maintaining the order of the original English text. We then either split $S$ until we reached the target size $n = 2^{27}$ or concatenated $S$ with initial sub-sequences of itself to reach the larger sizes $2^{28}, 2^{29}$ and $2^{30}$. We repeated each trial five times and recorded the median time \([122]\).

4.3.2 Construction Experiments

We tested the implementation of our parallel wavelet tree construction algorithms considering one pointer per level and without considering the construction time of

\(^1\)http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/xenoMrna.fa.gz (April, 2015)
\(^2\)http://pizzachili.dcc.uchile.cl/texts/protein/proteins.gz (April, 2015)
\(^3\)http://pizzachili.dcc.uchile.cl/texts/code/sources.gz (April, 2015)
\(^4\)http://pizzachili.dcc.uchile.cl/texts/nlang/english.1024MB.gz (March, 2013)
Table 4.1: Datasets used in the experiments of \textit{wtrees}.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>(n)</th>
<th>(\sigma)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 rna.512MB</td>
<td>536,870,912</td>
<td>4</td>
</tr>
<tr>
<td>2 rna.1GB</td>
<td>1,073,741,824</td>
<td>4</td>
</tr>
<tr>
<td>3 rna.2GB</td>
<td>2,147,483,648</td>
<td>4</td>
</tr>
<tr>
<td>4 rna.3GB</td>
<td>3,221,225,472</td>
<td>4</td>
</tr>
<tr>
<td>5 rna.4GB</td>
<td>4,294,967,296</td>
<td>4</td>
</tr>
<tr>
<td>6 rna.5GB</td>
<td>5,368,709,120</td>
<td>4</td>
</tr>
<tr>
<td>7 rna.6GB</td>
<td>6,442,450,944</td>
<td>4</td>
</tr>
<tr>
<td>8 rna.13GB</td>
<td>14,570,010,837</td>
<td>4</td>
</tr>
<tr>
<td>9 prot</td>
<td>1,184,051,855</td>
<td>27</td>
</tr>
<tr>
<td>10 src.200MB</td>
<td>210,866,607</td>
<td>230</td>
</tr>
<tr>
<td>11 src.98MB</td>
<td>25,910,717</td>
<td>2,446,383</td>
</tr>
<tr>
<td>12 src.512MB</td>
<td>134,217,728</td>
<td>2,446,383</td>
</tr>
<tr>
<td>13 src.1GB</td>
<td>268,435,455</td>
<td>2,446,383</td>
</tr>
<tr>
<td>14 src.2GB</td>
<td>536,870,911</td>
<td>2,446,383</td>
</tr>
<tr>
<td>15 en.x.27</td>
<td>134,217,728</td>
<td>2(^x)</td>
</tr>
<tr>
<td>16 en.x.28</td>
<td>268,435,456</td>
<td>2(^x)</td>
</tr>
<tr>
<td>17 en.x.29</td>
<td>536,870,912</td>
<td>2(^x)</td>
</tr>
<tr>
<td>18 en.x.30</td>
<td>1,073,741,824</td>
<td>2(^x)</td>
</tr>
</tbody>
</table>

**Running times and speedup.** Table 4.2 shows the running times of all tested algorithms. \texttt{Libcds} and \texttt{shun} work just for \(n < 2^{32}\), so we cannot report running times of these algorithms for the datasets \texttt{rna.4GB}, \texttt{rna.5GB}, \texttt{rna.6GB} and \texttt{rna.13GB}.

For each dataset, we underline the best sequential running times. We use those values to compute speedups. The best parallel times for \(p = 64\) are identified using a bold typeface. Although \texttt{Libcds} and \texttt{SDSL} are the state-of-the-art in sequential implementations of \textit{wtrees}, the best sequential running times were obtained from the parallel implementations running on one thread. The main reason for this is that \texttt{SDSL} implements a semi-external algorithm for \textit{wtree} construction, involving

\(^5\)We also tested a new version of \texttt{Libcds} called \texttt{Libcds2}. However the former had better running times for the construction of \textit{wtrees}. 

rank/select structures. We compared our algorithms against \texttt{Libcds}\(^5\) and \texttt{SDSL}. Both libraries were compiled with their default options and the -O2 optimization flag. With regards to the bitarray implementation, we use the 5%-extra space structure presented in [54] (as \texttt{Libcds} does). For \texttt{SDSL} we use the \texttt{bit\_vector} implementation with settings \texttt{rank\_support\_scan<1>}, \texttt{select\_support\_scan<1>} and \texttt{select\_support\_scan<0>} to skip construction time of rank/select structures. In our experiments, \texttt{shun} is the fastest of the three algorithms introduced in [116], compiled also with the -O2 optimization flag. Our \texttt{dd} algorithm was tested with \(k = p\) privileging time performance over memory.
In the case of one thread, that thread has to build the 14 levels. In the case of 4 datasets, with the largest shown using bold typeface. A "-" is shown for implementations that just work for and 64 threads. The best sequential times are underlined and the best parallel times are shown using bold typeface. A "-" is shown for implementations that just work for

<table>
<thead>
<tr>
<th>Datasets</th>
<th>libcds</th>
<th>sdsl</th>
<th>prwt</th>
<th>put</th>
<th>dd</th>
<th>shun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src.2GB</td>
<td>150.67</td>
<td>205.41</td>
<td>167.33</td>
<td>93.59</td>
<td>112.78</td>
<td>25.30</td>
</tr>
<tr>
<td>src.1GB</td>
<td>75.48</td>
<td>99.95</td>
<td>83.70</td>
<td>46.97</td>
<td>57.99</td>
<td>8.87</td>
</tr>
<tr>
<td>src.2GB</td>
<td>150.67</td>
<td>205.41</td>
<td>167.33</td>
<td>93.59</td>
<td>112.78</td>
<td>25.30</td>
</tr>
<tr>
<td>src.1GB</td>
<td>75.48</td>
<td>99.95</td>
<td>83.70</td>
<td>46.97</td>
<td>57.99</td>
<td>8.87</td>
</tr>
</tbody>
</table>

Table 4.2: Running times, in seconds, of the sequential and parallel algorithms with 1 and 64 threads. The best sequential times are underlined and the best parallel times are shown using bold typeface. A "-" is shown for implementations that just work for $n < 2^{32}$.

heavy disk access, while Libcds uses a recursive algorithm, with known memory and executions costs.

Figure 4.3 shows speedups for rna.3GB, prot, src.200MB, en.4.30, src.2GB, en.14.30 datasets, with the largest $n$. As expected, the put algorithm is competitive until $p < \lg \sigma$. Thus, for small $\sigma$ the put algorithm is not the best alternative as shown in Figures 4.3a, 4.3b and 4.3d. If the algorithm recruits more threads than levels, the overhead of handling these threads increases, generating some "noise" in the times obtained. The performance of put will be dominated also by the thread that builds more levels. For instance, in Figure 4.3f we created a wtree with 14 levels. In the case of one thread, that thread has to build the 14 levels. In the case of 4 threads, each has to build three levels. For 8 and 12 threads, some threads will build
Figure 4.3: Speedup with respect to the best sequential time. The caption of each figure indicates the name of the dataset, the input size $n$ and the alphabet size $\sigma$. 

(a) rna.3GB, $n \approx 2^{32}$, $\sigma = 4$.

(b) prot, $n \approx 2^{30}$, $\sigma = 27$.

(c) src.200MB, $n \approx 2^{28}$, $\sigma = 230$.

(d) en.4.30, $n = 2^{30}$, $\sigma = 2^{14}$.

(e) src.2GB, $n \approx 2^{29}$, $\sigma \approx 2^{21}$.

(f) en.14.30, $n = 2^{30}$, $\sigma = 2^{14}$. 

two levels, so those threads dominate the running time. Finally, for the case of 16 threads, each thread has to build at most one level. This explains the “staircase” effect seen for \texttt{pwt} in Figure 4.3f.

In all datasets shown in Figure 4.3, except for Figure 4.3e, the \texttt{dd} algorithm has a better speedup than both \texttt{pwt} and \texttt{shun}, especially for datasets with small alphabets, such as \texttt{rna}, \texttt{prot} and \texttt{en}. In the case of Figure 4.3e, \texttt{shun} has a better speedup, because our algorithms have worse data locality. We will discuss more about the impact of locality of reference at the end of this section. It is important to remember that although \texttt{shun} has a better speedup, its memory consumption is larger that in our algorithms, as can be seen in Figure 4.4.

**Memory consumption.** Figure 4.4 shows the amount of memory allocated with \texttt{malloc} and released with \texttt{free}. For all algorithms, we report the peak of memory allocation and only consider the memory allocated during construction, not the memory allocated to store the input text. The datasets are ordered incrementally by \( n \). In the case of the \texttt{dd} algorithm, the figure shows memory consumption for \( k = 1 \). \texttt{Libcdfs} and \texttt{shun} use more memory during construction time. In fact, \texttt{pwt} uses up to 33 and 25 times less memory than \texttt{Libcdfs} and \texttt{shun}, respectively. Memory usage in \texttt{libcdfs} is dominated by its recursive nature, while \texttt{shun} copies the input sequence \( S \), of \( O(n \lg n) \) bits, to preserve it and to maintain its permutations in each iteration. Additionally, \texttt{shun} uses an array of size \( O(\sigma \lg n) \) bits to maintain some values associated to the nodes of the \( \texttt{wtree} \), such as the number of bits, the range of the alphabet, and the offset. In our algorithms and in \texttt{sdsl}, memory consumption is dominated by the arrays which store offset values, not by the input sequence.

The main drawback of \texttt{dd} with respect to our own \texttt{pwt} is its memory consumption, since the latter increases with the alphabet size and the number of threads. For small alphabets, the working space of \texttt{dd} is almost constant. For instance, memory consumption for \texttt{rna.2GB} is 1GB, plus a small overhead for each new thread. For larger alphabets, such as \texttt{src.2GB} with \( \sigma \approx 2^{22} \), the working space increases linearly with the number of threads, using 1.46GB with 1 thread and 2.5GB with 12 threads. Fortunately, most of the sequences used in real-world applications have an alphabet size smaller than \( 2^{17} \). Such is the case of DNA sequences, the human genome, natural language alphabets (Unicode standard), etc.\(^6\).

**Other experiments.** In order to have a better understanding of our algorithms, we performed the following experiments:

**Limited resources.** When memory is limited, algorithms such as \texttt{Libcdfs} and \texttt{shun} suffer a decrement in their performance. This is evident in Figure 4.5, where we

\(^6\)The Unicode Consortium: [http://www.unicode.org/](http://www.unicode.org/)
Datasets
Peak(GB)
libcds
sdsl
prwt
pwt
dd
shun

Figure 4.4: Memory consumption sorted by $n$.

Figure 4.5: Running experiments in a machine with limited resources.

Figure 4.6: Speedup of the dataset $en.4.30$ encoding each symbol with 4 bytes.

Figure 4.7: Time over $n$ with $\sigma = 2^{14}$, 64 threads and $en.14$ datasets.

Figure 4.8: Time over $\sigma$ for the best and worst cases with $n = 2^{30}$ and $p = \lg \sigma$ threads.
tested the parallel algorithms with datasets `prot` and `src.1GB`\(^7\) on a 12-core computer with 6GB of DDR3 RAM (machine A). In this new set of experiments, the speedup of our algorithms exceeded the speedup shown by `shun`, both for datasets where we previously showed the better performance (see Figure 4.3b) and for datasets where previously `shun` showed better performance (see Figure 4.3e and Table 4.2).

**Encoding.** We observed that the encoding of the symbols of the original sequence has a great impact in the speedups of the construction algorithms. Figures 4.3a–4.3d have speedups greater than 27x, while there is a noticeable performance degradation in Figure 4.3e and Figure 4.3f. This is due to an encoding subtlety: The datasets used in the experiments resulting in Figures 4.3a–4.3d are encoded using one byte for each symbol, while the other used four bytes. To prove the impact of the encoding in the performance of the construction algorithms, we repeated the experiments using a dataset that used four bytes per symbol for \(\sigma \leq 2^8\). Figures 4.3d and 4.6 show the influence of encoding. As expected, the greater the memory used for encoding, the worse the performance. On multicore architectures, some levels of the memory hierarchy are shared by different cores. This increases the rate of memory evictions. Hence, it is crucial to reduce the number of memory transfers. Besides, in NUMA architectures, where each NUMA node has a local RAM and the transfers between local RAMs is expensive, the reduction of memory transfers is critical. In the case of one byte per symbol, each memory transfer carries four times more symbols than in the case of four bytes per symbol, effectively helping reduce memory transfers.

**Influence of the size of sequence.** Figure 4.7 shows that for the `en.14` dataset, fixing the number of threads to 64 and \(\sigma\) to \(2^{14}\), for larger \(n\) the domain decomposition algorithm behaves better in running time than the `pwt` algorithm and Shun’s algorithm. In other words, with more cores and enough work for each parallel task, the `dd` algorithm should scale appropriately.

**Influence of the locality of reference.** Theoretically, fixing \(n\) and varying \(\sigma\) with \(p = \lg \sigma\) threads, the `pwt` algorithm should show a constant-time behavior, no matter the value of \(\sigma\). However, in practice, the running times of `pwt` increase with the alphabet size. The reason for this difference in theoretical and practical results is that levels closer to the leaves in the `wtree` exhibit a weaker locality of reference. In other words, locality of reference of the `pwt` algorithm is inversely proportional to \(\sigma\). Additionally, the dynamic multithreading model assumes that the cost of access to any position in the memory is constant, but that assumption is not true in a NUMA architecture. In order to visualize the impact of the locality of reference over running times, we generate two artificial datasets with \(n = 2^{30}\), \(\Sigma = \{1 \ldots 2^y\}\), with \(y \in \{4, 6, 8, 10, 12, 14\}\) and encoding each symbol with four bytes. The first dataset,\(^7\) the construction times of `shun` with the `src.2GB` dataset exceeds one hour. To make the algorithms in the figures comparable, we report the running times for the dataset `src.1GB`.  

\(^7\)The construction times of `shun` with the `src.2GB` dataset exceeds one hour. To make the algorithms in the figures comparable, we report the running times for the dataset `src.1GB`. 

cont, was created by writing each symbol of $\Sigma n/\sigma$ times and then sorting the symbols according to their position in the alphabet. The second dataset, rand, was created in a similar fashion, but writing symbols at random positions. The objective of the cont dataset is to force the best case of the pwt algorithm, where the locality of reference is higher. In contrast, the rand dataset forces the average case, with a low locality of reference. In these experiments, we used the optimal number of threads of pwt, that is, $p = \lg \sigma$. Besides, we allocated evenly the memory over the NUMA nodes to ensure constant access cost to any position in the memory. The results are shown in Figure 4.8. In its average case, illustrated using dashed lines, the performance of the pwt algorithm is degraded for larger alphabets because locality of reference is low, increasing the amount of cache misses, and thus degrading the overall performance. In the best case, illustrated using solid lines, pwt shows a practical behavior similar to the theoretical one. Since the dd algorithm implements the pwt algorithm to build each partial wtree, the locality of reference also impacts its performance. However, because the construction of the partial utrees involves sequences of size $O(n/p)$, the impact is less than in the pwt algorithm. Finally, Shun’s algorithm is insensitive to the distribution of the symbols in the sequence.

The study of the impact of the architecture on the construction of wtrees and other succinct data structures, and the improvement of the locality of reference of our algorithms are interesting lines for future research.

**Discussion.** In most cases, the domain decomposition algorithm, dd, showed the best speedup. Additionally, dd can be adjusted either in favor of running time or

---

*To ensure the constant access cost, we use the *numactl* command with “interleave=all” option. The command allocates the memory using round robin on the NUMA nodes.*
memory consumption. \texttt{pwt} showed good scalability, but up to \( p < \lg \sigma \). This limitation may be overcome by using \texttt{pwt} as part of \texttt{dd}, dividing the input sequence in an adequate number of subsequences.

With respect to working space, \texttt{pwt} was the algorithm with the lowest memory consumption. This is important because an algorithm with low memory consumption can be executed in machines with limited resources, can reduce cache misses due to invalidations (false sharing) and can therefore reduce energy consumption. Even though memory consumption of the \texttt{dd} algorithm increases with the number of subsequences, it can be controlled by manipulating the number of segments. In the case of \texttt{shun}, its memory consumption is too large to be competitive in machines with limited memory.

The encoding and the distribution of the symbols of the input sequence impact the performance of the algorithms. All the parallel algorithms introduced here show a better speedup for encodings that use less bits because there are less memory transfers. Our algorithms are also sensitive to the distribution of the symbols. When the symbols are randomly distributed, the locality of reference is worse in comparison with more uniform distributions. This gives us a hint to improve the performance of our algorithms in the future.

To sum up, in general, the \texttt{dd} algorithm is the best alternative for the construction of \texttt{wtrees} on multicore architectures, considering both running time and memory consumption. For domains with limited resources, \texttt{pwt}, which is a building block of \texttt{dd}, arises as a good alternative on its own.

\subsection*{4.3.3 Querying Experiments}

To test our querying algorithms, we generated randomly 100,000 path queries and 10,000 branch queries. For branch queries, ranges over the text were selected with random bounds, the size was fixed at 1\%, and the range over the alphabet was fixed to 100\%. In order to stress the \texttt{parBQA} querying algorithm, we replaced the condition of line 16 of Function \texttt{batchRangeCount} by \texttt{if(nd.\sigma_l == nd.\sigma_r)}. This ensured that the query traversal reached the leaves of the \texttt{wtree}. All the queries were tested over the \texttt{wtree} created using the dataset \texttt{en.14.29} and the \texttt{pwt} algorithm.

Figure 4.9 shows the throughput of answering 100,000 path queries in parallel using domain decomposition. We made the experiments considering \texttt{select}, \texttt{rank}, and \texttt{access} queries. For the three type of queries, we can observe a linear increase in the throughput with respect to the number of threads. The lower throughput of \texttt{select} is due to the fact that it takes more time because it goes down and then goes up on the levels of the \texttt{wtree}. Meanwhile \texttt{rank} and \texttt{access} only need to go down the \texttt{wtree}.

As discussed in Section 4.2, the BQA technique implies a little more programming effort but improves throughput over IQA in both sequential and parallel settings. This is shown in the Figure 4.10. Figure 4.10a shows the sequential throughput of IQA and BQA. For BQA, we created batches of 2, 5, 10, 50, 100, 200, 300, 500, 1,000,
(a) Throughput comparison of IQA and BQA techniques.

(b) Read (−R) and write (−W) cache misses comparison of IQA and BQA techniques.

(c) Throughput comparison of dd-IQA and parBQA algorithms.

(d) Read (−R) and write (−W) cache misses of dd-IQA and parBQA algorithms.

Figure 4.10: Branch queries experiments over the dataset en.14.29. For BQA technique the batch size changes from 2 to 10,000.
Table 4.3: Throughput (Th), last level read misses (RM) and last level write misses (WM) of the dd-IQA and parBQA parallel algorithms. The experiments were run in the machine A with the dataset en.14.29.

