Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Doctorado en Ciencias de la Computación

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 Ingenería 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traditional increase in the machines' processing power (led by clock speed up until the beginning of the millenium, and superseded by adding processors as of 2004), and the more modern, algorithm-based minimization of the space required to store data.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, σ 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

4.3.3 Querying Experiments . . . 58
4.4 Extensions . . . 61

Chapter 5 Parallel Construction of Succinct Trees 63
5.1 Parallel Folklore Encoding Algorithm . . . 63
5.2 Parallel Succinct Tree Algorithm . . . 64
5.2.1 Theoretical analysis. . . . 67
5.3 Parallel Algorithm to Support Constant-Time Queries . . . 68
5.4 Experimental Results . . . 76
5.4.1 Experimental setup . . . 76
5.4.2 Experimental Results of the PSTA algorithm . . . 77
5.4.3 Experimental Results of the PFEA algorithm . . . 80
5.4.4 Discussion . . . 82

Chapter 6 Parallel Construction of Succinct Triangulated Plane Graphs 84
6.1 Succinct representation of triangulated plane graphs via canonical ordering . . . 84
6.1.1 Parallel computation of the multiple parentheses representation S_{co} . . . 84
6.1.2 Parallel construction of the succinct representation of the multiple parentheses sequence S_{co} . . . 88
6.1.3 Two approaches to compute canonical orderings in parallel . . . 88
6.2 Succinct representation of triangulated plane graphs via realizers . . . 93
6.2.1 Parallel computation of the multiple parentheses representation S'_{rz} . . . 94
6.2.2 Parallel construction of the succinct representation of the multiple parenthesis sequence S'_{rz} . . . 98
6.2.3 Realizers in parallel . . . 99
6.3 Experiments . . . 100
6.3.1 Experimental setup . . . 100
6.3.2 Running times and speedup. . . . 101
6.3.3 Memory consumption. . . . 104
6.3.4 Discussion. . . . 104

Chapter 7 Discussions and Future Work 105

Chapter 8 Conclusions 108

Appendix A Running times of the PGEA algorithm with extra operations 110

Appendix B Topology of the machines used in the experiments 112

Bibliography 114

List of Tables

3.1 Operations supported by the NS-representation 19
3.2 Execution of the sequential canonical ordering algorithm 29

4.1 Datasets used in the experiments of wavelet trees 51
4.2 Running times of the sequential and parallel algorithms with 1 and 64 threads 52
4.3 Throughput, last level read misses and last level write misses of the `dd-IQA` and `parBQA` parallel algorithms 60

5.1 Datasets used in the experiments of succinct trees 77
5.2 Running times of the algorithms `libcds`, `sdsl`, and `PSTA` . . . 78
5.3 Running times of `PFEA` algorithm 81

6.1 Datasets used in the experiments of succinct maximal plane graphs 100
6.2 Running times of the `PGEA` and `PSTA` algorithms on aggregate . 102

A.1 Running times of the `PGEA` algorithm by artificially increasing the workload with 16 `CAS` operations per edge 110
A.2 Running times of the `PGEA` algorithm by artificially increasing the workload with 32 `CAS` operations per edge 111
A.3 Running times of the `PGEA` algorithm by artificially increasing the workload with 128 `CAS` operations per edge 111

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 $S =$"`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 `pwt` 43
4.2 Snapshot of an execution of the algorithm `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 *en.4.30* encoding each symbol with 4 bytes 55
4.7 Time over n with $\sigma = 2^{14}$, 64 threads and *en.14* datasets. . . . 55
4.8 Time over σ 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 `en.14.29` 59

5.1 Tree representation used as input to the `PFEA` algorithm 64
5.2 Example of the proof of Lemma 1 69
5.3 Computation of the ladders of a tree $T^{\mathcal{B}}$, with embedding $\mathcal{B}$. . 71
5.4 Speedup of `PSTA` algorithm compared to its sequential version 79
5.5 Speedup of `PSTA` algorithm compared to the `sdsl` algorithm . 79

5.6 Memory consumption of the algorithms psta, libcds and sdsl 80
5.7 Speedup of the PFEA algorithm with datasets ctree25, dna and prot . . . 82
5.8 Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 16 CAS operations per edge . . . 82
5.9 Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 32 CAS operations per edge. . . . 82

6.1 Example of graph decompositions that do not meet the properties 89
6.2 Decomposition of a triangulated plane graph . . . 90
6.3 Parallel computation of canonical orderings based on dual graphs and BFS traversal . . . 91
6.4 An example of a generated dataset to test the PGEA algorithm 101
6.5 Speedup of the PGEA and PSTA algorithms . . . 103
6.6 Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 16 CAS operations per edge . . . 103
6.7 Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 32 CAS operations per edge . . . 103
6.8 Speedup of the PGEA and PSTA algorithms, increasing artificially the workload with 128 CAS operations per edge . . . 103
6.9 Running times of the PGEA and PSTA algorithms over the number of vertices, with 64 threads . . . 103
6.10 Memory consumption sorted by the number of vertices . . . 103

B.1 Topology of machine A. . . . 112
B.2 Topology of machine B. . . . 113

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 ($\sim$ 0.0034% of the total) and 14 billions were indexed by Microsoft's Bing ($\sim$ 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 by 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:

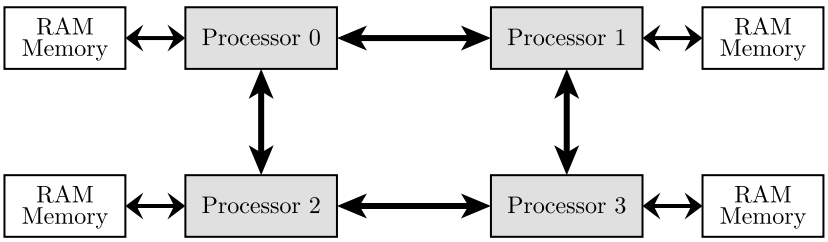


Figure 1.1: Example of a NUMA system and a grid topology with four processors.

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 **M***odified*-**E***xclusive*-**S***hared*-**I***nvalid (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.

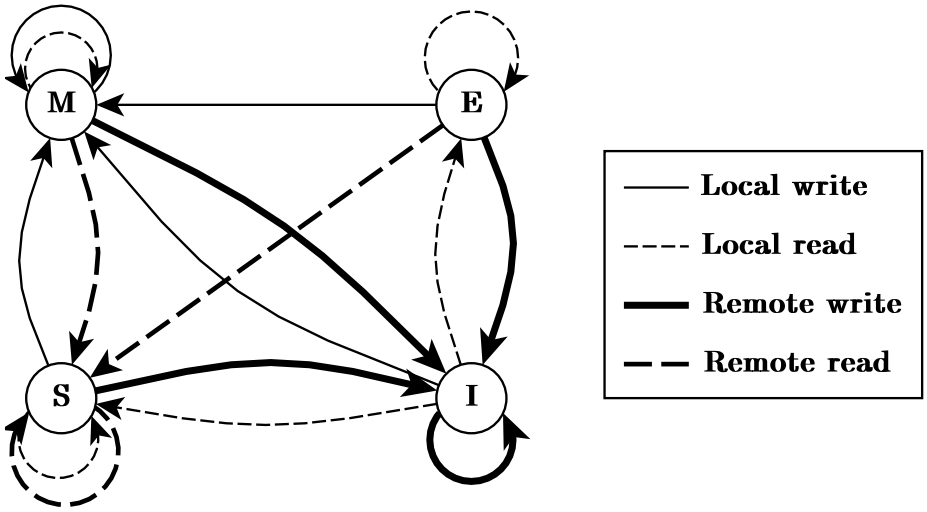


Figure 2.1: MESI protocol transitions.