<table>
<thead>
<tr>
<th>Algo</th>
<th>Batch size</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>dd-IQA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Th</td>
<td>137.57</td>
<td>290.30</td>
</tr>
<tr>
<td>RM</td>
<td>665860362</td>
<td>657818591</td>
</tr>
<tr>
<td>WM</td>
<td>48081819</td>
<td>48041149</td>
</tr>
<tr>
<td>2</td>
<td>100.56</td>
<td>194.62</td>
</tr>
<tr>
<td>RM</td>
<td>834877009</td>
<td>754355900</td>
</tr>
<tr>
<td>WM</td>
<td>152882639</td>
<td>154227895</td>
</tr>
<tr>
<td>5</td>
<td>146.82</td>
<td>301.31</td>
</tr>
<tr>
<td>RM</td>
<td>701741574</td>
<td>661159773</td>
</tr>
<tr>
<td>WM</td>
<td>114809684</td>
<td>117015656</td>
</tr>
<tr>
<td>10</td>
<td>221.74</td>
<td>397.68</td>
</tr>
<tr>
<td>RM</td>
<td>530267689</td>
<td>529014444</td>
</tr>
<tr>
<td>WM</td>
<td>101311270</td>
<td>103284485</td>
</tr>
<tr>
<td>50</td>
<td>343.40</td>
<td>586.75</td>
</tr>
<tr>
<td>RM</td>
<td>276960062</td>
<td>301410175</td>
</tr>
<tr>
<td>WM</td>
<td>91755956</td>
<td>95467415</td>
</tr>
<tr>
<td>100</td>
<td>335.09</td>
<td>608.06</td>
</tr>
<tr>
<td>RM</td>
<td>191175754</td>
<td>186654101</td>
</tr>
<tr>
<td>WM</td>
<td>91722746</td>
<td>93324149</td>
</tr>
<tr>
<td>200</td>
<td>348.47</td>
<td>659.30</td>
</tr>
<tr>
<td>RM</td>
<td>172426822</td>
<td>167873588</td>
</tr>
<tr>
<td>WM</td>
<td>91780910</td>
<td>92476066</td>
</tr>
<tr>
<td>500</td>
<td>363.48</td>
<td>641.26</td>
</tr>
<tr>
<td>RM</td>
<td>151292247</td>
<td>145254144</td>
</tr>
<tr>
<td>WM</td>
<td>91881274</td>
<td>9261973</td>
</tr>
<tr>
<td>1,000</td>
<td>392.69</td>
<td>751.09</td>
</tr>
<tr>
<td>RM</td>
<td>113568181</td>
<td>124760874</td>
</tr>
<tr>
<td>WM</td>
<td>90803770</td>
<td>81616490</td>
</tr>
<tr>
<td>10,000</td>
<td>406.16</td>
<td>809.51</td>
</tr>
<tr>
<td>RM</td>
<td>99570459</td>
<td>90650508</td>
</tr>
<tr>
<td>WM</td>
<td>85139187</td>
<td>69100728</td>
</tr>
</tbody>
</table>

and 10,000 queries. In all, except for batch size 2, the BQA technique has a better throughput than the IQA technique, until 3 times better. In general, the queries supported by a wtree incur in more read misses that in write misses, since most of the wtree has to be read while only few variables are written. Therefore, if we can reduce the read misses, we can improve the throughput of our query-answering algorithms. Since BQA exploits the spatial and temporal locality of the hierarchical memory, when the batch size increases, the amount of read misses and, in lesser extent, write misses decrease, increasing the throughput. For the case of batch size 2, the throughput of BQA is lower than IQA because its number of cache misses is greater than the cache misses of IQA. This is shown in Figure 4.10b.
Figure 4.10c shows the performance of \texttt{dd-IQA} and \texttt{parBQA}, with batch size of 2, 100, 500, 1,000 and 10,000 and varying the number of threads. Their corresponding read and write misses are shown in Figure 4.10d. For a complete report of throughputs and cache misses, see Table 4.3. The better throughput is reached by \texttt{parBQA} with a batch size of 10,000, 2.9 times better than \texttt{dd-IQA} for all the threads. For \texttt{dd-IQA}, the improvement with respect to itself was 10.7 times. For \texttt{parBQA}, it improvement was around 4 with batch sizes 2, 5, 10, 50, 100, 200, 300 and 500; 6.8 with batch size 1,000 and 10.7 with batch size 10,000. The amount of cache misses almost does not vary with the number of threads. For \texttt{parBQA}, with a batch size less than 10,000, the $O(\text{num.queries/batch.size})$ batches are processed in parallel, while inside each batch the $O(\sigma)$ independent paths are processes in parallel, too. For a batch size equal to the number of queries, there exist only one batch, processing in parallel only the $O(\sigma)$ tasks associated to the paths of the range queries. This implies that the \texttt{parBQA} algorithm has a good performance due more to the exploitation of the spatial and temporal locality and the parallel processing of the independent paths of range queries than the domain decomposition over all the queries.

### 4.4 Extensions

**Wavelet tree construction with very large inputs.** For all our construction algorithms we assume that the input sequence $S$ fits in memory. However, we can extend our results to the construction of \texttt{wtree}s where the input sequence $S$ and the \texttt{wtree} do not fit. Following some implementation ideas of SDSL\cite{51}, we can read the input sequence in buffers to construct partial \texttt{wtree}s for each buffer and finally merge all of them to obtain the final \texttt{wtree}. In more detail, we can extend our algorithms as follows:

1. Read the input sequence $S$ using a buffer of size $b$. We can use the portion of main memory that will not be used by the \texttt{wtree} as the buffer.

2. Create a partial \texttt{wtree} without rank/select structures taking the buffer as input. The partial \texttt{wtree} can be constructed in parallel using our \texttt{dd} algorithm with $O(b \lg \sigma / p)$ time and $O(1)$ span. (We could also use the \texttt{pwt} if the available memory is scarce). The starting position of each node in the partial \texttt{wtree} is stored in a bidimensional array $L$.

3. Repeat steps 1 and 2 until the complete input sequence is read.

4. After the complete input sequence is read, we compute the final position of the nodes of all the partial \texttt{wtree}s. These positions are computed by performing a parallel prefix sum\cite{65} over the values of the arrays $L$’s, similar to the \texttt{dd} algorithm. It takes $O(b \sigma / p + \lg p)$ time and $O(\lg (b \sigma))$ span.

5. The final \texttt{wtree} is constructed using Function \texttt{mergeBA} with $O(n \lg \sigma / pw)$ and $O(1)$ span, where $w$ is the word size of the architecture.
The extension takes $O(n \log \sigma/p + b \sigma/p + \log p)$ time and $O(n/b + \log(b\sigma))$ span. Notice that this idea is similar to the dd algorithm and it can be applied on multiple levels. For example, it can be used on distributed architectures, where the buffers are processed by different machines, and one machine merges all the partial wtrees. Additionally, observe that we can use the entire main memory as the buffer, storing the partial wtrees and the $L$ arrays on disk each time we finish the processing of a buffer. We leave the implementation and empirical evaluation of these ideas as future work.

**Huffman shaped wavelet trees.** In [42], Foschini et al. introduced an improvement for the wtree in order to reduce the costs of the queries and improve the compression of the sequence. The improvement changes the shaped of the original wtree by adopting a Huffman prefix tree shape. Thus, instead of using the original encoding of the symbols of $\Sigma$ to construct the bitmaps of the wtree, we use the Huffman codes of the symbols. We can use both the pwt and the dd algorithms to construct the Huffman shaped wtree. First of all, we need to compute the new encoding of the symbols in parallel. Edwards and Vishkin [34] introduced a parallel algorithm to compute the Huffman codes of a sequence of size $n$ and alphabet size $\sigma$ with $O(n+\sigma)$ work and $O(\log n + \sigma)$ span. Assume that the resulting Huffman codes and the length of each code are (temporarily) stored in a table $H$ of size $O(\sigma \log n)$. With the table $H$, we can use the pwt algorithm without changes. For all the levels at the same time, we count the number of bits on each node of the wtree, using the input sequence $S$, the table $H$ and bit shifting operations. With the length of the Huffman code of each symbol, we can detect when a symbol does not need to be represented in a particular level. After that, we traverse the input sequence $S$ again, writing the bits in their corresponding positions. Since the dd algorithm is based on the pwt algorithm, the previous explanation is valid for it. Finally, a Huffman shaped wtree of height $h$ can be constructed in parallel with $O(nh)$ work and $O(n+\sigma)$ span using the pwt algorithm, and with $O(nh)$ work and $O(n \log n + \sigma)$ span using the dd algorithm, by dividing $S$ into $k = O(p/h)$ segments.
Chapter 5

Parallel Construction of Succinct Trees

In this chapter, we describe our parallel algorithm for constructing the RMMT of a given tree, called the Parallel Succinct Tree Algorithm (PSTA). Its input is the balanced parenthesis sequence $P$ of an $n$-node tree $T$. This is a tree representation commonly used in practice, particularly in secondary storage. For trees whose folklore encoding is not directly available, in Section 5.1 we describe a parallel algorithm that can compute such an encoding in $O(n/p + \lg p)$ time. Our algorithms assume that manipulating $w$ bits takes constant time. Additionally, we assume the (time and space) overhead of scheduling threads on cores is negligible. This is guaranteed by the results of [12], and the number of available processing units in current systems is generally much smaller than the input size $n$, so this cost is indeed negligible in practice.

5.1 Parallel Folklore Encoding Algorithm

The PSTA algorithm requires the balanced parentheses representation $P$ of the input tree $T$, but in some applications $T$ may not be given in this form. Here, we present a parallel algorithm that constructs the balanced parenthesis sequence of $T$ from a representation of $T$ stored in adjacency list representation. Since the balanced parenthesis sequence of $T$ is also known as its folklore encoding, we call the algorithm the Parallel Folklore Encoding Algorithm (PFEA). The input tree is represented by an array of nodes, $V_T$, and an array of edges, $E_T$. Each node $v \in V_T$ stores two indices in $E_T$, $v\text{.first}$ and $v\text{.last}$, indicating the adjacency list of $v$, sorted counterclockwise around $v$ and starting with $v$’s parent edge. Notice that the number of children of $v$ is $(v\text{.last} - v\text{.first})$. Each edge $e \in E_T$ has three fields, $e\text{.src}$ which is a pointer to the source vertex, $e\text{.tgt}$ which is a pointer to the target vertex and $e\text{.cmp}$ which is the position in $E_T$ of the complement edge of $e$, $e'$, where the $e'\text{.src} = e\text{.tgt}$ and $e'\text{.tgt} = e\text{.src}$. For $x \in \{e\text{.src}, e\text{.tgt}\}$, we use $\text{next}(e.x)$ and $\text{first}(e.x)$ to denote the indices in $E_T$ of $e$’s successor and of the first element (parent edge) in $x$’s adjacency list, respectively. Both are easily computed in constant time by following pointers. Figure 5.1 shows an example of the used representation.

The idea behind the construction is the following: Given an Euler tour of $T$ that visits the children of each node in left-to-right order, the balanced parenthesis representation of $T$ can be obtained by following the Euler tour, writing down an open parenthesis for every edge traversed from parent to child (forward edge) and a closed parenthesis for every edge traversed from child to parent (backward edge), and finally enclosing the resulting sequence in a pair of parentheses representing the root of $T$. 
Algorithm 10 shows the pseudo-code of the construction. It creates two arrays, one an auxiliary array $ET$ of length $|E_T|$ to store the Euler tour of $T$, the other an array $P$ of size $|E_T| + 2$ to store the balanced parenthesis representation of $T$ (lines 1–2). Each entry in $ET$ represents the traversal of an edge of $T$ and stores three values: value is “(“ or “)“ depending on whether the edge is traversed from parent to child or from child to parent, that is, it is the corresponding parenthesis to be added to $P$; succ is the index in $ET$ of the next edge in the Euler tour; and rank is the rank in the Euler tour. Lines 4–16 of the algorithm populate $ET$ with entries representing the Euler tour, with the rank values initialized with 1. Line 17 computes the final ranks using a parallel list ranking algorithm [65]. Given these ranks, the balanced parenthesis representation can be obtained by writing $ET[i].value$ into $P[ET[i].rank]$. Lines 18–22 do exactly this.

5.2 Parallel Succinct Tree Algorithm

Before describing the PSTA algorithm, we observe that the entries in $e'$ corresponding to internal nodes of the RMMT need not be stored explicitly. This is because the entry of $e'$ corresponding to an internal node is equal to the entry that corresponds to the last leaf descendant of this node; since the RMMT is complete, we can easily locate this leaf in constant time. Thus, our algorithm treats $e'$ as an array of length $\lceil 2n/s \rceil$ with one entry per leaf. Our algorithm consists of three phases. In the first phase (Algorithm 11), it computes the leaves of the RMMT, i.e., the array $e'$, as well as the entries of $m'$, $M'$ and $n'$ that correspond to leaves. In the second phase (Algorithm 12), the algorithm computes the entries of $m'$, $M'$ and $n'$ corresponding to internal nodes of the RMMT. In the third phase (Algorithm 13), it computes the universal lookup tables used to answer queries. The input to our algorithm consists of the balanced parenthesis sequence, $P$, the size of each chunk, $s$, and the number of available threads, threads.

To compute the entries of arrays $e'$, $m'$, $M'$, and $n'$ corresponding to the leaves of the RMMT (Algorithm 11), we first assign the same number of consecutive chunks, $ct$, to each thread (line 4). We call such a concatenation of chunks assigned to a single thread a superchunk. For simplicity, we assume that the total number of chunks,
**Input**: An adjacency list representation of \( T \) consisting of arrays \( V_T \) and \( E_T \) and the number of threads, \( \text{threads} \).

**Output**: The balanced parenthesis sequence \( P \) of \( T \).

1. \( E_T = \) an array of length \(|E_T|\)
2. \( P = \) an array of length \(|E_T| + 2\) // equivalently to an array of length \( 2|V_T|\)
3. \( \text{chk} = |E_T|/\text{threads} \)
4. \text{parfor} \( t = 0 \) to \( \text{threads} - 1 \) do
5.  \( \text{for} \ i = 0 \) to \( \text{chk} - 1 \) do
6.  \( j = t \times \text{chk} + i \)
7.  \( e = E_T[j] \)
8.  \( E_T[j].rank = 1 \)
9.  \( \text{if} \ \text{isRoot}(e.\text{src}) \ OR \ \text{first}(e.\text{src}) \neq j \) then // Forward edge
10.  \( E_T[j].value = 1 \) // open parenthesis
11.  \( \text{if} \ e.\text{tgt} \text{ is a leaf} \) then
12.  \( E_T[j].\text{succ} = e.\text{cmp} \)
13.  \( \text{else} \)
14.  \( E_T[j].\text{succ} = \text{first}(e.\text{tgt}) + 1 \)
15.  \( \text{else} \)
16.  \( E_T[j].value = 0 \) // closed parenthesis
17.  \( \text{if} \ E_T[j] \text{ is the last edge in the adjacency list of } e.\text{src} \) then
18.  \( E_T[j].\text{succ} = \text{first}(e.\text{tgt}) \)
19.  \( \text{else} \)
20.  \( E_T[j].\text{succ} = \text{next}(e.\text{tgt}) \)
21. \( \text{ParListRanking}(E_T) \)
22. \text{parfor} \( t = 0 \) to \( \text{threads} - 1 \) do
23.  \( \text{for} \ i = 0 \) to \( 2 \times \text{chk} - 1 \) do
24.  \( P[ET[2 \times t \times \text{chk} + i + 1].\text{rank}] = ET[2 \times t \times \text{chk} + i + 1].\text{value} \)
25. \( P[0] = 1 \)
26. \( P[|E_T| + 1] = 0 \)

**Algorithm 10**: Parallel Folklore Encoding Algorithm (PFEA)

\( \lceil \frac{2n}{s} \rceil \), is divisible by \( \text{threads} \). Each thread then computes the local excess value of the last position in each of its assigned chunks, as well as the minimum and maximum local excess in each chunk, and the number of times the minimum local excess occurs in each chunk (lines 8–17). These values are stored in the entries of \( e', m', M' \), and \( n' \) corresponding to this chunk (lines 18–21). The local excess value of a position \( i \) in \( P \) is defined to be \( \text{sum}(P, \pi, j, i) \), where \( j \) is the index of the first position of the superchunk containing position \( i \). Note that the locations with minimum local excess in each chunk are the same as the positions with minimum global excess because the difference between local and global excess is exactly \( \text{sum}(P, \pi, 0, j - 1) \). Thus, the entries in \( n' \) corresponding to leaves store their final values at the end of the loop in lines 5–21, while the corresponding entries of \( e', m', \) and \( M' \) store local excess values.
Input: $P$, $s$, threads

Output: RMT represented as arrays $e', m', M', n'$ and universal lookup tables

1 $o = \lceil 2n/s \rceil - 1$ // # internal nodes
2 $e' = $ array of size $\lceil 2n/s \rceil$
3 $m', M', n' = $ arrays of size $\lceil 2n/s \rceil + o$
4 $ct = \lceil 2n/s \rceil$/threads
5 parfor $i = 0$ to $threads - 1$ do
6 $n' = 0$
7 for $chk = 0$ to $ct - 1$ do
8 $low = (t * ct + chk) * s$
9 $up = low + s$
10 for $par = low$ to $up - 1$ do
11 $e'_t += 2 * P[par] - 1$
12 if $e'_t < m'_t$ then
13 $m'_t = e'_t; n'_t = 1$
14 else if $e'_t = m'_t$ then
15 $n'_t += 1$
16 else if $e'_t > M'_t$ then
17 $M'_t = e'_t$
18 $e'[t * ct + chk] = e'_t$
19 $n'[t * ct + chk + o] = m'_t$
20 $M'[t * ct + chk + o] = M'_t$
21 $n'[t * ct + chk + o] = n'_t$
22 parPrefixSum($e', ct$)
23 parfor $i = 1$ to $threads - 1$ do
24 $t = 0$ to $ct - 1$ do
25 if $chk < ct - 1$ then
26 $e'[t * ct + chk] += e'[t * ct - 1]$
27 $m'[t * ct + chk + o] += e'[t * ct - 1]$
28 $M'[t * ct + chk + o] += e'[t * ct - 1]$

Algorithm 11: Parallel Succinct Tree Algorithm (PSTA), part I

$lvl = \lceil \lg threads \rceil$

parfor $st = 0$ to $2^{lvl} - 1$ do
for $l = \lceil \lg(2n/s) \rceil - 1$ downto $lvl$ do
for $d = 0$ to $2^{l-lvl} - 1$ do
for $l = lvl - 1$ to $0$ do
parfor $d = 0$ to $2^l - 1$ do
concat($i, m', M', n'$)

Algorithm 12: Parallel Succinct Tree Algorithm (PSTA), part II

parfor $x = -w$ to $w - 1$ do
parfor $y = 0$ to $\sqrt{2w} - 1$ do
$i = (x + w) < w$ OR $w$;
$near_{fwd\_pos}[i] = w$;
$p, excess = 0$;
repeat
$excess += 1 - 2 *$
((y AND(1 < p)) = 0);
if $excess = x$ then
$near_{fwd\_pos}[i] = p$;
break;
$p += 1$;
until $p \geq w$;

Algorithm 13: Parallel Succinct Tree Algorithm (PSTA), part III

Input: $i$, $m'$, $M'$, $n'$

$m'[i] = \min(m'[2i + 1], m'[2i + 2]);$
$M'[i] = \max(M'[2i + 1], M'[2i + 2]);$
if $m'[2i + 1] < m'[2i + 2]$ then
$n'[i] = n'[2i + 1];$
else if $m'[2i + 1] > m'[2i + 2]$ then
$n'[i] = n'[2i + 2];$
else if $m'[2i + 1] = m'[2i + 2]$ then
$n'[i] = n'[2i + 1] + n'[2i + 2];$

Function concat
To convert the entries in $e'$ into global excess values, observe that the global excess at the end of each superchunk equals the sum of the local excess values at the ends of all superchunks up to and including this superchunk. Thus, we use a parallel prefix sum algorithm [65] in line 22 to compute the global excess values at the ends of all superchunks and store these values in the corresponding entries of $e'$. The remaining local excess values in $e'$, $m'$, and $M'$ can now be converted into global excess values by increasing each by the global excess at the end of the preceding superchunk. Lines 23–28 do exactly this.

The computation of entries of $m'$, $M'$, and $n'$ (Algorithm 12) first chooses the level closest to the root that contains at least threads nodes and creates one thread for each such node $v$. The thread associated with node $v$ calculates the $m'$, $M'$, and $n'$ values of all nodes in the subtree rooted at $v$, by applying the function $\text{concat}$ to the nodes in the subtree bottom up (lines 2–6). The invocation of this function for a node computes its $m'$, $M'$, and $n'$ values from the corresponding values of its children. With a scheduler that balances the work, such as a work-stealing scheduler, cores have a similar workload. Lines 7–10 apply a similar bottom-up strategy for computing the $m'$, $M'$, and $n'$ values of the nodes in the top lvl levels, but they do this by processing these levels sequentially and, for each level, processing the nodes on this level in parallel.

Algorithm 13 illustrates the construction of universal lookup tables using the construction of the table $\text{near_fwd_pos}$ as an example. This table is used to support the $\text{fwd_search}$ operation (see Section 3.2.2). Other lookup tables can be constructed analogously. As each entry in such a universal table can be computed independently, we can easily compute them in parallel.

### 5.2.1 Theoretical analysis.

In the PFEA algorithm, lines 4–16 and 18–22 perform $O(n)$ work and have $T_p = O(n/p)$ and span $T_\infty = O(1)$. The whole computation here (and in Lines 18–22) could have been formulated as a single parallel loop. However, in the interest of limiting scheduling overhead, we create only as many parallel threads as necessary, similar to the PSTA algorithm in Section 5.2. Line 17 performs $O(n)$ work and has $T_p = O(n/p + \lg p)$ and span $O(\lg n)$. This gives a total work of $T_1 = O(n)$ and a span of $T_\infty = O(\lg n)$. The running time on $p$ cores is $T_p = O(n/p + \lg p)$.

The analysis of the PSTA algorithm is done in three steps: Lines 1–21 of Algorithm 11 require $O(n)$ work and have span $O(1)$. Line 22 requires $O(p)$ work and has span $O(\lg n)$ because we compute a prefix sum over only $p$ values. Lines 23–28 require $O(n)$ work and have span $O(1)$. Lines 1–6 of Algorithm 12 require $O(n/s)$ work and have span $O(1)$. Lines 7–10 require $O(p)$ work and have span $O(\lg n/s)$. Algorithm 13 requires $O(\sqrt{2^w\text{poly}(w)})$ work and has span $O(1)$, where $w$ is the machine word size. Thus, the total work of PSTA is $T_1 = O(n + \lg p + \sqrt{2^w\text{poly}(w)})$ and its span is $O(\lg n)$. This gives a running time of $T_p = O(T_1/p + T_\infty) = O(n/p + \lg p + \sqrt{2^w\text{poly}(w)}/p)$. 
on $p$ cores\footnote{Notice that the term $\lg n$ of the span is implicit in the term $n/p + \lg p$ of $T_p$. When $p \leq n/\lg n \rightarrow n/p \geq \lg n$. When $p > n/\lg n \rightarrow \lg p = \Theta(\lg n)$.}. The speedup is $T_1/T_p = O\left(\frac{p(n+\sqrt{T_p\text{poly}(w)})}{n+\sqrt{T_p\text{poly}(w)}+p\lg p}\right)$. Under the assumption that $p \ll n$, the speedup approaches $O(p)$. Moreover, the parallelism $T_1/T_\infty$ (the maximum theoretical speedup) of PSTA is \( \frac{2+\sqrt{T_p\text{poly}(w)}}{\lg n} \).

The PSTA algorithm does not need any extra memory related to the use of threads. Indeed, it just needs space proportional to the input size and the space needed to schedule the threads. A work-stealing scheduler, like the one used by the DyM model, exhibits at most a linear expansion space, that is, $O(S_1p)$, where $S_1$ is the minimum amount of space used by the scheduler for any execution of a multithreaded computation using one core. This upper bound is optimal within a constant factor \cite{12}. In summary, the working space needed by our algorithm is $O(n \lg n + S_1 p)$ bits. Since in our algorithm the scheduler does not need to consider the input size to schedule threads, $S_1 = O(1)$. Thus, since in modern machines it is usual that $p \ll n$, the scheduling space is negligible and the working space is dominated by $O(n \lg n)$.

Note that in succinct data structure design, it is common to adopt the assumption that $w = \Theta(\lg n)$, and when constructing lookup tables, consider all possible bit vectors of length $(\lg n)/2$ (instead of $w/2$). This guarantees that the universal lookup tables occupy only $o(n)$ bits. Adopting the same strategy, we can simplify our analysis and obtain $T_p = O(n/p + \lg p)$. Thus, we have the following theorem:

**Theorem 1.** A $(2n + O(n/\lg n))$-bit representation of an ordinal tree on $n$ nodes and its balanced parenthesis sequence can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ bits of working space. This representation can support the operations in Table 3.1 in $O(\lg n)$ time.

### 5.3 Parallel Algorithm to Support Constant-Time Queries

In this section we show how to construct the 2d-Min-Heap and its ladders, the sparse bitmap of Pătraşcu, fusion trees and range-minimum-query structure in parallel, plus the computation of marked blocks. All of these structures are built over the minima, maxima, excess and the number of minima values of the $\tau = \lceil 2n/w^c \rceil$ RMMTs and are used to support different operations over trees in constant time (see Section 3.2.2).

**2d-min-heap and ladders:** Let $S = (x_0, x_1, \ldots, x_n = -\infty)$ be a sequence on $n$ integers. Let the closest smaller successor of $x_i$ be the element $x_j$ such that $j = \min\{j'|j' > i \land x_{j'} < x_i\}$. Thus, $x_j$ is the parent of $x_i$ in the 2d-Min-Heap. The 2d-Min-Heap is then fully determined once we find the closest smaller successor of all elements $x_i \in S$.

Let $C$ be the cartesian tree of $S$. Let the closest right ancestor of $x_i$ in $C$ be the closest ancestor $x_j$ of $x_i$ such that $x_i$ is in the left subtree of $x_j$. Since $x_n = -\infty$, both the closest smaller successor and the closest right ancestor are well-defined for all $x_i$,
### Cartesian tree:

(a) Cartesian tree of the sequence \((1, 4, 9, 5, 10, 7, 3, 2, 5, -\infty)\). Dummy nodes are represented with squares.

(b) Euler Tour \(ET\) of the cartesian tree in Figure 5.2a and the sequence \(\delta(ET)\). The first occurrence of each node of the cartesian tree is highlighted with a square. The arrows show the closest right ancestor of each node.

**Figure 5.2:** Example of the proof of Lemma 1. The resulting 2d-Min-Heap corresponds to the tree in Figure 3.4a.

where \(0 \leq i \leq n - 1\). Observe that the closest smaller successor of \(x_i\) and the closest right ancestor are the same element.

Let \(ET = (y_0, y_1, \ldots, y_m)\) be the Euler tour of \(C\) that visits the children of each node in right-to-left order. To ensure that each node in \(C\) has two children, we add (virtual) dummy nodes. We assume that every node \(x_i\) in \(C\), and hence in \(ET\), is labelled with its index \(i\) in \(S\). We also assume that for some \(x_i\) in \(C\), we know the first occurrence of \(x_i\) in \(ET\). Both these assumption can be obtain as part of the construction of \(ET\). We can obtain the closest right ancestor by performing a list ranking of \(ET\), computing the sequence \(\delta(ET) = (z_0, z_1, \ldots, z_m)\) defined as \(z_0 = y_0\) and \(z_i = \delta(z_{i-1}, y_i)\), for all \(1 \leq i \leq n\), where \(\delta(\cdot, \cdot)\) is defined as

\[
\delta(x, y_i) = \begin{cases} 
    y_i, & \text{if } s(i) < i, \text{ where } s(i) \text{ denotes the index of } y_i \text{'s successor in } ET \\
    x, & \text{otherwise}
\end{cases}
\]

See Figure 5.2b as an example of the Euler Tour \(ET\) of the tree in Figure 5.2a and its corresponding sequence \(\delta(ET)\).
Lemma 1. If $y_i$ is the first occurrence of some element $x_j$ in $ET$, then the element $z_{i-1}$ in $\delta(ET)$ is $x_j$'s closest right ancestor in $C$.

Proof. Since $ET$ visits all the descendants of a node $x_k$ after the first occurrence of $x_k$ in $ET$, the first occurrence of $x_j$ in $ET$ comes after the first occurrence of $x_k$ if $x_k$ is $x_j$'s closest right ancestor. Now assume that $y_h$ is the last occurrence of $x_k$ before the first occurrence $y_i$ of $x_j$. Let $(y_h, y_{h+1}, \ldots, y_t)$ be the subsequence of $ET$ between $y_h$ and $y_t$, and let $(k = j_h, j_{h+1}, \ldots, j_i = j)$ be the sequence of indices such that $y_t = x_{j_i}$, for all $h \leq t \leq i$, where $x_{j_i}$ is the $j_i$-th element in $ET$.

To prove the lemma, we need to show that $j_{h+1} < j_h$ and $j_{i+1} > j_i$ for all $h < t < i$ because this implies that $\delta(x, y_h) = y_h = x_k$ and $\delta(x, y_t) = x$ for all $h < t < i$, that is, $z_h = \delta(z_{h-1}, y_h) = x_k$ and $\delta(z_{i-1}, y_t) = z_{i-1} = z_h = x_k$ for all $h < t < i$; in particular, $z_{i-1} = x_k$, as claimed. The node $y_{h+1}$ must be $x_k$'s left child in $C$ because the last visit to $x_k$ by $ET$ before visiting any node in $x_k$'s left subtree happens immediately before visiting $x_k$'s left child. Thus, $j_{h+1} < j_h$. All the nodes on the path from $y_{h+1}$ to $x_j$ in $C$ are left ancestors of $x_j$ because $x_k$ is the closest right ancestor of $x_j$. Thus, by the definition of $ET,(y_{h+1}, y_{h+2}, \ldots, y_t)$ is the sequence of nodes in this path. Since $y_{i+1}$ is the right child of $y_i$ for all $h < t < i$, we have that $j_{i+1} > j_i$.

See Figure 5.2 as an illustration of this proof.

We can parallelize this strategy using the results of [117] to obtain the cartesian tree $C$ of $S$ with $O(n)$ work, $O(\lg^2 n)$ span and $O(n)$ working space, our PFEA algorithm in Section 5.1 to compute the Euler tour $ET$ with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space, and the results of [65] to compute the list ranking using the function $\delta(\cdot, \cdot)$ with $O(n)$ work, $O(\lg n)$ span and $O(n)$ working space.

Next we compute $\delta(ET)$ using list ranking (Lemma 1). Then, we assign one core to every element $z_i \in \delta(ET)$. The core writes $z_i$ as the closest right ancestor of $x_j$ if and only if $y_i$ is the first occurrence of $x_j$ in $ET$. This is done in $O(1)$ time.

Thus, the complexity of constructing the 2d-Min-Heap in parallel is $O(n)$ work, $O(\lg^2 n)$ span and $O(n \lg n)$ working space.

After constructing the 2d-Min-Heap, the tree is decomposed into ladders. The ladders are constructed by recursively extracting the longest path of the tree. This gives us a set of paths. Then, each path of length $l$ is extended by adding at most $l$ nodes towards the root. Those extended paths are called ladders. To construct the ladders in parallel, assume that we have a tree $T^B$ with a particular embedding $B$: for each node $v$ of $T^B$, the children of $v$ are ordered by their height, with the highest child in the leftmost position.

Lemma 2. Given a tree $T^B$ with embedding $B$ and $n$ nodes, the ladders of $T^B$ can be constructed in parallel with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space.

Proof. To prove the lemma, we need to compute the depth of each node of the tree. This can be done in parallel by using the PFEA algorithm of Section Section 5.1,
Figure 5.3: Computation of the ladders of a tree $T^B$, with embedding $B$. The tree $T^B$ is the result of applying the embedding $B$ to the tree of Figure 3.4. In the tree, the depth of each node is shown. For example, $d:3$ means that the depth of a node is 3. In the Euler Tour $ET_n$, the dummy root is represented by the symbol $r$.

Adding 1 for each forward edge and subtracting 1 for each backward edge. It takes $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. Now, let $ET_n = (v_0, \ldots, v_m)$ be the Euler tour of $T^B$ that visits the children of each node in left-to-right order and writes the index of each node. Let $ET_d = (v'_0, \ldots, v'_m)$ be the Euler tour of $T^B$ that visits the children of each node in left-to-right order and writes the depth of each node. We can decompose the tree into paths by finding contiguous increasing subsequences in $ET_d$. By the definition of the embedding $B$, the resulting paths are the same ones obtained by recursively extracting the longest path of the tree. For the path represented by the subsequence $ET_d[a..b]$, $ET_d[b]$ corresponds to the depth of the leaf of this path and the length of $ET_d[a..b]$ is $b - a + 1$.

To compute the ladders, we need to extend each subsequence in $ET_d$. For a subsequence $ET_d[a..b]$, if $ET_n[a]$ is the root of the tree, then the subsequence does not need to be extended. Otherwise, the subsequence need to be extended by adding up to $x = (b - a - 1)$ extra nodes. If $x = 0$, then the subsequence does not need to be extended. The extra nodes that we need to add correspond to ancestors of the leaf $ET_n[b]$ at depths $ET_d[a] - i$, $i \in (1, \ldots, x)$. We use the operation $\text{level} \_ \text{anc}(ET_n[a], i)$, $i \in (1, \ldots, x)$, of the simplified NS-representation (see Table 3.1, operation 5) to obtain all the ancestors and the ladders.

The Euler Tours $ET_n$ and $ET_d$ can be found using the PF EA algorithm. The bounds of all increasing subsequences can be found in parallel by finding each index $i$, such that $ET_d[i] < ET_d[i - 1]$ or $ET_d[i] > ET_d[i + 1]$. This can be done with $O(n)$ work, $O(1)$ span and $O(n \lg n)$ working space. The simplified NS-representation can be constructed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. The $\text{level} \_ \text{anc}$ operation of the simplified NS-representation takes $O(\lg n)$ time to be answered. Since the total length of all the ladders is $2n$ [102], the amount of operations that we need to perform is $O(n)$. We can perform all the operations independently, so the $O(n)$ $\text{level} \_ \text{anc}$ operations can be answered in parallel with $O(n \lg n)$ work and $O(\lg n)$ span.

Figure 5.3 shows an example of the Lemma 2.
Alternatively, observe that we can extend a subsequence $ET_d[a..b]$ using wavelet trees. Since the extra nodes that we need to add correspond to the ancestors of the leaf $ET_n[b]$, they appear before $ET_n[b]$ in the Euler Tours, with depths $ET_d[a] - i, i \in (1, \ldots, x)$. Given a node $v$ at position $j$ in $ET_d$, we know that the parent of $v$ is at position $k$ in $ET_d$, where $k = \max\{k'|k' < j, ET_d[k'] = ET_d[j] - 1\}$. In general, to extend the subsequence $ET_d[a..b]$, we need the nodes at positions $\max\{k'|k' < a, ET_d[k'] = ET_d[a] - i\}$, with $i \in (1, \ldots, x)$. Those positions can be found by using rank/select operations over $ET_d$. To answer the rank/select operations efficiently, we could construct a wavelet tree over $ET_d$, considering the contiguous alphabet $\Sigma = \{0, \lfloor \lg |ET_d|\rceil - 1\}$. For example, to extend the subsequence $ET_d[a..b]$ with a node with depth $d'$, we need to perform $\text{select}_{d'}(ET_d, \text{rank}_{d'}(ET_d, a))$. Finally, once we find the position of all the nodes, we use $ET_n$ to obtain their indexes. To construct the wavelet trees in parallel, we can use the $\text{pw}$ algorithm, with $O(n \lg n)$ work, $O(n)$ span and $O(n \lg n)$ working space, or the $\text{dd}$ algorithm, with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n^2 \lg n)$ working space. Observe that if there are $p < \lg n$ available threads, the working space of the $\text{dd}$ algorithm is reduced to $O(n \lg^2 n)$. The rank/select operations over the wavelet tree take $O(\lg n)$. We can perform all the operations independently, so the $O(n)$ rank/select operations can be answered in parallel in $O(\lg n)$ time.

The following lemma presents an extension of the Lemma 2 for trees with arbitrary embeddings.

**Lemma 3.** Given a tree $T^A$ with an arbitrary embedding $A$ and $n$ nodes, the ladders of $T^A$ can be constructed in parallel with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space.

**Proof.** To prove this lemma, we need to map the embedding $A$ of $T^A$ to $B$. To compute the embedding $B$, we need to order all the children of the nodes of $T^A$ by height with the highest in the leftmost position. To do this, we use the $\text{PFEA}$ algorithm to compute the folklore encoding of $T$ and then construct its simplified NS-representation to use the $\text{height}$ operation to obtain the height of all the nodes. Both the folklore encoding and the simplified NS-representation can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. The $\text{height}$ operation takes $O(\lg n)$ and there are $n$ operations, and therefore all the operations can be done with $O(n \lg n)$ work and $O(\lg n)$ span. After that, we can use a parallel stable sorting algorithm over the children of the nodes of $T^A$. Raman [108] sorts an array of $n$ integers each in the domain $[1, \ldots, m]$, for $m = n^{O(1)}$, with $O(n \lg \lg m)$ work, $O(\lg n / \lg \lg n + \lg \lg m)$ span and $O(n \lg m)$ working space. In our case, the total number of children in $T$ is $n - 1$ or $2(n - 1)$ by using bidirectional edges, and the height of any node is less than $n$. Therefore, we can sort the children of all the nodes of $T$ with $O(n \lg \lg n)$ work, $O(\lg n / \lg \lg n)$ span and $O(n \lg n)$ space.

With the new embedding $B$, we use Lemma 2 to finish the proof.
In the NS-representation, the 2d-Min-Heap has \( \tau \) nodes, and therefore, the 2d-Min-Heap and its ladders can be computed with \( O(\tau \log \tau) \) work, \( O(\log^2 \tau) \) span and \( O(\tau \log \tau) \) working space.

**Pătraşcu’s bitmap:** Navarro and Sadakane use the sparse bitmap of Pătraşcu [106] to represent a bitmap with \( 2\tau \) 1’s and \( 2\tau \) 0’s using \( O(\tau \log \tau + \tau \frac{w^c}{\log (\tau w^c)} + (\tau w^c)^{3/4}) \) bits and supporting rank/select queries in \( O(t) \) time. Pătraşcu demonstrated how to use recursion to achieve a better redundancy. Given a sparse bitmap \( A \) of size \( n \), the succinct representation of \( A \) is constructed as follows: Choose \( B \geq 2 \) such that \( B \log B = \varepsilon \log n \), and \( r = B^t = (\frac{\log n}{t})^{\Theta(t)} \). We first divide the bitmap \( A \) into \( n/r \) segments of size \( r \). Each segment is stored in a succinct aB-tree. Each succinct aB-tree is constructed by dividing the bitmap into \( B \) independent segments. On each small segment, the author applies Lemma 3 of [106] recursively \( t \) times. In order to reduce the redundancy, on each application of the lemma, \( M \) memory bits are extracted from the values of the independent segments and stored, and the rest of the unextracted bits, called spill, are passed to the next iteration. Then, Lemma 5 of [106] is applied in each succinct aB-tree, storing the last spill and memory bits in the root of each aB-tree. For each segment of size \( r \), the index in memory of the segment’s memory bits are stored. Additionally, the number of ones of each segment are stored in a partial sums vector and a predecessor structure to support rank and select operations, respectively.

The parallel algorithm to construct the Pătraşcu’s bitmap is similar to the parallel algorithm we used to construct the RMT in Section 5.2. First, we construct the \( n/r \) succinct aB-trees in parallel. On each aB-tree, we divide the bitmap on \( B \) independent bitmaps of size \( r/B \), similar to the PSTA algorithm. We apply Lemma 3 of [106] recursively on each small bitmap, \( t \) times. Then, we apply Lemma 5 of [106] in each succinct aB-tree, storing the final spill and memory bits in the root of each aB-tree. After that, all the \( n/r \) succinct aB-trees are built with \( O(n) \) work and \( O(t) \) span. The next step consists of storing the values of the roots of each aB-tree. To support the rank operation, we compute in parallel the partial sum vector of these values with \( O(n/r) \) work and \( O(\log(n/r)) \) span using a parallel prefix sum algorithm. To support select operation, we can use a fusion tree. Below, we will explain how to construct a fusion tree in parallel with \( O(n/r) \) work and \( O(\log(n/r)) \) span using a parallel prefix sum algorithm. To support select operation, we can use a fusion tree. Below, we will explain how to construct a fusion tree in parallel with \( O(n/r) \) work and \( O(\log(n/r)) \) time. Finally, Pătraşcu’s sparse bitmap can be computed in parallel, with \( r = (\frac{\log n}{t})^{\Theta(t)} \), in \( O(n + \frac{n}{\log n}) \) work, \( O(t + \log(\frac{n}{\log n})) \) span and \( O(n) \) working space. In the context of succinct trees, the work is \( O(\tau w^c + \frac{\tau w^c t}{\log(\tau w^c)}) \), the span is \( O(t + \log(\frac{\tau w^c t}{\log(\tau w^c)})) \) and the working space is \( O(\tau w^c) \).

**Fusion tree:** A fusion tree stores an array \( A \) of size \( n \) of \( w \)-bit integers, supporting predecessor/successor queries in \( O(\log_w n) \) time. A fusion tree is essentially a B-tree with branching factor \( w^{1/5} \), and therefore, if we can construct a B-tree over the array \( A \) in parallel, we also obtain a parallel algorithm to construct fusion trees. In [125],
Wang and Chen present a parallel algorithm to construct B-trees in $O(\log \log n)$ time, for a sorted list. Since the $\log \tau$ values of the sequence of accumulated weights used to answer `fwd_search` queries are always increasing, we can apply the algorithm described in [125] to construct the B-tree. Henceforth, we will consider the array $A$ as a sorted list of $n$ keys. Although the algorithm of Wang and Chen is based on the EREW model, it can be applied in SMP systems without any modifications. If there are $p$ available cores, the complexity of the algorithm is $O(n/p)$.