Figure 2.1 shows the transitions among states. Initially, all cache lines are empty and the 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.

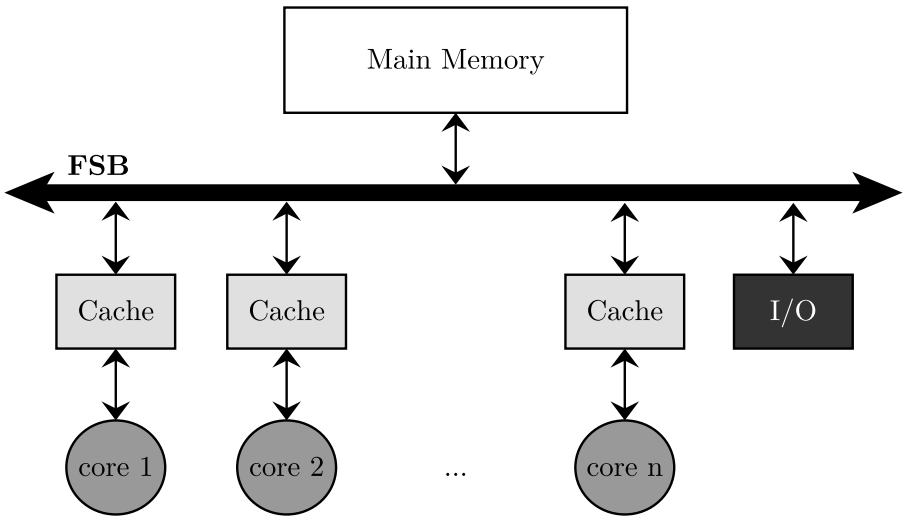


Figure 2.2: Symmetric Multiprocessor System (SMP).

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 $O(\log^4 n)$ 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[1] [86],

[1]Intel's CilkPlus: `www.cilkplus.org`

OpenMP[2], and Threading Building Blocks[3]. A comparison of some of those platforms can be found in [107].

DYM specifies the *logical parallelism* of an algorithm, using a reduced set of keywords: **spawn**, **sync**, and **parfor**. It is possible to obtain a serialization of a parallel algorithm in DYM by just deleting the keywords **spawn** and **sync**, and replacing the keyword **parfor** with the traditional keyword 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.

2.2.1 Dynamic Multithreading Model

As previously stated, DYM works by declaring the logical parallelism of an algorithm using the keywords **spawn**, **sync**, and **parfor**. In other words, **spawn**, **sync**, and **parfor** indicate which parts of the computation *may* proceed in parallel. The **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 **spawn**. In turn, the **sync** keyword signals that all spawned procedures must finish before proceeding with the next instruction in the stream. Finally, **parfor** is "syntactic sugar" for **spawn**ing one thread per iteration in the for loop, thereby allowing these iterations to run in parallel, followed by a **sync** operation that waits for all iterations to complete. In practice, the **parfor** keyword is implemented by halving the range of loop iterations, **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 **return** (which implicitly **sync**s) from a procedure, they form what is called a *strand*.

For all our parallel algorithms, their sequential versions can be obtained by replacing **parfor** instructions with sequential **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 dependes on two parameters of the computation: its *work* T_1 and its *span* T_∞. 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 *work*

[2]OpenMP: www.openmp.org
[3]Threading Building Blocks: www.threadingbuildingblocks.org

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

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.

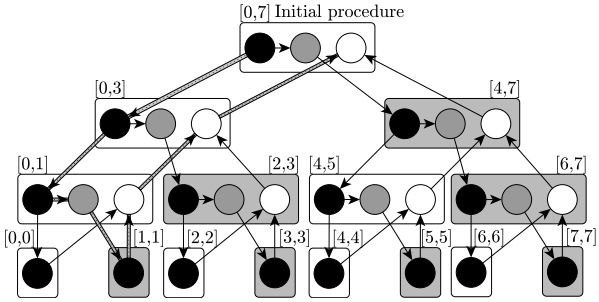


Figure 2.3: Example of a multithreaded computation on the Dynamic Multithreading Model. It corresponds to the directed acyclic graph representation of Algorithm 1. Vertices represent strands and edges represent dependences.

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_∞, 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_∞ 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 improvenment 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 **spawn**ed (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 **sync**s 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 = ⌊(s + e)/2⌋
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.

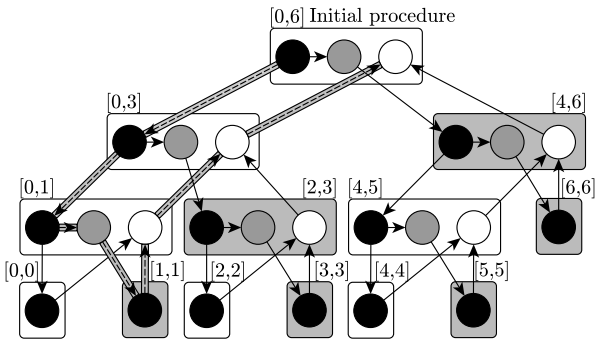


Figure 2.4: Example of a multithreaded computation on the Dynamic Multithreading Model. It corresponds to the directed acyclic graph representation of the call $pcount(A, v, 0, 6)$ of Algorithm 2. Vertices represent strands and edges represent dependences.

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_∞, and for any number p of cores, any greedy scheduler χ 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 χ. 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].

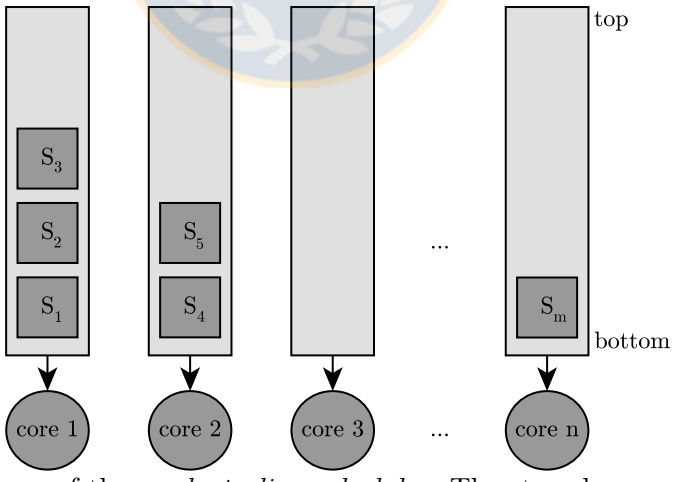


Figure 2.5: Diagram of the *work-stealing scheduler*. The strands are symbolized by light-blue squares, pools by gray rectangles, cores by green circumferences and threads by orange arrows. The thread associated with the core 3 must to steal a strand from another randomly selected thread, since this 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`[5]. 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.