Given the sorted list $A$, the algorithm of Wang and Chen constructs an uniquely defined B-tree with branching factor $m$ and the following properties:

- The B-tree has the minimal height $h = \lceil \log_m (n+1) \rceil + 1$
- The root owns $\lceil (n+1)/m^h - 1 \rceil$ keys
- There exists an integer $c$, $1 < c \leq h + 1$, such that all the nodes of the B-tree above the $c$-th level contain $m-1$ keys and all the non-leaf node below the $c$-th level contain $\lceil m/2 \rceil - 1$ keys.
- The leftmost leaf of the B-tree contains $s$ keys, $\lceil m/2 \rceil \leq s \leq m-1$. The rest of the leaves may contain $s$ or $s-1$ keys, but may not own more keys than the leaf node on its left.

With this well-defined B-tree, the parallel algorithm computes the position of each key of $A$ in the B-tree. Each node of the B-tree is identified by its order in a BFS traversal. The details of how to assign a position to each key of $A$ are shown in [125].

Once we have the B-tree, we apply the `sketch` algorithm [45] in parallel on each node of the tree, in $O(1)$ time. Hence, the fusion tree can be computed with $O(n)$ work, $O(\log \log n)$ span and $O(n)$ working space.

In the NS-representation, to support `fwd_search` and `bwd_search` operations, $\tau$ fusion trees are constructed over $\tau$ sorted arrays of $O(\log \tau)$ integers. Considering the previous parallel bounds, the $\tau$ fusion trees can be constructed with $O(\tau \log \tau)$ work, $O(\log \log \tau)$ span and $O(\tau \log \tau)$ working space.

**Range-minimum-query:** In [40], Fisher and Heun present a data structure to answer range minimum/maximum queries in constant-time using $O(n)$ bits over an array $A$ of $n$ elements. The array $A$ is preprocessed by dividing it into $\lceil n/s \rceil$ blocks, $B_1, \ldots, B_{\lceil n/s \rceil}$, of size $s = \lceil \frac{k \log n}{4} \rceil$. A query from $i$ to $j$, $\text{RMQ}(A, i, j)$, is divided into at most three subqueries: One in-block query over the block $B_{\lfloor i/s \rfloor}$, one out-of-block query over the blocks $B_{\lceil i/s \rceil}, \ldots, B_{\lceil j/s \rceil-1}$ and one in-block query over the block $B_{\lfloor j/s \rfloor}$. If $i$ and $j$ belong to the same block, then only one in-block query it is necessary. The in-block queries allow us to obtain the minimum/maximum element inside a block. On the other hand, out-of-block queries allow us to obtain a minimum/maximum element from consecutive blocks.

To answer in-block queries, authors use the fact that each block $B_x$ can be represented by an unique canonical cartesian tree $C_{B_x}^{\text{can}}$. The canonical cartesian tree
of $B_x$ is a cartesian tree with a total order $\prec$ defined as follows: $B_x[i] \prec B_x[j]$ iff $B_x[i] < B_x[j]$, or $B_x[i] = B_x[j]$ and $i < j$. The idea is to precompute all the answers for all $C_s$ possible canonical cartesian trees, where $C_s = \frac{1}{s+1} \binom{2s}{s}$ is the number of the rooted trees on $s$ nodes. Thus, all the answers are stored in a table $P[1, C_s][1, s]$.

The first dimension of the table $P$ corresponds to a descriptor of the blocks of size $s$. For all $\lceil n/s \rceil$ blocks of $A$, their descriptors are stored in an array $T$, requiring $O(s)$ time to compute each one [40].

To answer out-of-block queries, the minimum/maximum element of each block is stored in an array $A'[1, n']$, where $n' = \lceil n/s \rceil$. The array $A'$ is divided into $\lceil n'/s \rceil$ blocks, $B'_1, \ldots, B'_{\lceil n'/s \rceil}$. A RMQ query over $A'$ is answered as before: one out-of-block query and two in-block queries. The in-block queries can be answered by computing the descriptor of each block of $A'$, storing them in an array $T'$ and reusing the lookup table $P$. To answer the out-of-block queries of $A'$, a two-level storage scheme is used.

$s$ contiguous blocks of $A'$ are grouped into a superblock consisting of $s' = s^2$ elements. We precompute all the answers in $A'$ that cover at least one such superblock and store them into a table $M$. Similarly, we precompute all the answers in $A'$ that cover at least one block, but not over a superblock and store them into a table $M'$. Thus, to find the minimum/maximum element inside a superblock, we need to use the table $M$ twice. Summarizing, an out-of-block query of $A$ can be decomposed into two in-block queries in $A'$ (using $T'$ and $P$), two out-of-block queries in $A'$ (using $M'$) and one out-of-superblock query in $A'$ (using $M$).

Finally, the solution of Fisher and Heun has $O(n)$ construction time, $O(\lg^3 n)$ construction space (over the $O(n)$ space of the structure) and $O(1)$ query time.

Since the minimum/maximum operation is associative, we can use a domain decomposition strategy to parallelize the construction of the solution of Fisher and Heun. Thus, we can obtain a parallel solution with $T_1 = O(n)$ work, $T_\infty = O(\lg n)$ span and the same space complexity. The term $O(\lg n)$ is due to the traversal of the blocks of size $s = \lceil \frac{\lg n}{4} \rceil$, which is done sequentially.

In the context NS-representation, we need to answer queries over the root of the $\tau$ RMMTs. Therefore, to answer range minimum/maximum queries we can construct the solution in [40] with $O(\tau)$ work, $O(\lg \tau)$ span and $O(\tau + \lg^3 \tau)$ working space.

Degree, Child and Childrank operations: To support degree, child and child_rank, we need to compute marked blocks. Remember that pioneers are the tightest matching pairs of parentheses $(i, j)$, with $j = \text{find_close}(i)$, such that $i$ and $j$ belong to different blocks. A marked block is a block that has the opening parenthesis of a pioneer $(i, j)$ such $i$ and $j$ do not belong to consecutive blocks. To compute such marked blocks, we need to apply the find_close operation over all the $\tau$ blocks.

Since the find_close operation can be computed in constant time, all marked blocks can be computed with $O(\tau)$ work and $O(1)$ span. The child and child_rank operations additionally need to construct a sparse bitmap $C$ for each marked block, which encodes the number of children of the marked block, in left-to-right order, as gaps of 0’s between 1’s. Therefore, to construct each bitmap it is enough to find the position
of each 1 in the bitmap. To do so, we perform a parallel prefix sum over the blocks fully contained in a marked block. Let \( j \) be a marked block. The bitmap \( C_j \) of \( j \) can be constructed as follows: for each block \( j' \) fully contained in \( j \), if \( j' \) has at least one child of \( j \), we obtain the number of children of \( j' \). If \( j' \) is not a marked block, then the number of children corresponds to \( n_{j'} \) (the number of minima of block \( j' \)); if \( j' \) is a marked node, the number of children is 1. Notice that if \( j' \) is marked, then the blocks contained in \( j' \) do not have any children of \( j \) and they will not be considered for the rest of the computation of \( C_j \). After that, we perform a parallel prefix sum over the blocks that do have some children of \( j \), considering their left-to-right order.

The result of the prefix sum corresponds to the position of all the 1’s in \( C_j \). The final step is to write, in parallel, all the 1’s where they correspond. Following the same idea, we can compute a bitmap \( C \) that represents the concatenation of all the bitmaps of the marked nodes. The parallel prefix sum is the most expensive step of this algorithm, and its work is \( O(\tau) \), its span is \( O(\lg \tau) \) and its working space is \( O(\tau \lg w c) \) bits, which is dominated by the array of size \( O(\tau) \) used in the prefix sum, where each element uses \( O(\lg w c) \) bits.

Thus, with \( w = \Theta(\lg n) \) we have the following theorem:

**Theorem 2.** A \((2n + O(n/\lg^c n))\)-bit representation of an ordinal tree on \( n \) nodes and its balanced parenthesis sequence can be computed with \( O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c) \) work, \( O(c + \lg(\frac{n c}{\lg^c n})) \) span and \( O(n \lg n) \) bits of working space. This representation supports the operations in Table 3.1 in \( O(\epsilon) \) time, with \( c > 3/2 \).

**Proof.** Each of the \( \tau \) RMMTs can be constructed with \( O(\lg^c n) \) work, \( O(\lg^c n) \) span and \( O(\lg^c n \lg \lg^c n) \) bits of working space using Theorem 2. All the \( \tau \) RMMTs can be constructed with \( O(n) \) work, \( O(\lg \lg^c n) \) span and \( O(n \lg n) \) working space. Using the results of this section, with \( t = c \), the additional data structures can be constructed with \( O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c) \) work, \( O(c + \lg(\frac{n c}{\lg^c n})) \) span and \( O(n + \frac{n}{\lg^c n}) \) working space. Thus, total work is \( O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c) \), the maximum span is \( O(c + \lg(\frac{n c}{\lg^c n})) \) and the total working space is \( O(n \lg n) \) bits.

### 5.4 Experimental Results

In this section we present the experimental results of the implementations of our algorithms PSTA and PFEA.

#### 5.4.1 Experimental setup

We implemented the PSTA and PFEA algorithms in C and compiled it using GCC 4.9 with optimization level -O2 and using the -ffast-math flag. All parallel code was compiled using the GCC Cilk branch. The same flags were used to compile libcds and sds1, the state-of-the-art implementations of the RMMT, which were written in C++.
Table 5.1: Datasets used in the experiments of succinct trees.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of nodes ($n$)</th>
<th>Depth</th>
<th>Max fan-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctree25</td>
<td>33,554,432</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>wiki</td>
<td>249,376,958</td>
<td>5</td>
<td>15,206,668</td>
</tr>
<tr>
<td>prot</td>
<td>335,360,503</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>dna</td>
<td>577,241,094</td>
<td>305</td>
<td>16</td>
</tr>
<tr>
<td>ctree</td>
<td>1,073,741,823</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>osm</td>
<td>2,337,888,180</td>
<td>3</td>
<td>2,042,126,001</td>
</tr>
</tbody>
</table>

Table 5.1 shows the six inputs that we used in our experiments. The first two were suffix trees of the DNA (dna) and protein (prot) data from the Pizza & Chili corpus\(^2\). These suffix trees were constructed using code from \(\text{http://www.daimi.au.dk/~mailund/suffix\_tree.html}\). The next two inputs were XML trees of the Wikipedia dump (wiki)\(^3\) and OpenStreetMap dump (osm)\(^4\). The last two input were complete binary trees of depths 25 (ctree25) and 30 (ctree).

The experiments were carried out on the machine B.

5.4.2 Experimental Results of the PSTA algorithm

To evaluate the performance of our PSTA algorithm, we compare it against \texttt{libcds} [23] and \texttt{sdsl} [51], which are state-of-the-art implementations of the \texttt{RMMT}. Both assume that the input tree is given as a parenthesis sequence, as we do here. Our implementation of the PSTA algorithm deviates from the description in Section 5.2 in that we do not store the array $n'$, since \texttt{libcds} and \texttt{sdsl} do not store it and that the prefix sum computation in line 22 of the algorithm is done sequentially in our implementation. This changes the running time to $O(n/p+p)$ but simplifies the implementation. Since $p \ll n/p$ for the input sizes we are interested in and the numbers of cores available on current multicore systems, the impact on the running time is insignificant. In the experiments, the chunk size $s$ was fixed at 256.

Running time and speed-up

Table 5.2 shows the wall clock times achieved by \texttt{psta}, the sequential version of \texttt{psta}, called \texttt{seq}, \texttt{libcds}, and \texttt{sdsl} on different inputs. Each time corresponds to the median achieved over five non-consecutive runs, reflecting our assumption that slightly increased running times are the result of “noise” from external processes such

\(^2\)\text{http://pizzachili.dcc.uchile.cl}

\(^3\)\text{http://dumps.wikimedia.org/enwiki/20150112/enwiki-20150112-pages-articles.xml.bz2 (January 12, 2015)}

\(^4\)\text{http://wiki.openstreetmap.org/wiki/Planet.osm (January 10, 2015)}
Table 5.2: Running times of the algorithms libcds, sdsl, and PSTA, in seconds. seq corresponds to the sequential execution of PSTA.

<table>
<thead>
<tr>
<th></th>
<th>wiki</th>
<th>prot</th>
<th>dna</th>
<th>ctree</th>
<th>osm</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>1</td>
<td>2.81</td>
<td>4.10</td>
<td>7.15</td>
<td>12.17</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>.72</td>
<td>1.05</td>
<td>1.86</td>
<td>3.05</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>.40</td>
<td>.58</td>
<td>.95</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>.31</td>
<td>.43</td>
<td>.72</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>.24</td>
<td>.32</td>
<td>.55</td>
<td>.85</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>.19</td>
<td>.29</td>
<td>.49</td>
<td>.74</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>.19</td>
<td>.26</td>
<td>.42</td>
<td>.68</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>.16</td>
<td>.25</td>
<td>.43</td>
<td>.62</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>.18</td>
<td>.25</td>
<td>.38</td>
<td>.62</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>.20</td>
<td>.21</td>
<td>.36</td>
<td>.52</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>.21</td>
<td>.23</td>
<td>.35</td>
<td>.50</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>.22</td>
<td>.25</td>
<td>.34</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>.21</td>
<td>.26</td>
<td>.37</td>
<td>.49</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>.27</td>
<td>.30</td>
<td>.36</td>
<td>.50</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>.30</td>
<td>.36</td>
<td>.42</td>
<td>.50</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>.27</td>
<td>.40</td>
<td>.39</td>
<td>.50</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>.30</td>
<td>.33</td>
<td>.38</td>
<td>.54</td>
</tr>
</tbody>
</table>

The differences in running times of the psta algorithm on one core and seq are insignificant. This implies that the overhead of the scheduler is negligible. The psta algorithm on a single core and sdsl outperformed libcds by an order of magnitude. One of the reasons for this is that libcds implements a different version of RMMT including rank and select structures, while psta and sdsl do not. Constructing these structures is costly. On a single core, sdsl was about 1.5 times faster than psta, but neither sdsl nor libcds were able to take advantage of multiple cores, so psta outperformed both of them starting at $p = 2$. The advantage of sdsl over psta on a single core, in spite of implementing essentially the same algorithm, can be attributed to the lack of tuning of psta.

Up to 16 cores, the speed-up of psta with ctree and osm datasets is almost linear whenever $p$ is a power of 2 and the efficiency (speed-up/$p$) is 70% or higher with respect to seq and 60% with respect to sdsl, except for ctree on 32 cores. This is very good for a multicore architecture. When $p$ is not a power of 2, speed-up is slightly worse. The reason is that, when $p$ is a power of 2, psta can assign exactly one subtree to each thread (see Algorithm 12), distributing the work homogeneously as operating system and networking tasks. Figure 5.4 shows the speed-up compared to the running times of seq, and Figure 5.5 shows the speed-up compared to sdsl.
across cores without any work stealing. When the number of threads is not a power of two, some threads have to process more than one subtree and other threads process only one, which degrades performance due to the overhead of work stealing.

There were two other factors that limited the performance of psta in our experiments: input size and resource contention with the OS.

Input size. For the two largest inputs we tested, osm and ctree, speed-up kept increasing as we added more cores. For wiki, prot and dna, however, the best speed-up were achieved with 28, 36 and 44 cores, respectively. Beyond this, the amount of work to be done per thread was small enough that the scheduling overhead caused by additional threads started to outweigh the benefit of reducing the processing time per thread further.

Resource contention. For $p < 64$, at least one core on our machine was available to OS processes, which allowed the remaining cores to be used exclusively by psta. For $p = 64$, psta competed with the OS for available cores. This had a detrimental effect on the efficiency of psta for $p = 64$.

The network topology of our machine may also impact in the performance of our algorithm. In Chapter 7, we will discuss about the relationship of the topology of multicore machines and the performance of parallel algorithms.

Memory usage

We measured the amount of working memory (i.e., memory not occupied by the raw parenthesis sequence) used by psta, libcds, and sds1. We did this by monitoring how much memory was allocated/released with malloc/free and recording the peak usage. For psta, we only measured the memory usage for $p = 1$. The extra memory needed for thread scheduling when $p > 1$ was negligible. The results are shown in the Figure 5.6. Even though psta uses more memory than both libcds and sds1,
Figure 5.6: Memory consumption of the algorithms psta, libcds and sdsl.

The difference between psta and sdsl is a factor of less than 1.3. The difference between psta and libcds is no more than a factor of three and is outweighed by the substantially worse performance of libcds. The reduced working space used by libcds is due to the fact that its implementation does not store the array of excess values. Instead, libcds stores rank/select structures over the input bit vector $P$, computing excess values with $excess(i) = 2 \times rank_1(P, i) - i$, where $rank_1(P, i)$ gives the number of 1s on $P$ up to the index $i$. Part of the higher memory usage of psta stems from the allocation of $e'$, $m'$ and $M'$ arrays which store the partial excess values in the algorithm. Storing these values, however, is a key factor that helps psta achieve very good performance. The space used by our algorithm can be reduced by storing local excess values in the array $e'$, instead of global values. However, reducing the space in such a way will complicate the implementation of the queries over the RMMT.

5.4.3 Experimental Results of the PFEA algorithm

Table 5.3 shows the running times of the PFEA algorithm with the datasets ctree25, prot and dna. To compute the speedups, we used times obtained by seq. The best parallel times are identified using a bold typeface.

Figure 5.7 shows the corresponding speedup of the PFEA algorithm. Up to 16 threads, the speedup is almost linear, obtaining at least 49% of efficiency (speedup/$p$) for the ctree25 dataset, that is, our algorithm reaches at least 49% of the linear speedup (the ideal). With more than 16 threads, the performance of our algorithm is poor, reaching at most 16% of efficiency for the prot algorithm and 64 threads. The poor efficiency of our algorithm is not explained by the DYM model. We think that it can be explained by its low workload. Algorithms with a low workload do not scale properly since the workload of their parallel tasks is not enough to pay the overhead.
Table 5.3: Running times of PFEA algorithm, in seconds. seq corresponds to the sequential execution of PFEA. Columns with the superscript +16 and +32 represent the running times of PFEA algorithm by artificially increasing the workload with 16 and 32 CAS operations per edge, respectively. The best parallel times are shown using bold typeface.

<table>
<thead>
<tr>
<th>p</th>
<th>ctree25</th>
<th>prot</th>
<th>dna</th>
<th>ctree25+16</th>
<th>prot+16</th>
<th>dna+16</th>
<th>ctree25+32</th>
<th>prot+32</th>
<th>dna+32</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>5.67</td>
<td>52.87</td>
<td>31.33</td>
<td>55.04</td>
<td>473.92</td>
<td>275.06</td>
<td>102.94</td>
<td>887.61</td>
<td>514.73</td>
</tr>
<tr>
<td>1</td>
<td>6.07</td>
<td>34.83</td>
<td>57.29</td>
<td>54.99</td>
<td>275.23</td>
<td>474.07</td>
<td>102.80</td>
<td>514.09</td>
<td>886.01</td>
</tr>
<tr>
<td>4</td>
<td>1.68</td>
<td>9.35</td>
<td>15.62</td>
<td>14.22</td>
<td>70.84</td>
<td>121.92</td>
<td>26.15</td>
<td>30.30</td>
<td>224.86</td>
</tr>
<tr>
<td>8</td>
<td>1.05</td>
<td>5.77</td>
<td>9.80</td>
<td>7.33</td>
<td>36.39</td>
<td>62.79</td>
<td>13.32</td>
<td>66.35</td>
<td>114.42</td>
</tr>
<tr>
<td>12</td>
<td>0.87</td>
<td>3.99</td>
<td>6.90</td>
<td>5.02</td>
<td>25.18</td>
<td>43.23</td>
<td>8.99</td>
<td>45.02</td>
<td>77.49</td>
</tr>
<tr>
<td>16</td>
<td>0.73</td>
<td>3.66</td>
<td>5.57</td>
<td>3.91</td>
<td>19.59</td>
<td>33.50</td>
<td>6.86</td>
<td>34.37</td>
<td>59.15</td>
</tr>
<tr>
<td>20</td>
<td>0.76</td>
<td>3.42</td>
<td>5.53</td>
<td>3.21</td>
<td>16.06</td>
<td>27.61</td>
<td>5.58</td>
<td>28.16</td>
<td>48.26</td>
</tr>
<tr>
<td>24</td>
<td>0.76</td>
<td>3.41</td>
<td>5.25</td>
<td>2.77</td>
<td>13.91</td>
<td>24.03</td>
<td>4.73</td>
<td>23.96</td>
<td>41.04</td>
</tr>
<tr>
<td>28</td>
<td>0.66</td>
<td>3.28</td>
<td>5.36</td>
<td>2.47</td>
<td>12.40</td>
<td>21.23</td>
<td>4.16</td>
<td>20.87</td>
<td>36.05</td>
</tr>
<tr>
<td>32</td>
<td>0.67</td>
<td>3.37</td>
<td>5.71</td>
<td>2.30</td>
<td>12.52</td>
<td>19.49</td>
<td>4.08</td>
<td>18.62</td>
<td>35.22</td>
</tr>
<tr>
<td>36</td>
<td>0.68</td>
<td>3.23</td>
<td>5.57</td>
<td>2.27</td>
<td>11.61</td>
<td>19.95</td>
<td>3.71</td>
<td>18.70</td>
<td>32.29</td>
</tr>
<tr>
<td>40</td>
<td>0.68</td>
<td>3.15</td>
<td>5.48</td>
<td>2.16</td>
<td>10.91</td>
<td>18.75</td>
<td>3.38</td>
<td>17.17</td>
<td>29.59</td>
</tr>
<tr>
<td>44</td>
<td>0.68</td>
<td>3.16</td>
<td>5.44</td>
<td>2.04</td>
<td>10.20</td>
<td>17.70</td>
<td>3.19</td>
<td>15.98</td>
<td>27.56</td>
</tr>
<tr>
<td>48</td>
<td>0.69</td>
<td>2.97</td>
<td>5.19</td>
<td>1.92</td>
<td>9.67</td>
<td>16.79</td>
<td>3.00</td>
<td>15.02</td>
<td>25.82</td>
</tr>
<tr>
<td>52</td>
<td>0.64</td>
<td>3.05</td>
<td>5.37</td>
<td>1.84</td>
<td>9.23</td>
<td>16.01</td>
<td>2.83</td>
<td>14.12</td>
<td>24.33</td>
</tr>
<tr>
<td>56</td>
<td>0.65</td>
<td>3.01</td>
<td>5.33</td>
<td>1.79</td>
<td>8.83</td>
<td>15.18</td>
<td>2.71</td>
<td>13.37</td>
<td>22.87</td>
</tr>
<tr>
<td>60</td>
<td>0.61</td>
<td>3.19</td>
<td>5.35</td>
<td>1.72</td>
<td>8.46</td>
<td>14.75</td>
<td>2.58</td>
<td>12.63</td>
<td>21.82</td>
</tr>
<tr>
<td>64</td>
<td>0.71</td>
<td>3.05</td>
<td>5.18</td>
<td>1.80</td>
<td>8.29</td>
<td>15.13</td>
<td>2.55</td>
<td>12.32</td>
<td>20.90</td>
</tr>
</tbody>
</table>

of thread scheduling and memory transfers. In the case of the PFEA algorithm, each edge takes part of only a few comparisons and assignments. Therefore, the workload for each parallel task is not enough to take advantage of the 64 threads, even when we create $\Theta(p)$ parallel tasks. To demonstrate that the low workload is the reason of the low efficiency, we increased artificially the workload of our implementation. Between the lines 5 and 9, we added 16 and 32 CAS operations. On each iteration of the loop of line 5, each CAS operation was executed over $ET[i]$, increasing the workload for each edge. The complexity and correctness of our algorithm do not change with the addition of these extra operations. Columns 5–10 of Table 5.3 show the resulting running times after adding 16 and 32 extra operations. Figures 5.8 and 5.9 show the corresponding speedup. With 16 extra operations, the efficiency was at least 48% for the ctree25 dataset and 64 threads. For 16 threads, the efficiency increased, reaching a 88% for the ctree25 dataset. With 32 extra operations, the efficiency was at least 63% up to 64 threads and 94% up to 16 threads.

Another factor that, we think, limited the performance of the PFEA algorithm was the topology of the experiment. As was mentioned before, our machine has four processors connected in a grid topology, which involves communication costs among processors. Each processor executes up to 16 threads. We observe that in the Figure 5.7, the algorithm scales up to 16 threads. With more threads, the communication costs may affect the scalability. With more workload, Figure 5.8 shows a linear scalability up to 32 threads. After 32 threads, the efficiency of the algorithm decreases. For the experiment with 32 extra operations, Figure 5.9 shows a similar behavior, with the difference that after 32 threads, the efficiency is better than in Figure 5.8. For 64 threads, all the speedups have a slowdown, since the PFEA algorithm has to compete
Figure 5.7: Speedup of the PFEA algorithm with datasets ctree25, dna and prot.

Figure 5.8: Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 16 CAS operations per edge.

Figure 5.9: Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 32 CAS operations per edge.

with the OS for the available cores. In Chapter 7 we will discuss more about the effects of the machine topology in the performance of the PFEA algorithm.

5.4.4 Discussion

For domains where trees have billions of nodes, the psta algorithm exhibits a good speed-up up to 64 cores. The speed-up is degraded for trees with fewer nodes. However, even in such cases, our algorithm reaches good speed-up up to 32 cores. Additionally, our algorithm outperforms the state-of-the-art implementations using only $p = 2$ threads. Considering all of this, the psta algorithm is a good option to construction succinct trees in commodity multicore architectures.

With respect to the working space, our psta algorithm is competitive with sds1 and it does not use more than three times the memory used by libcds, which is the slowest algorithm. Despite our algorithm using more memory than sds1 and libcds, it is up to 20 times and 376 times faster with 64 cores, respectively.

The scalability of the PFEA algorithm up to 16 threads is good in practice (nearly
50% efficiency). However, the lack of workload of our algorithm prevents it from obtaining a good practical scalability with more threads. In this kind of algorithms, we cannot expect a better scalability adding more threads. Nevertheless, this poses an interesting problem: For a given algorithm, find the maximum number of threads that achieve at least a 50% efficiency. Once we find such number, the rest of the threads may be used potentially in other procedures. The implementation of multicore algorithms is non-trivial, since it needs to take care about the communication costs, memory hierarchy, cache coherency, etc, which may affect the performance. Therefore, we consider a parallel algorithm with a 50% efficiency a good parallel implementation.
Chapter 6
Parallel Construction of Succinct Triangulated Plane Graphs

In this chapter we will study the parallel construction of succinct representations of triangulated plane graphs. In Section 6.1 we present and discuss the parallel construction of succinct representation based on canonical orderings. In Sections 6.1.1 and 6.1.2 we introduce our parallel algorithms to compute the parentheses representation of a triangulated plane graph, given a canonical ordering. In Section 6.1.3 we discuss the challenges of computing canonical orderings of triangulated plane graph on SMP systems. In Section 6.2 we discuss how to extend the results of Section 6.1 to succinct representations based on realizers.

6.1 Succinct representation of triangulated plane graphs via canonical ordering

Let $G = (V, E)$ be a triangulated plane graph, with $|V| = n \geq 3$, $|E| = m$, with a canonical ordering $\Pi$ and canonical spanning tree $T_{co}$ (see Sections 6.1.3 and 6.1.1, where we discuss how to compute the canonical ordering and the canonical spanning tree in parallel, respectively). We compute a succinct representation of $G$ by obtaining in parallel its parentheses representation based on the canonical ordering and then, again in parallel, the succinct representation of such parentheses representation.

6.1.1 Parallel computation of the multiple parentheses representation $S_{co}$

In Section 3.2.3 we described how to compute the string of two types of parentheses, $S_{co}$, based on the work of [20, 19, 64]. The definition of the construction of $S_{co}$, introduced before in 3.2.3, is:

- $S_{co} = FE(T_{co})$.
- For each vertex $v_i$ of $T_{co}$, count the number of lower-numbered neighbors, $l_i$, and higher-numbered neighbors, $h_i$, of $v_i$ in $G \setminus T_{co}$.
- For each vertex $v_i$ of $T_{co}$, write $l_i$ “]”’s right after ($i$ and $h_i$ “[”’s right after )$i$.

Considering this definition, we can adapt the PFEA algorithm (Algorithm 10) to compute the string $S_{co}$ in parallel (see Algorithm 15). We call this algorithm the Parallel graph encoding algorithm (PGEA). The input spanning tree $T_{co}$ is represented by an array of vertices, $V_T$, and an array of edges, $E_T$. Each vertex $v \in V_T$ stores two indices, $v.first$ and $v.last$, to $E_T$, indicating the adjacency list of $v$, sorted counterclockwise around $v$, starting with $v$’s parent edge. Note that $(v.last - v.first +
Input : An adjacency list representation of the plane graph $G$ consisting of arrays $V_G$ and $E_G$, with a canonical ordering, and the number of threads, threads. 
Output: An adjacency list representation of the canonical spanning tree $T_{co}$ consisting of arrays $V_T$ and $E_T$.

1) is the degree of $v$. Each edge $e \in E_T$ has three fields, $e.src$ which is a pointer to the source vertex, $e.tgt$ which is a pointer to the target vertex and $e.cmp$ which is the position in $E_T$ of the complement edge, $e'$, of $e$, where $e'.src = e.tgt$ and $e'.tgt = e.src$. For $x \in \{e.src, e.tgt\}$, we use $next(x)$, $first(x)$ and $last(x)$ to denote the indices in $E_T$ of $e$’s successor, of the first element (parent edge) and of the last element in $x$’s adjacency list, respectively. The three of them are easily computed in constant time by following pointers, see Figure 5.1 for an example of the representation. We also use $co(v)$ to denote the canonical ordering of the vertex $v$. The representation of the graph $G$ is similar.

Given the canonical ordering, the input canonical spanning tree can be computed easily in parallel using Algorithm 14. Intuitively, the parent of each vertex in $G$ is its leftmost neighbor with lower canonical ordering. Formally, for each vertex $v \in V_G$, the parent of $v$ is given as follows (lines 4-7 Algorithm 14): Let $n^v_1, n^v_2, \ldots$ be the neighbors of $v$ in counterclockwise order. The parent of $v$ is its neighbor at the $j$-th position, such that $co(v) < co(n^v_{j-1})$ and $co(v) > co(n^v_j)$. Since the number of edges of $G$ is $3n - 6$, the work of Algorithm 14 is $O(n/p)$ and its span is $O(1)$.

Algorithm PGEA creates three arrays, an auxiliary array $C$ to store the lower-numbered and higher-numbered neighbors of each vertex of $G$, an auxiliary array $LE$ to store the Euler Tour of $T_{co}$ and the array $S_{co}$ to store the parentheses representation induced by $G \setminus T_{co}$ and $T_{co}$. The first step of Algorithm 15 is, for each vertex $v$ of $G$, to count the number of lower and higher neighbors of $v$ in $G \setminus T_{co}$ (see the definition at the beginning of this section), considering the canonical ordering of $G$. The values of lower-numbered and higher-numbered neighbors of $v$ are stored in $C[v].l$ and $C[v].h$, respectively (lines 6 and 7). Counting lower and higher neighbors can be done using a parallel prefix sum algorithm. The second step is to traverse $T_{co}$. Each entry in $LE$ represents the traversal of an edge of $T_{co}$ and stores three variables: value is (“)” followed by $C[v].l$ close brackets, or “(”)” followed by $C[v].h$ open brackets, depending on whether the edge is a forward or a backward edge. Variable succ stores the index in $LE$ of the next edge in the Euler tour. Finally, variable rank is the number of
Input: An adjacency list representation of the canonical spanning tree $T_{co}$ consisting of arrays $V_T$ and $E_T$, an adjacency list representation of the plane graph $G$ consisting of arrays $V_G$ and $E_G$ and the number of threads, threads.

Output: A two-type parentheses sequence $S_{co}$ induced by $G \setminus T_{co}$ and $T_{co}$.

1. $C = \text{an array of length } |V_G|$
2. $LE = \text{an array of length } |E_T|$
3. $S_{co} = \text{an array of length } |E_G| + 2$
4. $chk = |E_T|/\text{threads}$
5. parfor $i = 0 \text{ to } |V_G| - 1$ do
6. \hspace{1em} $[C[i].l, C[i].h] = \text{parallelCount}(V_G, E_G, V_T, i)$
7. parfor $t = 0 \text{ to } \text{threads} - 1$ do
8. \hspace{2em} for $i = 0 \text{ to } chk - 1$ do
9. \hspace{3em} $j = t * chk + i$
10. \hspace{3em} if $\text{co}(E_T[j].src) < \text{co}(E_T[j].tgt)$ then \hspace{1em} // forward edge
11. \hspace{4em} $LE[j].value = \oplus(\text{“} , \text{“} ) * C[E_T[j].tgt].l$
12. \hspace{4em} $LE[j].rank = C[E_T[j].tgt].l + 1$
13. \hspace{3em} if $E_T[j].tgt$ is a leaf then
14. \hspace{4em} $LE[j].succ = E_T[j].cmp$
15. \hspace{3em} else
16. \hspace{4em} $LE[j].succ = \text{first}(E_T[j].tgt) + 1$
17. \hspace{3em} else \hspace{1em} // backward edge
18. \hspace{4em} $LE[j].value = \oplus(\text{“} , \text{“} ) * C[E_T[j].src].h$
19. \hspace{4em} $LE[j].rank = C[E_T[j].src].h + 1$
20. \hspace{3em} if $E_T[j]$ is the last edge in the adjacency list of $E_T[j].src$ then
21. \hspace{4em} $LE[j].succ = \text{first}(E_T[j].tgt)$
22. \hspace{3em} else
23. \hspace{4em} $LE[j].succ = \text{next}(E_T[j].tgt)$

parallel_list_ranking($LE$)
25. parfor $t = 0 \text{ to } \text{threads} - 1$ do
26. \hspace{1em} for $i = 0 \text{ to } chk - 1$ do
27. \hspace{2em} $j = t * chk + i$
28. \hspace{2em} $S_{co}[LE[j].rank \ldots LE[j + 1].rank - 1] = LE[j].value$
29. \hspace{1em} $S_{co}[0] = \text{“}($$
30. \hspace{1em} $S_{co}[|E_G| + 1] = \text{“})$$

Algorithm 15: Parallel graph encoding algorithm (PGEA)

Parentheses and brackets in value, used to compute the rank of each symbol in $S_{co}$.

In a canonical spanning tree, the canonical ordering of a vertex is lower than the canonical ordering of its children. So, for an edge $e$ of $T_{co}$, if $\text{co}(e.src)$ is lower than $\text{co}(e.tgt)$, then $e$ is a forward edge. Otherwise, $e$ is a backward edge. For a forward edge $e \in E_T$, we write a “(”, representing the open parenthesis of $e.tgt$, followed by
$C[e tgt].l$ “]”s, representing the lower-numbered neighbors of $e tgt$. This is done in line 11, where $\oplus$ represents a concatenation function and “]” $*C[E_T[j].tgt].l$ represents the string composed by $C[E_T[j].tgt].l$ symbols “]”. Line 12 sets the rank, while lines 13 to 16 set the next edge of $e$ in the Euler tour. For backward edges, the procedure is similar (Lines 17 to 23). Line 24 computes ranks using a parallel list ranking algorithm [65]. Given these ranks, the parentheses and brackets representation can be obtained by writing $LE[i].value$ into the range $S[LE[i].rank ... LE[i + 1].rank - 1]$. Lines 25 to 28 do exactly this. Finally, open and closed parentheses are written in the first and last position of $S_{co}$, respectively, representing the root of $T_{co}$ (lines 29 and 30).

The theoretical analysis of the PGEA algorithm is similar to the analysis of the PFEA algorithm. In the PGEA algorithm, the parallel counting (lines 5–6) of lower-numbered and higher-numbered neighbors of all vertices of $G$ can be done with $O(n)$ work, $T_p = O(n/p + \lg p)$ and $T_\infty = O(\lg n)$. Lines 7–23 perform $O(n)$ work, have $T_p = O(n/p)$ and span $T_\infty = O(1)$. The whole computation could have been formulated as a single parallel loop. However, in the interest of limiting scheduling overhead, we create only as many parallel threads as necessary. Line 24 performs a parallel list ranking with $O(n)$ work, $T_p = O(n/p + \lg p)$ time and $O(\lg n)$ span. Since we previously compute the position of each parenthesis and bracket, we can make the assignment of line 28 in constant time. Therefore, lines 25–28 perform $O(n)$ work, $T_p = O(n/p)$ and span $T_\infty = O(1)$. Thus, the total work is $T_1 = O(n)$ and the span is $T_\infty = O(\lg n)$. The running time on $p$ cores is $T_p = O(n/p + \lg p)$.

To encode $S_{co}$ using the bit-vectors $S_1$, $S_2$ and $S_3$, we can use the PGEA algorithm as follows:

- $S_1$ can be obtained by writing “1” instead of “(” and “)”, and “0” instead of “[" and “]”, in the field value.

- $S_2$ can be obtained by writing always 1 in the field rank (lines 12 and 19), by writing “1” in the field value for forward edges and “0” for backward edges. Notice that $S_2$ corresponds to the folklore encoding of $T_{co}$, so, we also can use the PFEA algorithm to compute it.

- Finally, to obtain $S_3$, for forward edges we assign to the field rank the number of lower-numbered neighbors and to the field value a “0”, and for backward edges we assign to the field rank the number of higher-numbered neighbors and “1” to the field value.

Therefore, $S_1$, $S_2$ and $S_3$ can be computed with the same complexities as $S_{co}$. Indeed, $S_1$, $S_2$ and $S_3$ can be computed at the same time, by reusing the array $C$ and defining different $LE_1$ and $LE_2$ arrays for $S_1$ and $S_2$. $S_3$ can be computed by using $LE_1$ and $LE_2$ arrays.
6.1.2 Parallel construction of the succinct representation of the multiple parentheses sequence $S_{co}$

As was discussed in Section 3.2.3, operations over the sequence $S_{co}$ can be reduced to rank, select and enclose operations over the bit-vectors $S_1$, $S_2$ and $S_3$. Since all of these operations are supported by the solution proposed in [102], based on the Range Min-Max Tree, we can construct succinct representations of $S_1$, $S_2$ and $S_3$ in parallel by using our results in Chapter 5. Thus, the construction of the succinct representation of bit-vectors $S_1$, $S_2$ and $S_3$ can be done with $O(n)$ work, $O(lg n)$ span and $O(n lg n)$ working space, supporting operations in logarithmic time. Alternatively, the succinct representation of $S_1$, $S_2$ and $S_3$ can be done with $O(n + \frac{n}{lg n} lg(\frac{n}{lg n}) + c^c)$ work, $O(c + lg(\frac{n}{lg n}))$ span and $O(n lg n)$ working space, supporting operations in $O(c)$ time, where $c > 3/2$.

6.1.3 Two approaches to compute canonical orderings in parallel

The problem of computing the canonical ordering of $G$ in $T_p = O(n/p)$ for SMP systems is still open. In this section, we discuss two approaches to compute the canonical ordering in parallel: One approach is based on graph decomposition, where a parallel algorithm to compute the decomposition is still pending, and other approach based on parallel breadth-first traversal, with the problem that its theoretical speedup is low.

Parallel computation of canonical ordering based on graph decomposition

The idea behind this solution is the following: Decompose the set of vertices $V$ of $G$ into disjoint subsets and then compute the canonical ordering of the subgraphs induced for each subset, at the same time. If the subgraphs are enumerated according to their topology, the composition of the canonical orderings of each subgraph will be a canonical ordering of $G$.

Let $V_0, V_1, \ldots, V_k$, with $k = O(p)$, be a decomposition of $V$ with the following properties:

(a) $\forall i \in \{0, \ldots, k\}, |V_i| = O(n/p)$

(b) $\forall i, j \in \{0, \ldots, k\}, i \neq j, V_i \cap V_j = \emptyset$

(c) $V = V_0 \cup \ldots \cup V_k$.

(d) Let $G_i$ be the subgraph induced by $V_0 \cup \ldots \cup V_i$, with $i \in \{0, \ldots, k\}$. $G_i$ is 2-connected.

(e) $\forall v \in V$, all neighbors of $v$ that belongs to the same subset $V_i$ appear consecutively on the adjacency list of $v$. 
(a) Graph decomposition that violates property (e).
(b) Graph decomposition that violates property (f).