[5]We used the C function `clock_getres()` 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 structute 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 $lwr = \lg C$, which is fewer that $2n$ 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, 55, 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].

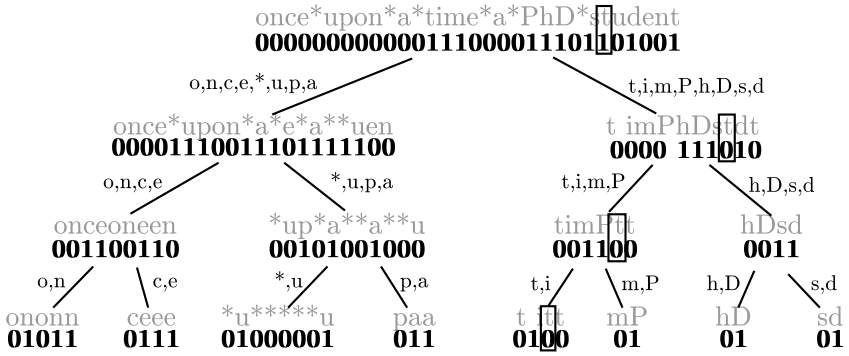
In this thesis we will work on three succinct data structures: *wavelet trees* (henceforth *wtree*), one of the most used succinct data structures in text indexing, *unlabeled succinct ordinal trees* (henceforth *stree*), which is the base of the succinct representation of other structures such as labeled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

3.2.1 Wavelet Tree

The *wtree* was introduced the first time in [57]. Although the *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 *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: $\texttt{access}(\texttt{S}, \texttt{i})$, which returns the symbol at position i in S; $\texttt{rank}_\texttt{c}(\texttt{S}, \texttt{i})$, which counts the times symbol c appears up to position i in S; and $\texttt{select}_\texttt{c}(\texttt{S}, \texttt{j})$, which returns the position in S of the j-th appearance of symbol c. *wtree*s 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 *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 Σ, 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 *wtree* where the symbols of Σ 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 *wtree* is a



(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 Σ in the edges are drawn for illustration purposes.

(b) Representation of a *wtree* using one pointer per level and its associtaed n-bit bitmap. It can simulate the nagivation on the tree by using the rank operation over the bitmaps.
Figure 3.1: A *wtree* for the sequence $S =$"`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.

balanced binary tree with $\lceil \lg \sigma \rceil$ levels.

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 $S =$ "*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 $\texttt{access}(\texttt{S}, 24)$. 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_{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 $rank_c(B_{curr}, i)$ be the number of c-bits up to position i in B_{curr}. At the beginning, we inspect the bit $B_{curr}[i]$. Since the bit is 1, we recompute $i = rank_1(B_{curr}, i) - 1 = 7$, change *curr* to be the right child of *curr* and halve $R = [8, 15]$. Then, we repeat the process. Since $B_{curr}[i] = 0$, $i = rank_0(B_{curr}, i) - 1 = 4$, *curr* is updated to be the left child of *curr* and $R = [8, 11]$. Now, $B_{curr}[i] = 0$, $i = rank_0(B_{curr}, i) - 1 = 2$, *curr* is changed to be the left child of *curr* and $R = [8, 9]$. Finally, in the last level, $B_{curr}[i] = 0$, so the range $R = [8, 8]$ and the answer for $\texttt{access}(\texttt{S}, 24)$ is $\Sigma[8] =$‘ t ’. $\texttt{rank}_\texttt{c}(\texttt{S}, \texttt{i})$ and $\texttt{select}_\texttt{c}(\texttt{S}, \texttt{i})$ perform similar traversals to $\texttt{access}(\texttt{S}, \texttt{i})$. 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 *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 *wtree*s. In [4], the authors explore the use of *wtree*s in distributed web search engines. They assume a distributed memory model and propose partition techniques to balance the workload of processing *wtree*s. 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 *wtree*s (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,

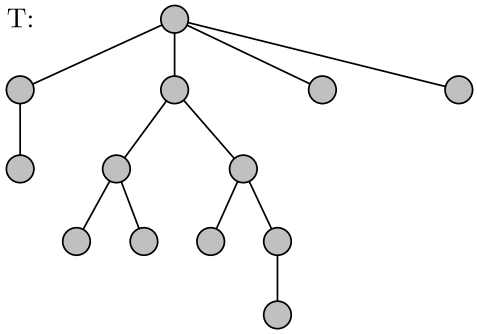


Figure 3.2: Balanced parentheses representation P of a tree T. This representation, also known as folklore encoding, can be stored using a bit vector, writing a 1 for each open parenthesis and a 0 for each closed parenthesis.

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 descendants. Thus, the length of P is $2n$. See Figure 3.2 as an example.

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

	Operation	Description
1	`child`(x, i)	Find the ith child of node x
2	`child_rank`(x)	Report the number of left siblings of node x
3	`degree`(x)	Report the degree of node x
4	`depth`(x)	Report the depth of node x
5	`level_anc`(x, i)	Find the ancestor of node x that is i levels above node x
6	`subtree_size`(x)	Report the number of nodes in the subtree rooted at node x
7	`height`(x)	Report the height of the subtree rooted at x
8	`deepest_node`(x)	Find the deepest node in the subtree rooted at node x
9	`LCA`(x, y)	Find the lowest common ancestor of nodes x and y
10	`lmost_leaf`(x) /`rmost_leaf`(x)	Find the leftmost/rightmost leaf of the subtree rooted at node x
11	`leaf_rank`(x)	Report the number of leaves before node x in preorder
12	`leaf_select`(i)	Find the ith leaf from left to right
13	`pre_rank`(x)/`post_select`(x)	Report the number of nodes preceding node x in preorder/postorder
14	`pre_select`/`post_select`(i)	Find the ith node in preorder/postorder
15	`level_lmost`(i)/`level_rmost`(i)	Find the leftmost/rightmost node among all nodes at depth i
16	`level_succ`(x)/`level_pred`(x)	Find the node immediately to the left/right of node x among all nodes at depth i
17	`access`(i)	Report $P[i]$
18	`find_open`(i)/`find_close`(i)	Find The matching parenthesis of $P[i]$
19	`enclose`(i)	Find the closest enclosing matching parenthesis pair for $P[i]$
20	`rank_open`(i)/`rank_close`(i)	Report the number of open/closed parentheses in $P[1..i]$
21	`select_open`(i)/`select_close`(i)	Find the ith open/closed parenthesis