Figure 6.1: Example of graph decompositions that do not meet the properties.

(f) Let \( \tilde{G}_i \) be the subgraph induced by \( V_i \) and let \( \tilde{C}_i \) be the contour of \( \tilde{G}_i \). The neighbors of \( \tilde{C}_i \) do not belong to the same subgraph.

The properties (a) and (b) imply that the subgraphs \( \tilde{G}_i \) have a similar amount of vertices and that they do not share vertices. Property (c) implies that all the vertices belong to a subgraph \( \tilde{G}_i \). Property (d) implies that a subgraph \( \tilde{G}_{i-1} \) has at least two vertices incident to the edges of the consecutive subgraph \( \tilde{G}_i \). Property (e) allows us to prove the next Proposition 2. Finally, property (f) implies that any subgraph \( G_i \) cannot be wrapped by other subgraph. These properties induce an order between the subgraphs.

For example, Figure 6.2 shows an example of a decomposition that follows all the properties. Figure 6.1 shows two examples that violate the properties.

**Proposition 2.** Let \( \tilde{G}_i \) be the subgraph induced by \( V_i \), with \( i \in \{0, \ldots, k\} \). Let \( \Pi_i \) be a canonical ordering of the subgraph \( \tilde{G}_i \). It is possible to obtain a canonical ordering of \( G \) considering the canonical ordering of each subgraph of \( \tilde{G} \).

**Proof.** Without loss of generality, assume that vertices \( v_1 \) and \( v_2 \) and are in subgraph \( \tilde{G}_0 \) and \( v_n \) is in the subgraph \( \tilde{G}_k \).

The canonical ordering \( \Pi_i \) of each subgraph \( \tilde{G}_i \) can be computed using a variation of the sequential algorithm introduced in Section 3.2.3. The main difference is that this new algorithm does not start labelling all the vertices with \(-1\). Instead, we distinguish between two kind of vertices: marked vertices and unmarked vertices. For a vertex \( u \in V_i \), \( u \) is marked if it is incident to two vertices \( v \in V_j \) and \( w \in V_k \) such that \( i > j \) and \( i > k \); otherwise, \( u \) is unmarked. In Figure 6.2, gray vertices are marked. In the array of labels of each partition, all unmarked vertices will be labelled with \(-1\) and marked vertices with \(1\). The vertices of the first partition, \( \tilde{G}_0 \), will be
Figure 6.2: Decomposition of a triangulated plane graph.

labelled with \(-1\), except for vertices \(v_1\) and \(v_2\). After that, the sequential algorithm is applied to each partition. Observe that this algorithm can be parallelized easily: Finding all the marked nodes and processing each partition in parallel.

Once we have the canonical orderings \(\Pi_0, \Pi_1, \ldots, \Pi_k\), we can define the canonical ordering of \(G\), \(\Pi\), as follows:

- \(\Pi\) must begin with the paths in the canonical ordering of \(\tilde{G}_0\), i.e. \(\Pi = (\Pi_0)\).

- Subsequently, add the canonical ordering of \(\tilde{G}_1\), \(\Pi = (\Pi_0, \Pi_1)\) and repeat the same process until the last subgraph.

As \(\Pi_0\) is a canonical ordering and the first vertex in \(\Pi_1\), \(v\), is incident to \(\Pi_0\), \(\Pi_0 \cup v\) is a canonical ordering of \(G_0 \cup v\). At the same time, \(v\) is part of the canonical ordering \(\Pi_1\), then \((\Pi_0 \cup \Pi_1)\) is a canonical ordering of \(G_1\). Therefore, following this procedure, we can obtain the canonical ordering of \(G_k = G\).

Graph decomposition algorithms based on the decomposition of the spanning tree of the graph cannot be used to compute the decomposition introduced in this section. If we cut the paths of a spanning tree only based in the number of nodes per path, then we cannot ensure the properties (e) and (f). The design of the algorithm to compute such decomposition is left as an open problem.

Parallel computation of canonical ordering based on breadth-first search

An alternative to compute the canonical ordering of a maximal plane graph \(G\), with external vertices \(v_1, v_2\) and \(v_n\), is to parallelize the sequential algorithm explained in 3.2.3. Notice that the sequential algorithm uses labels to identify the vertices that can be included in the canonical ordering and more than one vertex may be ready
Figure 6.3: Parallel computation of canonical orderings based on dual graphs and BFS traversal.
to be included. In the parallel version of this algorithm, all the vertices that are ready to be part of the canonical ordering will be added, no matter if we have more than one. To improve the identification of the vertices that are ready, we propose to use the dual graph of \( G \). Once we have the dual graph, we perform a parallel breadth-first search (BFS) over the dual, with the following conditions: (1) An edge \( e \) will be traversed if and only if it is at distance 1 (the traditional condition of the BFS) and (2) the target vertex of \( e \) is ready to be added. When a vertex \( v \) is ready, all its neighbors that have been processed are consecutive in the adjacency list of \( v \).

Each time when we traverse a new edge during the BFS, we are discovering a new face \( f \) of \( G \). This face \( f \) has an useful property: the face \( f \) contains one vertex that has at least two consecutive edges incidents to previous discovered faces. With this property, we can compute a canonical ordering of \( G \) by adding at most one vertex to the canonical ordering for each discovered face. Figure 6.3 shows an illustration of the idea. In Figures 6.3a and 6.3b, the dual graph is computed. Then, in step 2, the traversal starts from the vertex that represents the external face of \( G \). To obtain a correct algorithm, we choose, arbitrarily, one outgoing edge of the initial vertex. The other two edges will not be considered in the traversal (white vertices in the figure). In the same step, we discover the first face, adding vertices \( a, b \) and \( c \) to the canonical ordering. Then, in step 3, we discover two new faces, adding at the same time vertices \( d \) and \( f \) to the canonical ordering. In the next step, we add vertex \( e \). Observe that using a traditional BFS algorithm, we should traverse the edges associated to the edges \( (a, d) \) and \( (b, f) \) of \( G \). However, we do not traverse such edges, because the target vertex \( h \) is not ready to be added. Finally, in step 5 we add the last vertex \( h \), discovering four new faces. Thus, we obtain canonical ordering \( \Pi = \{ a, b, c, d, f, e, h \} \).

At some point, it is possible to discover new faces that do not add new vertices to the canonical ordering, but it does not affect the correctness of the algorithm.

The dual graph \( G_D = (V_{G_D}, E_{G_D}) \) of a maximal plane graph \( G = (V_G, E_G) \), can be computed in parallel using Algorithm 16, called \textit{Parallel dual graph algorithm} (PDGA). The algorithm is based on the property that each vertex of \( G_D \) has degree 3, since \( G \) is a maximal plane graph and the number of edges of \( G_D \) is the same as the number of edges in \( G \). The PDGA algorithm assumes that the input graph will be represented in its adjacency list representation, where the indices of the vertices are unique and consecutive. Since each vertex of \( G_D \) will have degree 3, lines 4–6 of the algorithm sets the first and last fields of each vertex. Thus, the adjacency list of a vertex \( v_i^d \) of the dual graph, with index \( i \), will be stored in \( E_{G_D} \) at indices \( 3i, 3i + 1 \) and \( 3i + 2 \). For the rest of the algorithm, we need a more precise definition of a face and the concept of ownership. A face \( f : (e_1, e_2, e_3) \) corresponds to three edges, ordered in counterclockwise order, where \( e_1.tgt = e_2.src, e_2.tgt = e_3.src, e_3.tgt = e_1.src \) and \( e_1.src < e_2.src < e_3.src \). An edge \( e \) of \( G \) is the owner of a face \( f = (e_1, e_2, e_3) \) if \( e = e_1 \). For example, in Figure 6.3a, assuming a lexicographic order in the vertices, the edge \( (c, f) \) is the owner of the face \( \langle (c, f), (f, e), (e, c) \rangle \) and the edge \( (c, e) \) is the owner of the face \( \langle (c, e), (e, d), (d, c) \rangle \). In lines 7–9 of the PDGA algorithm and in the Function \textit{ownership}, the ownership of each face in \( G \) is tested.
Each time when an owner edge is found, the algorithm marks the index of such an edge in an array $A$ with a 1. Then, with all the owner edges detected – one per vertex in $G_D$ – the algorithm performs a prefix sum over $A$ to determine the final position of each face in $V_{G_D}$. The final step (lines 11–16) is to connect the edges of the dual graph with the corresponding nodes in $V_{G_D}$. Given an edge $e$ of $G$, we say that its associated edge in $G_D$ is the one that connects the faces where $e$ and its complement, $e.cmp$, belong. For a vertex $v^d_i$ of $G_D$, associated to the face $f = \langle e_1, e_2, e_3 \rangle$, the associated edge of $e_1$ is stored at position $3i$, the associated edge of $e_2$ is stored at position $3i + 1$ and the associated edge of $e_3$ is stored at position $3i + 2$, respecting the relative position of the edges $e_1$, $e_2$ and $e_3$ in $f$. Such relative position is given by the Function newIndex. This algorithm has $O(n)$ work and $O(\lg n)$ span.

Leiserson and Schardl introduced in [87] a parallel algorithm to compute the BFS traversal of a graph, called PBFS. The authors proposed to replace the traditional FIFO data structure, used in the most classical BFS algorithm, by a thread-safe data structure called bag, which supports insert, union and split operations in parallel. The algorithm PBFS is iterative. In the $i$-th iteration, the algorithm inserts, at the same time, all the vertices at distance $i$ from $v_0$ into the bag, where $v_0$ is the source vertex of the algorithm. After all the vertices at distance $i$ are inserted, the bag is recursively split into two new bags, until reaching bags with only a few elements. The maximum number of elements of the final bags are defined a priori. Then, each bag is processed in parallel. The neighbors of each vertex in a bag are examined to find unvisited vertices. Those unvisited vertices are added to a new bag, which will be the input of the next iteration. The PBFS algorithm has $O(n + m)$ work, $O(d \lg (n/d) + d \lg \Delta)$ span and $T_p = O((n + m)/p + d \lg^3 (n/d))$, where $d$ is the diameter of the input graph and $\Delta$ is the maximum out-degree of any vertex.

If we perform atomic operations to update the array of labels of the sequential algorithm to compute the canonical ordering and use that array in the PBFS algorithm to choose the vertices that will be inserted in the bag, we can compute the canonical ordering of a maximal plane graph with the same bounds of the PBFS algorithm.

The main problem with this idea is that the diameter of a maximal plane graph is $O(n)$, and therefore, the span will be $O(n)$. However, in practice, a machine with a limited amount of cores could exhibit a good speedup. The implementation and evaluation of this idea is left as future work.

### 6.2 Succinct representation of triangulated plane graphs via realizers

In this section we discuss how to compute, in parallel, a succinct representation of a triangulated plane graph $G$, based on a realizer $T_1, T_2, T_3$ of $G$. In Section 6.2.1 we discuss how to adapt the algorithm PGEA to compute the succinct representation $S'_{rz}$ of $G$. In Section 6.2.2 we discuss how to construct the succinct representation of the bit-vectors derived from $S'_{rz}$. Finally, in Section 6.2.3 we discuss the parallel computation of the realizers of $G$. 
### 6.2.1 Parallel computation of the multiple parentheses representation $S'_{rz}$

In this section we will show how to compute the string $S'_{rz}$, introduced in Section 3.2.3. The definition of the construction of $S'_{rz}$ (see Section 3.2.3) is given as follows:

- Classify all the edges of $G$ as part of $T_1$, $T_2$ or $T_3$. At the end of this stage, we will have the three spanning trees.

- Perform an Euler tour over $T_1$ to define a new order among the vertices of $G$.

- For each vertex $v_i$ of $G$, count its number of neighbors in $T_2$ that are lower, $l_{T_2}^i$, and higher, $h_{T_2}^i$, numbered. The same is done for the neighbors in $T_3$.

- Perform a new Euler tour over $T_1$. Each time when we visit a forward edge, write a “(” followed by $l_{T_2}^i$”s and $l_{T_3}^i$”s. Each time when we visit a backward edge, write $h_{T_3}^i$ “{”s followed by $h_{T_2}^i$ “[”s and a “)”. The resulting parentheses sequence is $S'_{rz}$.

**Input**: An adjacency list representation of a plane graph $G$ consisting of arrays $V_G$ and $E_G$.

**Output**: An adjacency list representation of the dual graph, $G_D$ of $G$ consisting of arrays $V_{G_D}$ and $E_{G_D}$.

```
1 $V_{G_D}$ = an array of vertices of length $2|V_G| - 4$ // $2|V_G| - 4$ is the number of faces of $G$
2 $E_{G_D}$ = an array of edges of length $|E_G|$ // Number of edges of G and $G_D$ is the same
3 $A$ = an array of length $|E_{G_D}|$
4 parfor $i = 0$ to $|V_{G_D}| - 1$ do
5 $V[i]$.first = $3 \times i$  // Each node of $G_D$ has 3 neighbors
6 $V[i]$.last = $3 \times i + 2$
7 parfor $i = 0$ to $|E_{G_D}| - 1$ do
8     if $i = ownership(E_G, i)$ then
9         $A[i] = 1$
10     parallel_prefix_sum($A$)
11 parfor $i = 0$ to $|E_{G_D}| - 1$ do
12     $c = E_G[i].cmp$
13     $idx = 3 \times (A[ownership(E_G, i)] + newIndex(E_G, i))$
14     $E_{G_D}[idx].src = A[ownership(E_G, i)]$
15     $E_{G_D}[idx].tgt = A[ownership(E_G, i)]$
16     $E_{G_D}[idx].cmp = 3 \times E_{G_D}[idx].tgt + newIndex(E_G, c)$
17 return $V_{G_D}, E_{G_D}$
```

**Algorithm 16**: Parallel dual graph algorithm (PDGA)
**Input**: An array of edges $E_G$ and the index of an edge in $E_G$, $i$.

**Output**: The index of the edge that is owner of the face where the edge $E_G[i]$ is involved.

```plaintext
1 owner = i
2 next = prev($E_G[owner].cmp$)
3 if $E_G[owner].src > E_G[next].src$ then
4   owner = next
5 next = prev($E_G[next].cmp$)
6 if $E_G[owner].src > E_G[next].src$ then
7   owner = next
8 return owner
```

**Function** ownership

This definition is similar to the alternative definition given for the parallel construction based on canonical ordering. Therefore, we can adapt the PGEA algorithm (Algorithm 15) to compute $S'_rz$ in parallel. Algorithm 17 does this. We call this algorithm *Parallel graph encoding algorithm - realizers version (PGEA-rz)*. The representation of the input graph $G$ is the same representation used in Section 6.1.1.

The algorithm creates four arrays, two auxiliar arrays $C_{T_2}$ and $C_{T_3}$ to store the lower-numbered and higher-numbered neighbors of each vertex of $G$, with respect to $T_2$ and $T_3$, an auxiliar array $LE$ to store the Euler Tour of $T_1$ and the array $S'_rz$ to store the parentheses representation induced by $T_1$, $T_2$ and $T_3$. The first step of the algorithm is to build the realizers $T_1$, $T_2$ and $T_3$, through the function buildRealizers. The explanation of this function will be done in Section 6.2.3. The second step is to compute the new order of the vertices of $G$ through the function newOrder. Remember that this new order corresponds to the counterclock-wise order of the nodes of $T_1$, which is a canonical ordering of $G$. The tree $T_1$ can be obtained using Algorithm 14 and the new order can be obtained using a variation of the PFEA algorithm (see Section 5.1). The variation of the PFEA algorithm can be seen in Algorithm 18. The idea behind Algorithm 18 is to increase rank just for forward edges, perform a parallel prefix sum algorithm and finally obtain the new order of each vertex in the first occurrence of this vertex in the implicit Euler tour. The next step of Algorithm PGEA-rz is to count the number of lower and higher adjacent vertices of $v$ in $T_2$ and $T_3$, considering the new ordering of the vertices of $G$. The values of lower-numbered and higher-numbered neighbors of $v$ are stored in $C_{T_2}[v].l$ and $C_{T_2}[v].h$ for $T_2$ and in $C_{T_3}[v].l$ and $C_{T_3}[v].h$ for $T_3$, respectively (lines 6 and 7). It can be done using a parallel prefix sum algorithm. The next step is to traverse $T_1$. Each entry in $LE$ represents the traversal of an edge of $T_1$ and stores three values: value is (“” followed by $C_{T_2}[v].l””) and $C_{T_3}[v].l””)”, or “”) preceded by $C_{T_3}[v].h””
Algorithm 17: Parallel graph encoding algorithm - realizers version (PGEA-rz)
Input: An adjacency list representation of a tree \( T \) consisting of arrays \( V \) and \( E \) and the number of threads, \( \text{threads} \).

Output: The tree \( T \) with a new ordering.

\[
\begin{align*}
\text{let } ET &= \text{ an array of length } |E| \\
\text{let } \text{chk} &= |E| / \text{threads} \\
\text{parfor } t = 0 \text{ to } \text{threads} - 1 \text{ do} \\
& \quad \text{for } i = 0 \text{ to } \text{chk} - 1 \text{ do} \\
& \quad \quad j = t \times \text{chk} + i \\
& \quad \quad \quad \text{if } \text{co}(E_T[j].\text{src}) < \text{co}(E_T[j].\text{tgt}) \text{ then} // \text{forward edge} \\
& \quad \quad \quad \quad ET[j].\text{rank} = 1 \\
& \quad \quad \quad \quad \text{if } E[j].\text{child} \text{ is a leaf then} \\
& \quad \quad \quad \quad \quad ET[j].\text{succ} = ET[j].\text{cmp} \\
& \quad \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad \quad \quad ET[j].\text{succ} = \text{first}(E[j].\text{tgt}) + 1 \\
& \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad ET[j].\text{rank} = 0 \\
& \quad \quad \quad \quad \text{if } E[j] \text{ is the last edge in the adjacency list of } E[j].\text{src} \text{ then} \\
& \quad \quad \quad \quad \quad ET[j].\text{succ} = \text{first}(E[j].\text{tgt}) \\
& \quad \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad \quad \quad ET[j].\text{succ} = \text{next}(E[j].\text{tgt}) \\
& \quad \quad \quad \text{parallel_list_ranking}(ET) \\
& \quad \text{parfor } i = 0 \text{ to } \text{threads} - 1 \text{ do} \\
& \quad \quad j = t \times \text{chk} + i \\
& \quad \quad \quad \text{if } \text{first}(E[j].\text{src}) = j \text{ then} // \text{First occurrence of the node } E[j].\text{src} \\
& \quad \quad \quad \quad V[E[j].\text{src}].\text{co} = ET[j].\text{rank}
\end{align*}
\]

Algorithm 18: Parallel algorithm \texttt{newOrder} to define a new order of the nodes in a maximal plane graph

of parentheses in \textit{value}, used to compute the rank of each symbol in \( S'_{rz} \).

The rest of the explanation of the \texttt{PGEA-rz} algorithm is the same as the \texttt{PGEA} algorithm.

The theoretical analysis of the \texttt{PGEA-rz} algorithm is given by the complexity of the functions \texttt{buildRealizers} and \texttt{newOrder}, and the complexity of the rest of the algorithm, which is, essentially, the complexity of the \texttt{PGEA} algorithm. Algorithm \texttt{newOrder} has the same complexity of the \texttt{PFEA} algorithm, \( O(n) \) work, \( O(\lg n) \) span and \( T_p = O(n/p + \lg p) \). The complexity of the function \texttt{buildRealizers} is also \( O(n) \) work, \( O(\lg n) \) span and \( T_p = O(n/p + \lg p) \). Thus, the complexity of the \texttt{PGEA-rz} algorithm is \( T_1 = O(n) \) work, \( T_\infty = O(\lg n) \) span and \( T_p = O(n/p + \lg p) \).

Similar to Algorithm \texttt{PGEA}, we can use Algorithm \texttt{PGEA-rz} to encode \( S'_{rz} \) using the parentheses sequences \( S'_1, S'_2, S'_3 \) and the bit-vectors \( B_1 \) and \( B_2 \) as follows:
\(S'_1\) can be obtained by writing just the parentheses “(” and “)”, without the other two kind of parentheses, and setting the corresponding rank to 1 (Lines 14-15 and 21-22 of the \texttt{PGEA-rz} algorithm).

\(S'_2\) can be obtained by writing just the \(C_{T_2}[E_{T_1}[j].tgt].l\) “)”s and the \(C_{T_2}[E_{T_1}[j].tgt].h\) “)”s parentheses, without the other two kinds of parentheses, and setting the corresponding rank to \(C_{T_2}[E_{T_1}[j].tgt].l\) or \(C_{T_2}[E_{T_1}[j].tgt].h\), as appropriate (Lines 14-15 and 21-22 of the \texttt{PGEA-rz} algorithm).

Similarly, \(S'_3\) can be obtained by writing just \(C_{T_3}[E_{T_1}[j].tgt].l\) “}”s and the \(C_{T_3}[E_{T_1}[j].tgt].h\) “}”s parentheses, without the other two kinds of parentheses, and setting the corresponding rank to \(C_{T_3}[E_{T_1}[j].tgt].l\) or \(C_{T_3}[E_{T_1}[j].tgt].h\), as appropriate (Lines 14-15 and 21-22 of the \texttt{PGEA-rz} algorithm).

\(B_1\) can be obtained by writing a “1” followed by \(C_{T_2}[E_{T_1}[j].tgt].l + C_{T_3}[E_{T_1}[j].tgt].l\) “0”s for forward edges or \(C_{T_2}[E_{T_1}[j].tgt].h + C_{T_3}[E_{T_1}[j].tgt].h\) “0”s followed by a “1” for backward edges (Lines 14 and 21).

Finally, \(B_2\) can be obtained by writing \(C_{T_2}[E_{T_1}[j].tgt].l\) “1”s followed by \(C_{T_3}[E_{T_1}[j].tgt].l\) “0”s, and setting the rank value to \(C_{T_2}[E_{T_1}[j].tgt].l + C_{T_3}[E_{T_1}[j].tgt].l\), for forward edges or by writing \(C_{T_3}[E_{T_1}[j].tgt].h\) “0”s followed by \(C_{T_2}[E_{T_1}[j].tgt].h\) “1”s, setting the rank value to \(C_{T_2}[E_{T_1}[j].tgt].h + C_{T_3}[E_{T_1}[j].tgt].h\), for backward edges (Lines 14 and 21).

Thus, the parentheses sequences \(S'_1, S'_2, S'_3\) and the bit-vectors \(B_1\) and \(B_2\) can be computed within the same bounds of \(S'_rz\). Observe that if we define an array \(LE\) for each parentheses sequence and bit-vectors, we can use the Algorithm \texttt{PGEA-rz} to compute \(S'_1, S'_2, S'_3, B_1\) and \(B_2\) at the same time, with just one parallel algorithm and the same bounds of constructing just one parenthesis sequence or bit-vector.

### 6.2.2 Parallel construction of the succinct representation of the multiple parenthesis sequence \(S'_rz\)

In Section 21 we showed that to support operations over maximal plane graphs, parenthesis sequences \(S'_1, S'_2\) and \(S'_3\) must support \texttt{rank}, \texttt{select}, \texttt{match}, \texttt{first} and \texttt{last} operations and bit-vectors \(B_1\) and \(B_2\) must support \texttt{rank} and \texttt{select} operations. All operations over parenthesis sequences and bit-vectors are supported by the solution in [102], which can be constructed in parallel with \(O(n + \lg p)\) work, \(T_p = O(n/p + \lg p)\) time, \(O(\lg n)\) span and \(O(n \lg n)\) working space, supporting operations in logarithmic time; or with \(O(n + \frac{n}{\lg n} \lg(n \frac{n}{\lg n}) + c)\) work, \(O(c + \lg(n \frac{n}{\lg n}))\) span and \(O(n \lg n)\) working space, supporting operations in \(O(c)\) time, where \(c > 3/2\) (see Chapter 5). Alternatively, bit-vectors \(B_1\) and \(B_2\) can be constructed with \(O(n)\) work, \(O(\lg n)\) span and \(O(n \lg n)\) working space, using the results of [116] (see Section 3.3 for more details).
Input: An adjacency list representation of the plane graph $G$ consisting of arrays $V_G$ and $E_G$, with a canonical ordering, and the number of threads, threads.

Output: Adjacency list representation of the realizers of $G$, $T_1$, $T_2$ and $T_3$. The representation of the tree $T_1$ consists of arrays $V_{T_1}$ and $E_{T_1}$. The representation of $T_2$ and $T_3$ is similar.

\begin{algorithm}
\begin{enumerate}
\item $V_{T_1} = V_G$
\item $V_{T_2} = V_G \setminus \{v_1\}$ \hfill // $v_1$ is the vertex with canonical ordering 1
\item $V_{T_3} = V_G \setminus \{v_1, v_2\}$ \hfill // $v_2$ is the vertex with canonical ordering 2
\item $L$ = an array of length $|V_G|$
\hfill // Stage 1
\begin{enumerate}
\item \textbf{parfor} $i = 0$ to $|V_G|$ \do
\item \textbf{parfor} $j = \text{first}(V_G[i])$ to $\text{last}(V_G[i]) - 1$ \do
\item $n1 = E_G[j].tgt$ \hfill // neighbor 1
\item $n2 = E_G[j+1].tgt$ \hfill // neighbor 2
\item \textbf{if} $\text{co}(V_G[i]) < \text{co}(V_G[n1])$ \textbf{AND} $\text{co}(V_G[i]) > \text{co}(V_G[n2])$ \then
\item $L[i].p = j + 1$
\item \textbf{else if} $\text{co}(V_G[i]) > \text{co}(V_G[n1])$ \textbf{AND} $\text{co}(V_G[i]) < \text{co}(V_G[n2])$ \then
\item $L[i].q = j$
\end{enumerate}
\hfill // Stage 2
\begin{enumerate}
\item \textbf{parfor} $i = 0$ to $|V_G|$ \do
\item addEdge($E_{T_1}$, $E_G[L[i].p].tgt$, $E_G[L[i].p].src$)
\item addEdge($E_{T_2}$, $E_G[L[i].q].tgt$, $E_G[L[i].q].src$)
\item \textbf{parfor} $j = L[i].p + 1$ to $L[i].q - 1$ \do
\item addEdge($E_{T_3}$, $E_G[j].src$, $E_G[j].tgt$)
\end{enumerate}
\end{enumerate}
\end{algorithm}

Algorithm 19: Parallel computation of realizers (buildRealizers)

6.2.3 Realizers in parallel

As was shown in Section 21, the realizers of a maximal plane graph $G$ can be computed given a canonical ordering of $G$. The sequential algorithm introduced in [19, 101] can be parallelized in two stages: First, finding, in parallel, the limits $p$ and $q$ of the neighbors of $v_k$ in $C_{k-1}$, with $p < q$, for each $v_k$, $3 \leq k \leq n$. Second, adding to $T_1$ the edge at position $p$, to $T_2$ the edge at position $q$ and all the edges at positions $p + 1, \ldots, q - 1$ to $T_3$. This idea is shown in Algorithm 19. In the algorithm, the array $L$ is used to store the limits of each vertex: $L.p$ stores the lower limit and $L.q$ stores the upper limit. In the algorithm, each edge of $G$ is visited independently once and only once. Therefore the complexity of both stages is $O(n)$ work, $O(1)$ span and $T_p = O(n/p)$ time. Finally, the setting of the limits of the adjacency lists of each node of $T_1$, $T_2$ and $T_3$ can be done with $O(n)$ work, $O(\log n)$ span and $T_p = O(n/p + \log p)$ time, using a parallel prefix sum algorithm over the lengths of the adjacency list of the tree spanning trees. Therefore, the complexity of the Algorithm 19 is $O(n)$ work, $O(\log n)$ span and $T_p = O(n/p + \log p)$ time.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes (n)</th>
<th>Edges (m)</th>
<th>Min fan-out</th>
<th>Max fan-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 worldcities</td>
<td>2,243,467</td>
<td>6,730,395</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>2 rand-1M</td>
<td>1,000,000</td>
<td>2,999,994</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>3 rand-2M</td>
<td>2,000,000</td>
<td>5,999,994</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4 rand-4M</td>
<td>4,000,000</td>
<td>11,999,994</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>5 rand-8M</td>
<td>8,000,000</td>
<td>23,999,994</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>6 rand-10M</td>
<td>10,000,000</td>
<td>29,999,994</td>
<td>3</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 6.1: Datasets used in the experiments of succinct maximal plane graphs.

6.3 Experiments

In this section, we study the scalability of our algorithms. As discussed, the PGEA-rz algorithm is similar to the PGEA algorithm. Therefore, we based all our experiments on the PGEA algorithm, extending our conclusions to both the PGEA and PGEA-rz algorithms. The experiments were carried out on machine B.

6.3.1 Experimental setup

We implemented the PGEA algorithm to compute the bit-vectors $S_1$, $S_2$ and $S_3$ and the PSTA algorithm of Chapter 5 to construct the succinct representations of the three bit-vectors. All the algorithm were implemented in C and compiled using GCC 4.9 with optimization level -O2 and using the -ffast-math flag. All parallel code was compiled using the GCC Cilk branch.

The experimental trials consisted in running the algorithm on artificial datasets of different number of nodes and cores. The datasets are shown in Table 6.1. Each dataset was generated in four stages: In the first stage, we used the function `rnorm` of R to generate random coordinates $(x, y)$\(^1\). The only exception was the dataset `worldcities`, which corresponds to the coordinates of 2,243,467 uniques cities in the world.\(^2\) In the second stage, we generated the Delaunay Triangulation of the coordinates generated in the first stage. The triangulations were generated using Triangle, a piece of software dedicated to the generation of meshes and triangulations\(^3\). In the third stage, we generated the maximal plane graph and the canonical ordering of the Delaunay triangulation computed in the second stage. Both the graph and

\(^1\)The `rnorm` function generates random numbers for the normal distribution given a mean and a standard deviation. In our case, the $x$ component was generated using mean 0 and standard deviation 45 and the $y$ component was generated using mean 0 and standard deviation 90. For more information about the `rnorm` function, please visit [https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html](https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html)

\(^2\)The dataset containing the coordinates was created by MaxMind, available from [https://www.maxmind.com/en/free-world-cities-database](https://www.maxmind.com/en/free-world-cities-database). The original dataset contains 3,173,959 cities, but some of them have the same coordinates. We selected the 2,243,467 cities with unique coordinates to build our dataset `worldcities`.

\(^3\)The software is available at [http://www.cs.cmu.edu/~quake/triangle.html](http://www.cs.cmu.edu/~quake/triangle.html). Our triangulations were generated using the options `-cezCBVPNE`.  

Figure 6.4: An example of a generated dataset to test the PGEA algorithm. Figure 6.4a shows the initial random coordinates. Figure 6.4b shows the Delaunay triangulation of the coordinates. Figure 6.4c shows the final maximal plane graph. The thick edge in Figure 6.4c represents the edge that was added to convert the graph in Figure 6.4b into a maximal plane graph.

The canonical ordering were computed using the Boost Library [29]. The graph was generated with the function `make_maximal_planar` and the canonical ordering was computed with the function `planar_canonical_ordering`\(^4\). Finally, in the fourth stage, we generated the canonical spanning tree of each maximal plane graph. See Figure 6.4 as an example of the stages. We repeated each trial five times and recorded the median time [122].

6.3.2 Running times and speedup.

Table 6.2 shows the sequential and parallel running times of the implemented algorithm. To compute the speedups, we used times obtained by `seq`. The best parallel times are identified using a bold typeface.

Figure 6.5 shows speedups for all the datasets in Table 6.1. Up to 16 threads, the speedup is almost linear, with an efficiency (speedup/p) of at least 56%, i.e., up to 16 threads, our algorithm reaches at least a 56% of the ideal speedup. With 16 or more threads, the speedup that our implementation exhibits is poor, reaching at most a 38% efficiency with `rand-10M` dataset and 64 threads, i.e., the obtained speedup is low with respect to the number of available threads. The reason of the low efficiency of our algorithm is its low workload. For each edge, the PGEA algorithm performs few comparisons and assignments. Therefore, the workload for each parallel task is not enough to take advantage of the 64 threads in machine B, even when we create \(\Theta(p)\) parallel tasks. To demonstrate that the low workload is the reason of the low efficiency,

\(^4\)For more details on the function `make_maximal_planar`, please visit [http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/make_maximal_planar.html](http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/make_maximal_planar.html). For more details of the function `planar_canonical_ordering`, please visit [http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_canonical_ordering.html](http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/planar_canonical_ordering.html)
we increased artificially the workload of our implementation of the PGEA algorithm. Between the lines 8-9 and 26-27 we added $x$ CAS operations, with $x \in \{16, 32, 128\}$. On each iteration of the loops of lines 8 and 26, each CAS operation was executed over $ET[i]$, increasing the workload per each edge. Observe that these extra operations do not change the complexity or the correctness of our algorithm. Figures 6.6, 6.7 and 6.8 show the speedups for all the datasets, with 16, 32 and 128 extra operations, respectively. As we increase the amount of artificial workload, efficiency is increased. For 16, 32 and 128 extra operations, we reached at least 50% of efficiency with up to 36, 40 and 56 threads. With rand-10M dataset and 64 threads, we reached at most 57%, 63% and 100% of efficiency with 16, 32 and 128 extra operations. Therefore, by increasing the workload, the speedup of our algorithm was close to the ideal speedup, i.e., the linear speedup. Since the real workload of our algorithm is less than the workload used in the experiments of Figures 6.6, 6.7 and 6.8, we cannot expect a good efficiency with an arbitrary number of threads. However, if we have to use few threads, we still can expect good efficiency (nearly 50% efficiency). The running times of the PGEA algorithm with extra operations are shown in the Appendix A.

The running time is directly dependent of the number of nodes, no matter if we consider extra operations or not. This can be seen in Figure 6.9.

<table>
<thead>
<tr>
<th>$p$</th>
<th>rand-1M</th>
<th>rand-2M</th>
<th>worldcities</th>
<th>rand-4M</th>
<th>rand-8M</th>
<th>rand-10M</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1.80</td>
<td>4.29</td>
<td>4.89</td>
<td>9.63</td>
<td>20.79</td>
<td>25.82</td>
</tr>
<tr>
<td>1</td>
<td>1.35</td>
<td>4.46</td>
<td>3.46</td>
<td>6.63</td>
<td>19.61</td>
<td>17.99</td>
</tr>
<tr>
<td>4</td>
<td>0.57</td>
<td>1.23</td>
<td>1.45</td>
<td>2.61</td>
<td>5.59</td>
<td>7.24</td>
</tr>
<tr>
<td>8</td>
<td>0.34</td>
<td>0.71</td>
<td>0.82</td>
<td>1.51</td>
<td>3.17</td>
<td>3.94</td>
</tr>
<tr>
<td>12</td>
<td>0.26</td>
<td>0.51</td>
<td>0.59</td>
<td>1.07</td>
<td>2.11</td>
<td>2.76</td>
</tr>
<tr>
<td>16</td>
<td>0.20</td>
<td>0.41</td>
<td>0.48</td>
<td>0.82</td>
<td>1.69</td>
<td>2.18</td>
</tr>
<tr>
<td>20</td>
<td>0.20</td>
<td>0.39</td>
<td>0.37</td>
<td>0.76</td>
<td>1.54</td>
<td>1.78</td>
</tr>
<tr>
<td>24</td>
<td>0.17</td>
<td>0.36</td>
<td>0.39</td>
<td>0.67</td>
<td>1.19</td>
<td>1.58</td>
</tr>
<tr>
<td>28</td>
<td>0.15</td>
<td>0.37</td>
<td>0.41</td>
<td>0.64</td>
<td>1.24</td>
<td>1.47</td>
</tr>
<tr>
<td>32</td>
<td>0.18</td>
<td>0.32</td>
<td>0.34</td>
<td>0.64</td>
<td>1.17</td>
<td>1.51</td>
</tr>
<tr>
<td>36</td>
<td>0.17</td>
<td>0.32</td>
<td>0.33</td>
<td>0.54</td>
<td>1.09</td>
<td>1.53</td>
</tr>
<tr>
<td>40</td>
<td>0.16</td>
<td>0.31</td>
<td>0.35</td>
<td>0.52</td>
<td>1.04</td>
<td>1.29</td>
</tr>
<tr>
<td>44</td>
<td>0.18</td>
<td>0.30</td>
<td>0.32</td>
<td>0.57</td>
<td>1.12</td>
<td>1.14</td>
</tr>
<tr>
<td>48</td>
<td>0.14</td>
<td>0.28</td>
<td>0.33</td>
<td>0.56</td>
<td>1.02</td>
<td>1.30</td>
</tr>
<tr>
<td>52</td>
<td>0.15</td>
<td>0.30</td>
<td>0.30</td>
<td>0.46</td>
<td>0.99</td>
<td>1.19</td>
</tr>
<tr>
<td>56</td>
<td>0.26</td>
<td>0.35</td>
<td>0.34</td>
<td>0.57</td>
<td>1.10</td>
<td>1.27</td>
</tr>
<tr>
<td>60</td>
<td>0.19</td>
<td>0.36</td>
<td>0.32</td>
<td>0.58</td>
<td>0.99</td>
<td>1.10</td>
</tr>
<tr>
<td>64</td>
<td>0.27</td>
<td>0.33</td>
<td>0.34</td>
<td>0.52</td>
<td>0.92</td>
<td>1.06</td>
</tr>
</tbody>
</table>

**Table 6.2:** Running times, in seconds, of the PGEA and PSTA algorithms, on aggregate, to construct the succinct representation of $S_1$, $S_2$ and $S_3$. seq represents the algorithms running sequentially. The best parallel times are shown using bold typeface.
Figure 6.5: Speedup of the PGEA and PSTA algorithms with all the datasets.

Figure 6.6: Speedup of the PGEA and PSTA algorithms with all the datasets, artificially increasing the workload with 16 CAS operations per edge.

Figure 6.7: Speedup of the PGEA and PSTA algorithms with all the datasets, artificially increasing the workload with 32 CAS operations per edge.

Figure 6.8: Speedup of the PGEA and PSTA algorithms with all the datasets, artificially increasing the workload with 128 CAS operations per edge.

Figure 6.9: Time over the number of vertices (n), with 64 threads. PGEA/PSTA corresponds to the execution of PGEA and PSTA without extra operations. For the other lines, the superscript indicates the number of extra CAS operations.

Figure 6.10: Memory consumption sorted by the number of vertices (n).
6.3.3 Memory consumption.

Figure 6.10 shows the memory consumption of our algorithm. We measured the memory allocated with malloc and released with free, reporting the peak of memory allocation and only considering memory allocated during construction, not memory allocated to store the input graph. The datasets are ordered incrementally by \( n \) and we only measured the memory usage for \( p = 1 \), since the extra memory needed for thread scheduling when \( p > 1 \) was negligible (See 2.2.1). The memory consumption is directly proportional to the number of vertices and it is composed by the allocation of the arrays \( C \), \( LE \) and \( S_\infty \). In particular, the array \( LE \) is the one with the biggest memory allocation, because it stores three number per edge.

6.3.4 Discussion.

The main drawback of our algorithm is its poor scalability. However, its scalability with few threads (\(< 16 \) threads) is good in practice (over 50% of efficiency). Since this scalability problem is due to the lack of workload of the algorithm, we cannot expect to get a better scalability by adding more threads. An interesting research problem is to find the number of threads needed to obtain at least 50% of efficiency. Finding the correct number of threads, we may use the rest of the threads in other procedures.

With respect to the memory consumption, our algorithm is efficient, since it uses an amount of memory proportional to the size of the input graph.

Another factor that may affect the performance of our algorithm is the topology of our machine. We will discuss this point in the Chapter 7.
Chapter 7

Discussions and Future Work

In this section we provide a more general discussion of the experiments, the model and the algorithms introduced in previous sections. We also present some open problems and potential new research avenues related to this thesis.

It is important to emphasize that this thesis involves what we think are important contributions to practical implementations of parallel algorithms in commodity architectures. In a way, then, topology has become important again, though thankfully not a show-stopper. For example, in the experiments of Sections 5.4 and 6.3, we observed that algorithms had a slowdown of the speedup at 16, 32, 48 and 64 threads. We hypothesize that the factor that generated the slowdown of the speedups of the algorithms had to do with the topology of the machine where we ran our experiments. The four processors on our machine were connected in a grid topology [33]. Each processor executes up to 16 threads. Up to 32 threads, all threads can be run on a single processor or on two adjacent processors in the grid, which keeps the cost of communication between threads low. Beyond 32 threads, at least three processors are needed and at least two of them are not adjacent in the grid. This increases the cost of communication between threads on these processors noticeably. Additionally, there exists other factors of the architecture that can impact the performance of multicore algorithms to construct succinct data structures, such as, cache inclusion policy which may vary for each new architecture, special wiring among cores and among caches, and cache coherency protocol. The impact of all of these factors in the implementation of multicore construction algorithms need to be studied in more detail.