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 $\{0, 1\}$:

$$
\begin{aligned}
\texttt{sum}(P, g, i, j) &= \textstyle\sum_{k=i}^{j} g(P[k]) \\
\texttt{fwd\_search}(P, g, i, d) &= \min\{j \mid j \geq i, \texttt{sum}(P, g, i, j) = d\} \\
\texttt{bwd\_search}(P, g, i, d) &= \max\{j \mid j \leq i, \texttt{sum}(P, g, j, i) = d\} \\
\texttt{rmq}(P, g, i, j) &= \min\{\texttt{sum}(P, g, i, k) \mid i \leq k \leq j\} \\
\texttt{RMQ}(P, g, i, j) &= \max\{\texttt{sum}(P, g, i, k) \mid i \leq k \leq j\} \\
\texttt{rmqi}(P, g, i, j) &= \operatorname*{argmin}_{k \in [i,j]}\{\texttt{sum}(P, g, i, k)\} \\
\texttt{RMQi}(P, g, i, j) &= \operatorname*{argmax}_{k \in [i,j]}\{\texttt{sum}(P, g, i, k)\}
\end{aligned}
$$

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

$$
\begin{array}{lll}
\pi : 1 \mapsto 1 & \phi : 1 \mapsto 1 & \psi : 1 \mapsto 0 \\
\quad 0 \mapsto -1 & \quad 0 \mapsto 0 & \quad 0 \mapsto 1
\end{array}
$$

For example, assuming the ith parenthesis in P is an open parenthesis, the matching closed parenthesis can be found using $\texttt{fwd\_search}(P, \pi, i, 0)$. 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 π). Navarro and Sadakane [102] discussed how to make it support these operations also for ϕ and ψ 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.

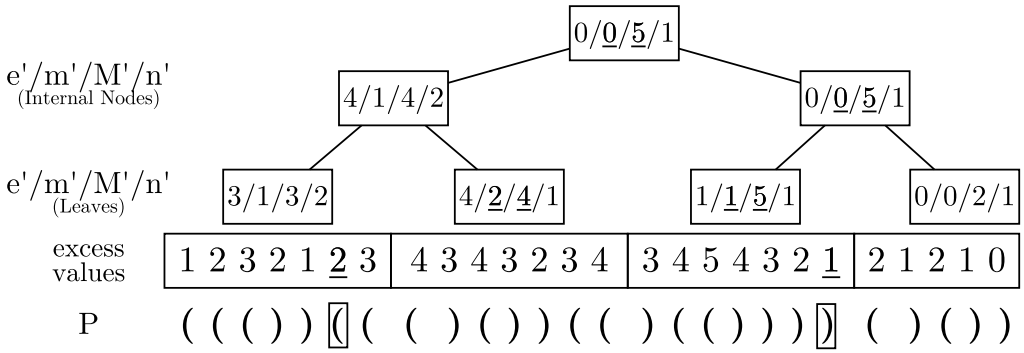


Figure 3.3: Range min-max tree of the balanced parentheses sequence of the Figure 3.2, with $s = 7$. In the figure, the m' and M' values involved in the operation $\texttt{fwd\_search}(P, \pi, 5, 1) = 20$ are underlined.

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 uth entry of each of these arrays stores some crucial information about P_u: Let the *excess* at position i of P be defined as $\texttt{sum}(P, \pi, 0, i) = \sum_{k=0}^{i} \pi(P[k])$. $e'[u]$ 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 π in $O(\lg n)$ time. Consider $\texttt{fwd\_search}(P, \pi, i, d)$ 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 $\texttt{fwd\_search}(P, \pi, i, d)$ 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 $2^{w/2}\texttt{poly}(w)$ 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 $\texttt{fwd\_search}(P, \pi, 5, 1)$. 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 $\lfloor 5/7 \rfloor = 0$. 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 ith leaf from the left, reduces to `rank` and `select` operations over a bit vector $P_1[1..2n]$ where $P_1[i] = 1$ iff $P[i] = 1$ and $P[i+1] = 0$. `rank` and `select` operations over P_1 in turn reduce to `sum` and `fwd_search` operations over P_1 and can thus be supported by an `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

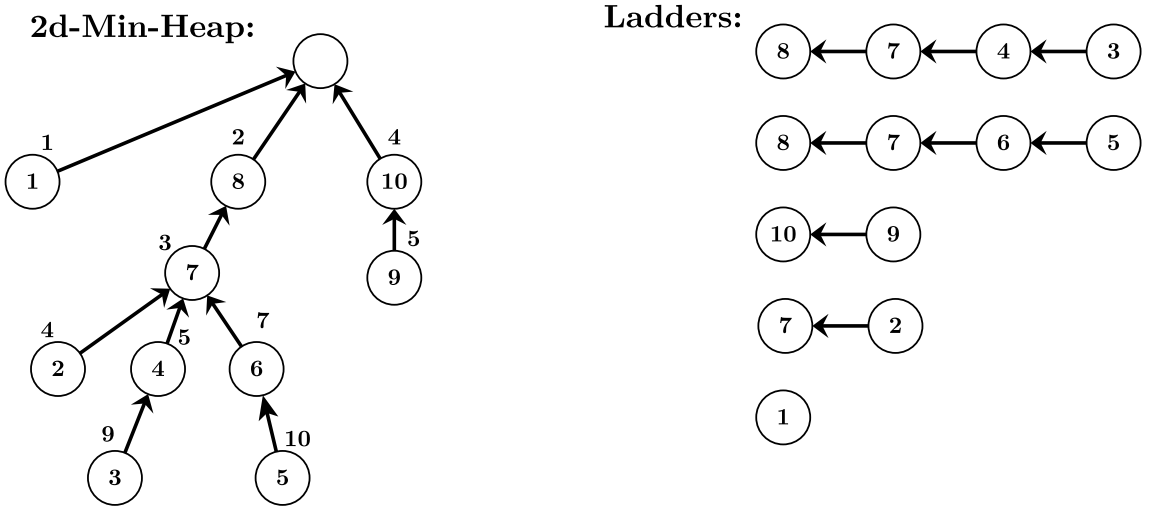
To support constant time queries on arbitrary-sized trees, the balanced parentheses representation P needs to be partitioned into blocks. We represent each block using a `RMMT` and then construct additional data structures considering the minimum, maximum and excess values of the `RMMT` of each block. The size of each block is w^c, so we have $\tau = \lceil 2n/w^c \rceil$ of such blocks. To support constant time queries inside each block, we construct a `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 τ `RMMT`s. Depending on the operations, the additional data structures differ.

To solve `fwd_search`(P, π, i, d), we first try to solve it inside block $j = \lfloor i/w^c \rfloor$. The answer is returned if it is found in that block. If it is not, we must find the first excess $d' = d + e_{j-1} +$ `sum`$(P, \pi, 0, i - 1 - w^c \cdot (j-1))$ in the `RMMT`s 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 `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 *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 $lrm(j) = (j_0, j_1, \ldots)$ where $j_0 = j$, $j_r < j_{r+1}$, $m_{j_{r+1}} < m_{j_r}$, and $m_{j_r+1}, \ldots, m_{j_{r+1}-1} \geq m_{j_r}$*

Once two *lrm* sequences coincide, they do so until the end. Thus, a 2d-Min-Heap is defined as a trie of τ nodes, composed of the reversed *lrm* sequences. Since



(a) *2d-Min-Heap* of the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4)$.

(b) The *ladders* generated by the 2d-Min-Heap of Figure 3.4a.