Another factor that may impact the performance of the algorithms is the cache misses generated by the succinct representations. In order to obtain succinct representations, the data is reordered, which can impact negatively in the cache locality. For example, in the case of \textit{wtree}, each level corresponds to a reordering of the bit representation of the input sequence. The study of the trade off of succinct representations and structures with better cache locality is needed. Unfortunately, there are only a few models that take into account cache coherence, cache topology and more complicated memory models.

During the empirical evaluation of the \textbf{PFEA} and \textbf{PGEA} algorithms (see Algorithms 10 and 15, respectively), we observed a discrepancy between the practical results and the theoretical complexities. As was discussed in Sections 5.4.3 and 6.3.2, the discrepancy was due to the lack of workload for the parallel tasks. We think that it is interesting to find the maximum number of threads that achieves a good performance, where good performance means at least a 50% of efficiency. To find that
number of threads, we believe it is necessary to complement the theoretical analysis with a more empirical measure $W$: for example, total amount of CPU operations. With such measure, we can obtain an estimation of the amount of work per parallel task in practice. If we can ensure that the maximum amount of practical work per parallel task is bounded by $O(W_e/p)$, then we can expect better efficiency and use any remaining computational power for other processes in the OS. With some extra computational power to run other parallel processes, one interesting problem is the simultaneous execution of parallel algorithms designed under the DYM model.

Although PFEA and PGEA algorithms exhibit an almost linear speedup up to only 16 cores, there are current architectures where that amount of threads are enough. Currently, mobile devices, such as cellphones, tablets, among others, have multicore processors with at most 10 cores. If we can design and implement algorithms that scale up to 16 cores, then, we can use that algorithms in such mobile device. In particular, parallel algorithms that construct succinct data structures and scale up to 16 cores are suitable for mobile devices that have a limited memory capacity.

One open problem is the parallel computation of the canonical ordering of a triangulated plane graph. In Section 6.1.3, we present two approaches to try to solve it. In the first approach, we present the definition of a decomposition of the input graph. By computing that decomposition, we could apply a sequential algorithm in each non-overlapping subgraph to compute the canonical ordering of the complete graph. Currently, we do not have a parallel algorithm to compute the decomposition described in Section 6.1.3. Notice that decompositions based on the depth-first traversal of the graph should not be used, since they do not ensure the properties of the required decomposition. We think that the design of a parallel algorithm to compute such decomposition is of special interest, since it allows us to compute canonical orderings. With the canonical ordering, we can design parallel algorithms to compute succinct representation of triangulated plane graphs and to compute the straight-line embedding of plane graphs.

The second approach involves parallelizing the breadth-first traversal of the dual graph of the input graph. This approach takes advantages of the fact that, during the sequential computation of the canonical ordering, more than one vertex may be added to the canonical ordering (see Section 3.2.3). Therefore, we can add, in parallel, all the eligible vertices. The main problem with this approach is that its work and span are $O(n)$. However, we believe it is worthwhile to evaluate this idea in practice, since a parallel algorithm to compute the breadth-first traversal has shown good practical speedups [87].

There may be other succinct data structures that could benefit from a parallelization of their construction step. In the construction of succinct representation of trees, new parallel algorithms can be designed, based on a different parentheses representation. In Chapter 5, we construct a succinct representation based on a balanced parentheses representation of a tree. Instead, succinct representations based on depth-first unary degree sequence (DFUDS) representation [8, 78] or level-ordered unary degree sequence (LOUDS) representation [8, 75, 32] can be studied. Succinct
representations of two-dimensional point sets are another interesting domain where we can construct succinct data structures in parallel. In particular, two succinct data structures that have good practical behavior, both in time and space, are compressed quadtrees [48] and $K^2$-trees [15]. Another interesting succinct data structure to construct in parallel is the succinct representation of permutations based on the shortcut method of Munro et al. [95]. Using ideas of parallel list ranking algorithms [65], we can detect the cycles of a permutation, in parallel. With the cycles, applying the shortcut method in parallel is straightforward. With the parallel construction of succinct representation of permutations, we can construct data structures like that of Golynski et al. [53].

In this thesis, we focused on the parallel construction of static succinct data structures. The parallel construction of dynamic versions of those structures is still open. There are dynamic versions of succinct data structures with sequential construction algorithms for wavelet trees [90] and succinct ordinal trees [102] that can be studied. In particular, we will study how we can adapt our solutions for static succinct data structures to dynamic succinct data structures.

The DYM model is a good model to study the parallel construction of succinct data structures, since it is closely related to practical platforms, such as Cilk-Plus. However, there are other models to design parallel algorithms that do consider variables of the architecture that DYM does not and are interesting to study for the construction of succinct data structures. For example, the Multi-BPS model proposed by Valiant [123] considers the size of the cache memories, the number of cores, communication costs and synchronization costs between caches. The Message-passing parallel programming model [85] considers a distributed environment and network capabilities. The transactional memory model [67] allows us to define customized atomic operations that may involve reading or writing several words of memory. A few years ago, the transactional memory model was considered in the design of the multicore processor called Haswell [74], improving the speed of the customized atomic operations. The study of succinct data structures under these models is left as future work.
Chapter 8

Conclusions

Today, the amount of available data that need to be stored, read and processed is more than ever. It is imperative to find approaches that combine both the advances of modern architecture and software solutions to improve data manipulation, both in space, time, and in query complexity. In this thesis, we improve the construction time of succinct data structures by using multicore architectures. Thus, we can design succinct data structures with competitive querying time, efficient space usage and fast/scalable construction time.

We have introduced and implemented two parallel algorithms, pwt and dd, for the parallel construction of wavelet trees. The pwt algorithm constructs all the levels of the wavelet at the same time, reaching a work of \(O(n \lg \sigma)\) and a span of \(O(n)\), where \(n\) is the size of the input sequence and \(\sigma\) is the size of the alphabet. The dd algorithm constructs the wavelet tree in a domain-decomposition fashion, using the pwt in each segment. The dd algorithm reaches a work of \(O(n \lg \sigma)\) and a span of \(O(\lg n)\). For both algorithms we performed experiments with real-world and artificial datasets, reaching competitive speedups in a machine with 32 cores/64 running threads. We also faced the problem of answering queries in parallel. By grouping queries in batches, we obtained a parallel querying algorithm with \(O(q \lg \sigma)\) work and \(O(\lg \sigma)\) span, where \(q\) is the number of branch queries. In the experiments, our querying algorithm reaches a linear throughput, compared with an increasing number of threads.

For succinct ordinal trees, we presented a parallel algorithm to construct the structure of Navarro and Sadakane [102]. We presented a practical version with \(O(n)\) work, \(O(\lg n)\) span and \(O(\lg n)\) query time, where \(n\) is the number of nodes of the input tree. We also presented a second version which supports queries in \(O(c)\) time, with \(O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c)\) work and \(O(c + \lg(\frac{n}{\lg^c n}))\) span. The practical version was tested with several datasets, reaching competitive speedups in a multicore machine. As far as we know, our algorithm is the first one to construct the structure of Navarro and Sadakane in parallel.

For succinct triangulated plane graphs, we presented an algorithm to construct succinct representations based on canonical ordering. Given a triangulated plane graph with a canonical ordering, our algorithm construct its succinct representation with \(O(n)\) work and \(O(\lg n)\) span, supporting queries in \(O(\lg n)\) time, where \(n\) is the number of vertices of the input graph. Alternatively, we can construct a succinct representation that supports queries in \(O(c)\) time, with \(O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c)\) work and \(O(c + \lg(\frac{n}{\lg^c n}))\) span. We also explained how to construct succinct representation based on realizers, using a similar algorithm, with the same complexities. In our experiments, our algorithm to construct the succinct representation based on canonical
orderings reaches competitive speedups up to 16 threads. With more than 16 threads, our algorithm has a degradation of its speedup due to its lack of workload.

In this thesis we show how to construct some succinct data structures in parallel and obtain competitive speedups in multicore machines. With the introduction of such parallel algorithms we have made the succinct data structures more competitive in the multicore environment. We hope that this thesis can help to strengthen the research of practical succinct data structures in multicore systems.
Appendix A

Running times of the PGEA algorithm with extra operations

In this Appendix we show the running time of the PGEA algorithm when we increase the workload artificially. Table A.1 shows the running time by increasing the workload with 16 extra CAS operations per edge. Similarly, Tables A.2 and A.3 show the running times for 32 and 128 extra CAS operations per edge, respectively.

<table>
<thead>
<tr>
<th>$p$</th>
<th>rand-1M$^{+16}$</th>
<th>rand-2M$^{+16}$</th>
<th>worldcities$^{+16}$</th>
<th>rand-4M$^{+16}$</th>
<th>rand-8M$^{+16}$</th>
<th>rand-10M$^{+16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>3.85</td>
<td>8.74</td>
<td>9.77</td>
<td>18.49</td>
<td>38.82</td>
<td>47.19</td>
</tr>
<tr>
<td>1</td>
<td>3.14</td>
<td>6.70</td>
<td>7.56</td>
<td>17.19</td>
<td>29.03</td>
<td>47.31</td>
</tr>
<tr>
<td>4</td>
<td>1.08</td>
<td>2.22</td>
<td>2.28</td>
<td>4.96</td>
<td>9.39</td>
<td>12.07</td>
</tr>
<tr>
<td>8</td>
<td>0.56</td>
<td>1.23</td>
<td>1.36</td>
<td>2.54</td>
<td>5.30</td>
<td>6.59</td>
</tr>
<tr>
<td>12</td>
<td>0.41</td>
<td>0.83</td>
<td>0.96</td>
<td>1.77</td>
<td>3.55</td>
<td>4.54</td>
</tr>
<tr>
<td>16</td>
<td>0.33</td>
<td>0.66</td>
<td>0.74</td>
<td>1.37</td>
<td>2.74</td>
<td>3.44</td>
</tr>
<tr>
<td>20</td>
<td>0.28</td>
<td>0.55</td>
<td>0.62</td>
<td>1.13</td>
<td>2.30</td>
<td>2.84</td>
</tr>
<tr>
<td>24</td>
<td>0.25</td>
<td>0.51</td>
<td>0.59</td>
<td>0.94</td>
<td>1.89</td>
<td>2.41</td>
</tr>
<tr>
<td>28</td>
<td>0.27</td>
<td>0.46</td>
<td>0.52</td>
<td>0.87</td>
<td>1.74</td>
<td>2.16</td>
</tr>
<tr>
<td>32</td>
<td>0.23</td>
<td>0.44</td>
<td>0.50</td>
<td>0.84</td>
<td>1.57</td>
<td>1.99</td>
</tr>
<tr>
<td>36</td>
<td>0.20</td>
<td>0.41</td>
<td>0.45</td>
<td>0.77</td>
<td>1.60</td>
<td>1.94</td>
</tr>
<tr>
<td>40</td>
<td>0.22</td>
<td>0.40</td>
<td>0.42</td>
<td>0.70</td>
<td>1.42</td>
<td>1.75</td>
</tr>
<tr>
<td>44</td>
<td>0.24</td>
<td>0.38</td>
<td>0.41</td>
<td>0.67</td>
<td>1.42</td>
<td>1.61</td>
</tr>
<tr>
<td>48</td>
<td>0.22</td>
<td>0.35</td>
<td>0.42</td>
<td>0.67</td>
<td>1.30</td>
<td>1.63</td>
</tr>
<tr>
<td>52</td>
<td>0.26</td>
<td>0.34</td>
<td>0.43</td>
<td>0.60</td>
<td>1.21</td>
<td>1.52</td>
</tr>
<tr>
<td>56</td>
<td>0.28</td>
<td>0.34</td>
<td>0.42</td>
<td>0.60</td>
<td>1.24</td>
<td>1.49</td>
</tr>
<tr>
<td>60</td>
<td>0.30</td>
<td>0.38</td>
<td>0.40</td>
<td>0.56</td>
<td>1.17</td>
<td>1.40</td>
</tr>
<tr>
<td>64</td>
<td>0.37</td>
<td>0.38</td>
<td>0.42</td>
<td>0.59</td>
<td>1.04</td>
<td>1.29</td>
</tr>
</tbody>
</table>

**Table A.1:** Running times, in seconds, of the PGEA algorithm by artificially increasing the workload with 16 CAS operations per edge.
### Table A.2: Running times, in seconds, of the PGEA algorithm by artificially increasing the workload with 32 CAS operations per edge.

<table>
<thead>
<tr>
<th>p</th>
<th>rand-1M+32</th>
<th>rand-2M+32</th>
<th>worldcities+32</th>
<th>rand-4M+32</th>
<th>rand-8M+32</th>
<th>rand-10M+32</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>5.27</td>
<td>11.55</td>
<td>13.04</td>
<td>24.24</td>
<td>50.13</td>
<td>61.34</td>
</tr>
<tr>
<td>1</td>
<td>4.58</td>
<td>9.56</td>
<td>10.76</td>
<td>19.75</td>
<td>47.59</td>
<td>60.13</td>
</tr>
<tr>
<td>4</td>
<td>1.44</td>
<td>2.94</td>
<td>3.24</td>
<td>6.12</td>
<td>12.33</td>
<td>16.37</td>
</tr>
<tr>
<td>8</td>
<td>0.76</td>
<td>1.59</td>
<td>1.76</td>
<td>3.25</td>
<td>6.59</td>
<td>8.42</td>
</tr>
<tr>
<td>12</td>
<td>0.54</td>
<td>1.08</td>
<td>1.22</td>
<td>2.23</td>
<td>4.52</td>
<td>5.73</td>
</tr>
<tr>
<td>16</td>
<td>0.43</td>
<td>0.84</td>
<td>0.93</td>
<td>1.71</td>
<td>3.44</td>
<td>4.30</td>
</tr>
<tr>
<td>20</td>
<td>0.35</td>
<td>0.71</td>
<td>0.79</td>
<td>1.41</td>
<td>2.79</td>
<td>3.58</td>
</tr>
<tr>
<td>24</td>
<td>0.32</td>
<td>0.62</td>
<td>0.71</td>
<td>1.28</td>
<td>2.46</td>
<td>3.01</td>
</tr>
<tr>
<td>28</td>
<td>0.29</td>
<td>0.56</td>
<td>0.64</td>
<td>1.07</td>
<td>2.14</td>
<td>2.61</td>
</tr>
<tr>
<td>32</td>
<td>0.28</td>
<td>0.50</td>
<td>0.58</td>
<td>1.05</td>
<td>1.96</td>
<td>2.50</td>
</tr>
<tr>
<td>36</td>
<td>0.32</td>
<td>0.50</td>
<td>0.56</td>
<td>0.97</td>
<td>1.88</td>
<td>2.37</td>
</tr>
<tr>
<td>40</td>
<td>0.24</td>
<td>0.47</td>
<td>0.53</td>
<td>0.87</td>
<td>1.76</td>
<td>2.14</td>
</tr>
<tr>
<td>44</td>
<td>0.27</td>
<td>0.40</td>
<td>0.46</td>
<td>0.81</td>
<td>1.66</td>
<td>1.99</td>
</tr>
<tr>
<td>48</td>
<td>0.29</td>
<td>0.41</td>
<td>0.45</td>
<td>0.79</td>
<td>1.54</td>
<td>1.96</td>
</tr>
<tr>
<td>52</td>
<td>0.29</td>
<td>0.43</td>
<td>0.53</td>
<td>0.78</td>
<td>1.50</td>
<td>1.78</td>
</tr>
<tr>
<td>56</td>
<td>0.29</td>
<td>0.45</td>
<td>0.42</td>
<td>0.70</td>
<td>1.44</td>
<td>1.73</td>
</tr>
<tr>
<td>60</td>
<td>0.30</td>
<td>0.42</td>
<td>0.57</td>
<td>0.71</td>
<td>1.35</td>
<td>1.72</td>
</tr>
<tr>
<td>64</td>
<td>0.30</td>
<td>0.44</td>
<td>0.49</td>
<td>0.73</td>
<td>1.27</td>
<td>1.51</td>
</tr>
</tbody>
</table>

### Table A.3: Running times, in seconds, of the PGEA algorithm by artificially increasing the workload with 128 CAS operations per edge.

<table>
<thead>
<tr>
<th>p</th>
<th>rand-1M+128</th>
<th>rand-2M+128</th>
<th>worldcities+128</th>
<th>rand-4M+128</th>
<th>rand-8M+128</th>
<th>rand-10M+128</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>13.89</td>
<td>28.41</td>
<td>32.19</td>
<td>57.90</td>
<td>117.14</td>
<td>145.80</td>
</tr>
<tr>
<td>1</td>
<td>13.14</td>
<td>26.67</td>
<td>32.15</td>
<td>57.16</td>
<td>109.21</td>
<td>145.74</td>
</tr>
<tr>
<td>4</td>
<td>3.56</td>
<td>7.24</td>
<td>8.15</td>
<td>14.90</td>
<td>29.71</td>
<td>37.76</td>
</tr>
<tr>
<td>8</td>
<td>1.82</td>
<td>3.72</td>
<td>4.17</td>
<td>7.55</td>
<td>15.20</td>
<td>19.07</td>
</tr>
<tr>
<td>12</td>
<td>1.23</td>
<td>2.51</td>
<td>2.84</td>
<td>5.12</td>
<td>10.19</td>
<td>12.84</td>
</tr>
<tr>
<td>16</td>
<td>0.96</td>
<td>1.92</td>
<td>2.17</td>
<td>3.84</td>
<td>7.76</td>
<td>9.68</td>
</tr>
<tr>
<td>20</td>
<td>0.79</td>
<td>1.57</td>
<td>1.77</td>
<td>3.06</td>
<td>6.29</td>
<td>7.87</td>
</tr>
<tr>
<td>24</td>
<td>0.67</td>
<td>1.33</td>
<td>1.50</td>
<td>2.65</td>
<td>5.27</td>
<td>6.57</td>
</tr>
<tr>
<td>28</td>
<td>0.61</td>
<td>1.18</td>
<td>1.32</td>
<td>2.33</td>
<td>4.55</td>
<td>5.70</td>
</tr>
<tr>
<td>32</td>
<td>0.56</td>
<td>1.10</td>
<td>1.22</td>
<td>2.10</td>
<td>4.13</td>
<td>5.02</td>
</tr>
<tr>
<td>36</td>
<td>0.53</td>
<td>0.98</td>
<td>1.13</td>
<td>1.98</td>
<td>3.97</td>
<td>4.95</td>
</tr>
<tr>
<td>40</td>
<td>0.47</td>
<td>0.92</td>
<td>1.04</td>
<td>1.78</td>
<td>3.58</td>
<td>4.47</td>
</tr>
<tr>
<td>44</td>
<td>0.49</td>
<td>0.82</td>
<td>0.94</td>
<td>1.66</td>
<td>3.31</td>
<td>4.10</td>
</tr>
<tr>
<td>48</td>
<td>0.47</td>
<td>0.82</td>
<td>0.94</td>
<td>1.55</td>
<td>3.13</td>
<td>3.89</td>
</tr>
<tr>
<td>52</td>
<td>0.43</td>
<td>0.73</td>
<td>0.88</td>
<td>1.46</td>
<td>2.87</td>
<td>3.56</td>
</tr>
<tr>
<td>56</td>
<td>0.45</td>
<td>0.75</td>
<td>0.80</td>
<td>1.38</td>
<td>2.77</td>
<td>3.33</td>
</tr>
<tr>
<td>60</td>
<td>0.52</td>
<td>0.71</td>
<td>0.78</td>
<td>1.28</td>
<td>2.64</td>
<td>3.18</td>
</tr>
<tr>
<td>64</td>
<td>0.39</td>
<td>0.74</td>
<td>0.85</td>
<td>1.27</td>
<td>2.42</td>
<td>2.94</td>
</tr>
</tbody>
</table>
Appendix B

Topology of the machines used in the experiments

Topology of the machines used in the experiments. Figure B.1 shows the topology of the machine A and Figure B.2 shows the topology of the machine B. The topology was obtained using the command `lstopo` with options “–no-io –of pdf > output.pdf”

![Figure B.1: Topology of machine A.](image)
Figure B.2: Topology of machine B.
Bibliography