Figure 3.4: Example of the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4)$ and its ladders decomposition. In the 2d-Min-Heap, the indices of the sequence are inside and the values are outside of the nodes of the tree.

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 Pătraşcu [106]. Additionally, for each node v of the 2d-Min-Heap, the $\lg \tau$ ancestors at depths $depth(v) - 2^i$, $i \geq 0$ 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 $O(\lg_l z)$ 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 $2^{\lfloor \lg(depth(j)-d'') \rfloor}$. The answer is in the ladder of j'. The ancestor j' can be founded in constant the 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 rank/select queries over the bitmap of the ladder of node j', we find the node j_r.

To solve $\texttt{rmqi}(P, g, i, j)$ and $\texttt{RMQi}(P, g, i, j)$ operations on the τ 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 $\texttt{degree}(i)$ operations, we need to consider *pioneers*. Let *pioneers* be the tighest matching pair of parentheses (i, j), with $j = \texttt{find\_close}(i)$, 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 pionner (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 pionner (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 $\texttt{degree}(i)$ 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 $\texttt{child}(i, q)$ and $\texttt{child\_rank}(i)$ operations. Both for $\texttt{child}(i, q)$ and $\texttt{child\_rank}(i)$, 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 $\texttt{child}(i, q)$ query, we first check if $\texttt{child}(i, q)$ lies in the block of i or in $\texttt{find\_close}(i)$. If true, we solve it with an in-block query. If not, we compute $p = \texttt{rank}_1(C_i, \texttt{select}_0(C_i, q))$, 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 $q' = q - \texttt{rank}_0(C_i, \texttt{select}_1(C_i, p))$. $\texttt{child\_rank}(i)$ 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 $\texttt{rank}$ and $\texttt{select}$ on P, or the virtual bit vectors P_1 and P_2. For $\texttt{rank}$, it is necessary to store the answers at the end of each block,

finishing the query inside the corresponding block. For $\texttt{select}_1$ (and $\texttt{select}_0$), 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 P_1 is the same. Since we can assume that $w = \Omega(\lg n)$, 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 $\tau = \lceil 2n/w^c \rceil$ 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 $O(\frac{n \lg n}{w^c})$ bits, the arrays of ancestors use $O(\frac{n \lg^2 n}{w^c})$ bits, the sparse bitmap uses $O(\frac{n \lg w^c}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4})$ bits and the fusion trees use $O(\frac{n \lg^2 n}{w^c})$ bits. Thus, the extra structures to support fwd_search uses $O(\frac{n \lg^2 n}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4})$ bits, with $t > 0$. The rmqi and RMQi queries add $O(n/w^c)$ extra bits. Since there are $O(n/w^c)$ marked blocks, the degree operation uses $O(n \lg n/w^c)$ extra bits. The bitmaps of the remaining operations, such as child and child_rank, uses $\frac{2n}{w^c}\lg(w^c) + O(\frac{nt^t}{\lg^t n} + n^{3/4})$ extra bits, since they correspond to the sparse bitmap of [106]. Therefore, the total space used by the additional data structures is $O(\frac{n(c \lg w + \lg^2 n)}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4} + \sqrt{2^w})$ bits, where the term $\sqrt{2^w}$ corresponds to the lookup tables. With $w = \Omega(\lg n)$ and $t = c$, the extra space is $O(\frac{n(c \lg \lg n + \lg^2 n) + nc^c}{\lg^c n} + n^{3/4} + \sqrt{n})$ bits. Combined with the $2n + O(n/\lg n)$ bits of the RMMTs, the NS-representation requires $2n + O(n/\lg n + \frac{n(c \lg \lg n + \lg^2 n)}{\lg^c n})$ 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 $\varphi \in \Phi$. 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 $A = P$, $B = k = s = O(\frac{w}{c \lg w})$, φ-values encoding e', m', M', n' values and blocks of size $N = B^c$, it is possible to store each RMMT in $N + 2 + O(\sqrt{2^w})$ bits. The sum of all the RMMTs is $2n + O(\frac{n}{B^c} + \sqrt{2^w}) = 2n + O(\frac{n(c \lg \lg n)^c}{\lg^c n} + \sqrt{2^w})$. Finally, using aB-trees to store the RMMTs, the space usage of the NS-representation is reduced to $2n + O(\frac{n(c \lg \lg n + \lg^2 n)}{\lg^c n} + \sqrt{2^w})$ bits.

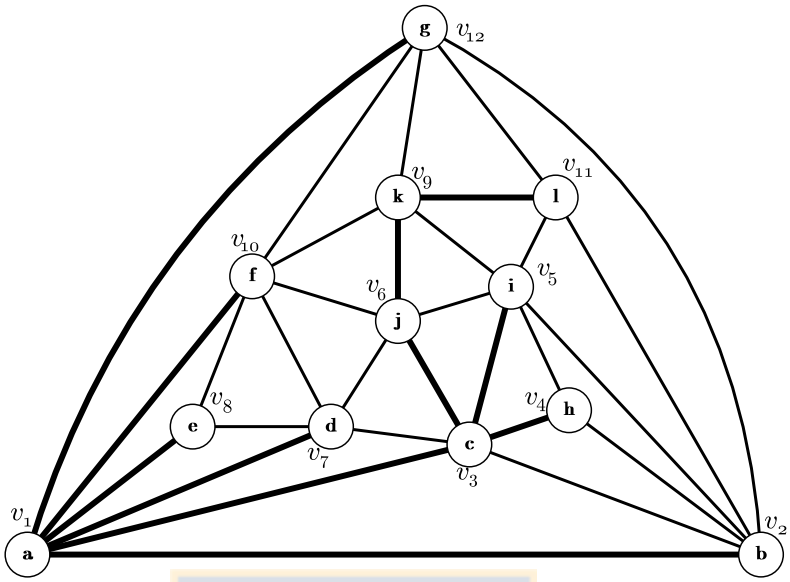


Figure 3.5: Triangulated plane graph with $n = 12$, $m = 30$ and $f = 20$. The canonical ordering of the graph is given beside each vertex. Thick edges indicate the canonical spanning tree T_{co}.

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 $|V| = n \geqslant 3$, $|E| = m$ 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 Π 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 neighbors 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.

```
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.

		a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**
0	Initial	①	1	−1	−1	−1	−1	v_{12}	−1	−1	−1	−1	−1
1	Processing v_1/a	v_1	①	0	0	0	0	v_{12}	−1	−1	−1	−1	−1
2	Processing v_2/b	v_1	v_2	①	0	0	0	v_{12}	0	0	−1	−1	0
3	Processing v_3/c	v_1	v_2	v_3	1	0	0	v_{12}	①	2	0	−1	0
4	Processing v_4/h	v_1	v_2	v_3	1	0	0	v_{12}	v_4	①	0	−1	0
5	Processing v_5/i	v_1	v_2	v_3	1	0	0	v_{12}	v_4	v_5	①	0	1
6	Processing v_6/j	v_1	v_2	v_3	①	0	2	v_{12}	v_4	v_5	v_6	1	1
7	Processing v_7/d	v_1	v_2	v_3	v_7	①	2	v_{12}	v_4	v_5	v_6	1	1
8	Processing v_8/e	v_1	v_2	v_3	v_7	v_8	1	v_{12}	v_4	v_5	v_6	①	1
9	Processing v_9/k	v_1	v_2	v_3	v_7	v_8	①	v_{12}	v_4	v_5	v_6	v_9	1
10	Processing v_{10}/f	v_1	v_2	v_3	v_7	v_8	v_{10}	v_{12}	v_4	v_5	v_6	v_9	①
11	Processing v_{11}/l	v_1	v_2	v_3	v_7	v_8	v_{10}	v_{12}	v_4	v_5	v_6	v_9	v_{11}

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 $O(\lg^4 n)$ 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 spannning 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 equivalente 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:

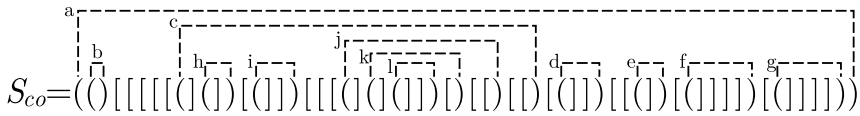
- If $S_{co}[i]$ is a parenthesis, with $1 \leq i \leq n_{()} + n_{[]}$, then $S_1[i] = 1$. Otherwise $S_1[i] = 0$.
- If the j-th parenthesis of S_{co} is open, with $1 \leq j \leq n_{()}$, then $S_2[j] = 1$. Otherwise $S_2[j] = 0$.

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 $S_2[\texttt{rank}_1(S_1, i)] = 1$, 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 $S_2[\texttt{select}_1(S_1, \texttt{rank}_1(S_1, i))] = 1$, 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.



S_1=11100000101010100100010101001010010010100100101010000101000011
S_2=11011010111000010101010 0
S_3=11111001001110000111111001101000010000

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:

- $\texttt{first}_1(S_{co}, i)$: The position of the closest 1 after $S_{co}[i]$. It can be computed in $O(1)$ with $\texttt{select}_1(S_1, \texttt{rank}_1(S_1, i) + 1)$.
- $\texttt{last}_1(S_{co}, i)$: The position of the closest 1 before $S_{co}[i]$. It can be computed in $O(1)$ with $\texttt{select}_1(S_1, \texttt{rank}_1(S_1, i) - 1)$.
- $\texttt{enclose}_1(S_3, i, j)$: 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).
- $\texttt{op}(S_{co}, i)$: The position in S_{co} of the open parenthesis associated to the vertex v_i.
- $\texttt{cp}(S_{co}, i)$: 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

T_{co}, then $\texttt{level\_anc}(v_j, 1) = v_i$, 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 $cp(S_{co}, i) < p < q < \texttt{first}_1(S_{co}, op(S_{co}, j))$, where $(p, q) = \texttt{enclose}(S_3, \texttt{rank}_0(S_1, \texttt{first}_1(S_{co}, cp(S_{co}, j))), op(S_{co}, j))$, 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 $(\texttt{first}_1(S_{co}, op(S_{co}, j)) - op(S_{co}, j)) + (\texttt{first}_1(S_{co}, cp(S_{co}, j)) - cp(S_{co}, j))$.

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, $3 \leqslant k \leqslant n - 1$, let C_{k-1} be $(w_0 = v_1, w_1, \ldots, w_l = v_2)$, 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.

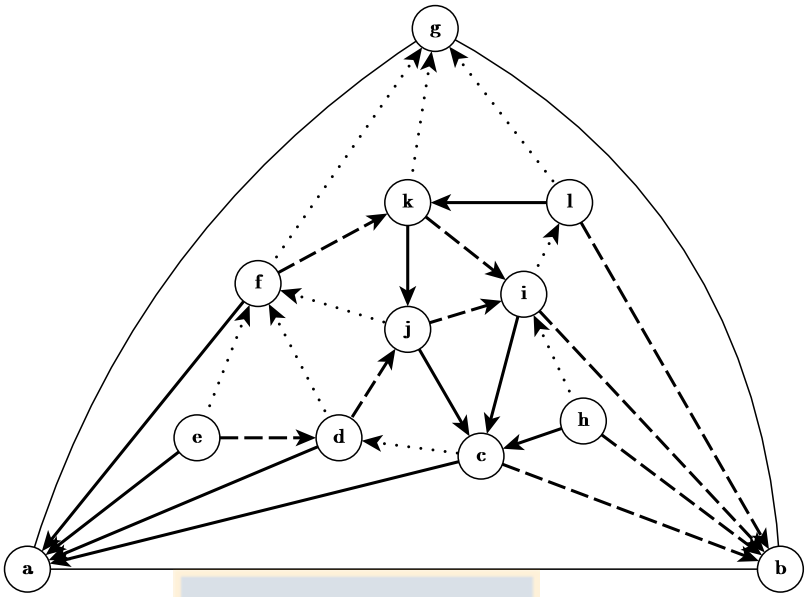


Figure 3.7: Realizer of the graph in Figure 3.5. T_1 is represented by thick edges, T_2 by dashed edges and T_3 by dotted edges. This realizer is also induced by the canonical ordering of Figure 3.5.

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.

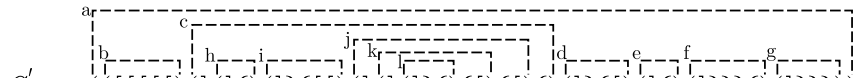
Barbay et al. [7] introduced a succinct representation of plane graphs based on realizers. With T_1, T_2 and T_3, the authors define three new orders of the vertices that are used later to encode the graph using a string of three kinds of parentheses. The first order is called **zeroth order** and correspond 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 ith 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 $(_i$ 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 egde (v_i', v_j') in T_2', where $i < j$, right before $)_i$.



$S'_{rz} = ((([[[[[]()[]{}()[]]{[[]()[]()[]]}{}{[]}{}()[]]{[]()[]{}()[]}}}{}()[]}}}))$

$S'_1 = (()(()()(((()))))()()()())$

$S'_2 = [[[[[]]]][[]]][[][]]]$

$S'_3 = \{\}\{\}\{\{\{\}\{\}\{\}\}\{\}\}\}$

$B_1 = 11000001101001100000110101000100100101100001100110000011000011$

$B_2 = 11111110100111110001010100110100001000$

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 egde (v'_i, v'_j) in T'_2, where $i < j$, right after $(_j$.
- A "}" for each egde (v'_i, v'_j) in T_3, where $i > j$, right after $(_i$.
- A "{" for each egde (v'_i, v'_j) in T_3, where $i > j$, right before $)_j$.

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 equivalente 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 $l_i^{T_3}$ "}"s. Each time that we visit a backward edge, write $h_i^{T_3}$ "{"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

$n_{\{\}}$ be the number of "()", "[]" and "{}", respectively. The succinct representation of S'_{rz} is constructed as follows:

- If $S'_{rz}[i]$ is a "(" or ")", with $1 \leq i \leq n_{()} + n_{[]} + n_{\{\}}$, then $B_1[i] = 1$. Otherwise $B_1[i] = 0$.
- If the parenthesis associated to the ith 0 in B_1 is either a "[" or "]", with $1 \leq i \leq n_{[]} + n_{\{\}}$, 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 $2m \lg 3 + o(m)$ 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 $2m \lg 6 + o(m)$ 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 `level_anc`$(v_j, 1)$ is equal to v_i, where `level_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, $\texttt{first}_0(S'_k, i)$ 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 $\texttt{select}_0(S'_k, \texttt{rank}_0(S'_k, i) + 1)$.
- `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 $(o_{first} - o_{left} - 1) + (c_{right} - c_{last} - 1)$, where o_{left} and c_{right} correspond to the open and closed parentheses of the vertex v_i in S'_{rz}, o_{first} is the position of the open parenthesis in S'_{rz} of the first child of v_i in T'_1 and c_{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, $\texttt{last}_0(S'_k, i)$ 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 $\texttt{select}_0(S^0_k, \texttt{rank}_0(S'_k, i) - 1)$.

- `select_neighbor_ccw`(v_i, v_j, r): The r-th neighbor of vertex v_i starting from vertex v_j in ccw order if v_i and v_j are adjacent, and ∞ otherwise. To support this operations, it is necessary to use the *local condition* of realizers. The local condition indicates that, given a vertex v_i, its neighbors listed in ccw order form the following six types of vertices: v_i's parent in $T_1^{'}$, v_i's children in T_3, v_i's parent in $T_2^{'}$, v_i's children in $T_1^{'}$, v_i's parent in T_3 and v_i's children in $T_2^{'}$. The constant time operations `child`(v_i, i) and `child_rank`(v_j) of [102] allow us to obtain the i-th child of v_i and the number of siblings before v_j in ccw order (where v_j is a neighbor of v_i) in $T_1^{'}$, respectively. Besides, the number of children of v_i in $T_2^{'}$ can be computed by counting the "["s right before the closed parenthesis of v_i and the number of children of v_i in T_3 can be computed by counting the "}"s right after the open parenthesis of v_i. Additionally, we can compute the number of each type of neighbors of v_i in constant time. Since all the previous operations are supported by rank/select operations over the parenthesis sequences $S_1^{'}$, $S_2^{'}$, $S_3^{'}$ and the bit-vectors B_1 and B_2, the operation `select_neighbor_ccw`(v_i, v_j, r) is supported in constant time.

- `rank_neighbor_ccw`(v_i, v_j, v_k): The number of neighbors of vertex v_i between (and including) the vertices v_j and v_k in ccw order if v_j and v_k are both neighbors of v_i, and ∞ otherwise. This operation is supported in the same way as the `select_neighbor_ccw`(v_i, v_j, 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: *wtree*s and rank/select structures. Based on work in [46], in [116], the author introduced two new algorithms to construct *wtree*s in parallel. The first algorithm, called `levelWT`, constructs the *wtree* level-by-level. In each level of the $\lceil \lg \sigma \rceil$ levels, the algorithm construct the nodes and their bitmaps in parallel with $O(n)$ work and $O(\lg n)$ span, which gives an algorithm of $O(n \lg \sigma)$ work and $O(\lg n \lg \sigma)$ 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 $O(W_{sort} \lg \sigma)$, 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 $O(n \lg \sigma)$ work

Input : Vertices v_i and v_j
Output: *True* if v_i and v_j are adjacent or *False* otherwise

if $\texttt{level\_anc}(v_j,\ 1) = v_i$ **then**
  return *True*

$a = \texttt{select}_1(S'_1, v_j)$
$b = \texttt{select}_1(B_1, a)$
$c = \texttt{rank}_1(B_2,$
$\texttt{rank}_0(B_1, b)\text{-}1)\text{-}1$
$d = \texttt{first}_0(S'_2, c)$
$e = \texttt{match}(S'_2, d)$
$f = \texttt{select}_1(B_2, e)$
$g = \texttt{rank}_1(B_1, f)\text{-}1$
$h = \texttt{first}_0(S'_1, g)$
$i = \texttt{match}(S'_1, h)$

if $i = v_i$ **then**
  return *True*

return *False*

Algorithm 4: *Adjacency* operation of the succinct representation of maximal plane graphs based on realizers.

Input : Vertex v_i
Output: The degree of the vertex v_i in the graph G

$d = \texttt{degree}(S'_1, v_i)$

$o_{left} = \texttt{select}_1(B_1, \texttt{select}_1(S'_1, v_i))$
$c_{right} = \texttt{select}_1(B_1,$
$\texttt{select}_0(S'_1, \texttt{match}(S'_1, v_i)))$
if v_i *is a leaf of* T'_1 **then**
  $o_{first} = 0$
  $c_{last} = 0$
else
  $o_{first} = \texttt{select}_1(B_1,$
  $\texttt{select}_1(S'_1, \texttt{first}_1(S'_1, v_i)))$
  $c_{last} = \texttt{select}_1(B_1,$
  $\texttt{select}_0(S'_1, \texttt{last}_0(S'_1, \texttt{match}(S'_1, v_i))))$

$d \mathrel{+}= o_{first} - o_{left} - 1$
$d \mathrel{+}= c_{right} - c_{last} - 1$

return d

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][2], 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).

[2]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 tesis 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 *wtree*s, where the symbols in Σ are contiguous in $[1, \sigma]$. Under these restrictions, the *wtree* is a balanced binary tree with $\lg \sigma$ levels. We will build the representation of *wtree*s that removes the $O(\sigma \lg n)$ bits of the topology. Hence, when we refer to a *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 wtree can be built independently from the others. Unlike in classical *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,

Proposition 1. *Given a symbol $s \in S$ and a level i, $0 \leq i < l = \lceil \lg \sigma \rceil$, of a 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 Σ 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:

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 Σ 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 Σ. We parallelized it by the technique of **spawn**ing 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 `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, σ. The output is a *wtree* WT that represents S. We denote the ith level of WT as $WT[i]$, $\forall i, 0 \leq i < \lceil \lg \sigma \rceil$.

Input : S, n, σ
Output: A wavelet tree representation WT of S

```
1 WT is a new wavelet tree with ⌈lg σ⌉ levels
2 B is an array of ⌈lg σ⌉ bitarrays of size n
3 createNode(S,n,B,0,0, ⌈lg σ⌉)
4 parfor i = 0 to ⌈lg σ⌉ − 1 do
5 |   WT[i] = createRankSelect(B[i])
6 return WT
```

Algorithm 6: Parallel recursive algorithm (`prwt`)

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

The Function 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 *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 *wtree*, *levels*. The initial call of the function creates the bitarray of the first level of the *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_{left} and S_{right}, from S. The size of S_{left} and S_{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_{left} and S_{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 Σ for the current node. Notice that each call of the function represents the computation of one (virtual) node of the *wtree*. If the symbol belongs to the first half of Σ assigned to the node, the bit at position $offset+i$ of $B[lvl]$ is set to 0 using *bitmapSetBit*, and the symbol $S[i]$ is copied into S_{left} at position $llen$ (lines 17 to 20). Similarly, if the symbol belongs to the second half, the bit at position $offset + i$ of $B[lvl]$ is set to 1 and the symbol $S[i]$ is copied into S_{right} at position $rlen$ (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_{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_{right} as input (line 23). All parallel recursive calls are **sync**ed at line 24.

Under the DYM model, the DAG of the `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^(⌈lg σ⌉−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^(⌈lg σ⌉−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, $T_\infty = O(n \lg \sigma)$. This is the case, for example, when Σ is ordered by frequency. In the best case, when all symbols in Σ 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 $O(\lg \sigma)$, which means that the algorithm scales on σ.

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 $O(n \lg \sigma)$ 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, σ. The output is a *wtree* WT that represents S. We denote the ith level of WT as $WT[i]$, $\forall i, 0 \leq i < \lceil \lg \sigma \rceil$.

Input : S, n, σ
Output: A wavelet tree representation WT of S

```
1  WT is a new wavelet tree with ⌈lg σ⌉ levels
2  parfor i = 0 to ⌈lg σ⌉ − 1 do
3  |  B is a bitarray of size n
4  |  C is an integer array of size 2^i
5  |  for j = 0 to n − 1 do
6  |  |  increment(C[S[j]/2^(⌈lg σ⌉−i)])
7  |  parPrefixSum(C)
8  |  for j = 0 to n − 1 do
9  |  |  if (S[j] & 2^(⌈lg σ⌉−i−1)) == 1 then
10 |  |  |  bitmapSetBit(B, C[S[j]/2^(⌈lg σ⌉−i)], 1)
11 |  |  else
12 |  |  |  bitmapSetBit(B, C[S[j]/2^(⌈lg σ⌉−i)], 0)
13 |  |  increment(C[S[j]/2^(⌈lg σ⌉−i)])
14 |  WT[i] = createRankSelect(B)
15 return WT
```

Algorithm 7: Per-level parallel algorithm (`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 ith 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 ith 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 Σ for that node. The corresponding bit is set using *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 ith level.

Figure 4.1 shows a snapshot of the execution of the `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 `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 `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

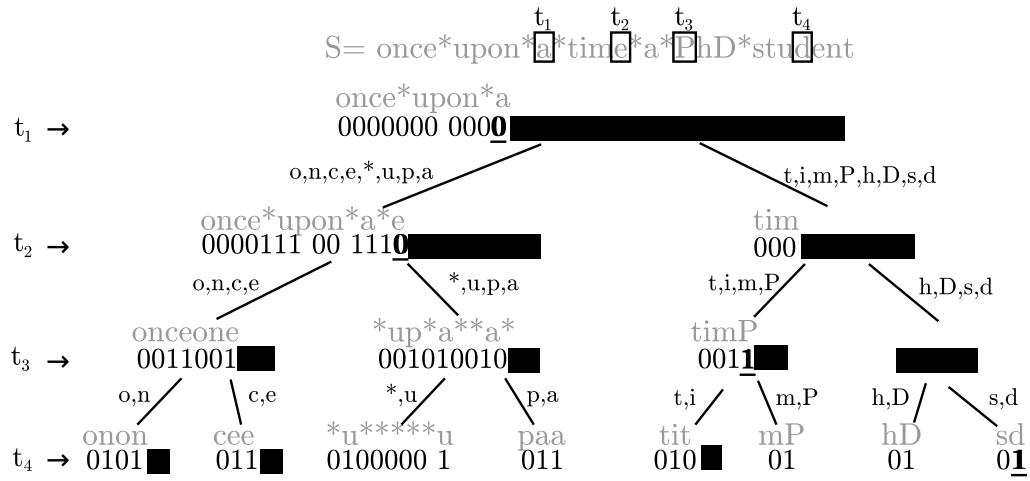


Figure 4.1: Snapshot of an execution of the algorithm `pwt` for the sequence introduced in Figure 3.1. In the snapshot, thread t_1 is writing the first bit of the symbol $S[10] =$‘ a ’ at level 0, thread t_2 is writing the second bit of $S[15] =$‘ e ’ at level 1, thread t_3 is writing the third bit of $S[19] =$‘ P ’ at level 2 and thread t_4 is writing the fourth bit of $S[26] =$‘ d ’ at level 3. Black areas represent bits associated to unprocessed symbols.

the *wtree*. That is, for $p = \infty$, $T_\infty = O(n+\lg\sigma) = O(n)$. In the same vein, parallelism will be $T_1/T_\infty = O(\lg\sigma)$. It follows that having $p \leq \lg\sigma$ the algorithm will obtain optimal speedup. The overhead added for the **parfor**, $O(\lg\lg\sigma)$ is negligible. With respect to the working space, the algorithm `pwt` needs the space of the *wtree* and the extra space for the array C, that is, a working space of $O(n\lg\sigma + \sigma\lg n)$ bits.

The main drawback of the `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 *wtree*s.

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 `pwt` algorithm on each segment, generating $O(\lg\sigma)$ tasks per segment and creating k partial *wtree*s. After that, the algorithm merges all the partial *wtree*s into a single one that represents the entire input text. We call this algorithm `dd` because of its domain decomposition nature.

The `dd` algorithm is shown in Algorithm 8. It takes the same input as `pwt` with the addition of the number of segments, k. The output is a *wtree* WT, which represents the input data S.

The first step of `dd` (lines 1 to 4) allocates memory for the output *wtree*, its

Input : S, n, σ, 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 $k \times \lceil \lg \sigma \rceil \times 2^{level}$
5 **parfor** $i = 0$ **to** $k - 1$ **do**
6 | $pB[i]$ = `createPartialBA(`S,σ,i,n/k`)`
7 **parfor** $i = 0$ **to** $\lceil \lg \sigma \rceil - 1$ **do**
8 | `parPrefixSum(`i,k`)`
9 B = `mergeBA(`n,σ,k,pB`)`
10 **parfor** $i = 0$ **to** $\lceil \lg \sigma \rceil - 1$ **do**
11 | $WT[i]$ = `createRankSelect(`$B[i]$`)`
12 **return** WT

Algorithm 8: Domain decomposition parallel algorithm (`dd`)

bitarrays, B, the bitarrays of the partial *wtree*s, 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 *wtree*s with respect to the partial *wtree* containing them. Similarly, G stores the offsets of all the nodes of the partial *wtree*s with respect to the final *wtree*. In other words, each entry $L[a][b][c]$ stores the position of node c at level b whose parent is partial *wtree* a. Each entry $G[a][b][c]$ stores the position of node c at level b in the partial *wtree* a inside the final *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 *wtree*s of the k segments in parallel. For each segment, createPartialBA is called to create the partial *wtree*. This function is similar to the one in the `pwt` algorithm, performing a prefix sum (line 5 in Function 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 *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 *wtree*s, in parallel. Function mergeBA creates one parallel task for each node in the partial *wtree*s. In each paralle task (lines 5 to 10) the function concatenates the bitarray of the node m/k of the ith level of the $m\%k$ partial *wtree*s into the corresponding bitarray, $B[i]$, of the final *wtree*. Using the local and the global offsets,

Input : S, σ, k', n
Output: A bitarray representation B of the k'th segment of S

```
1  B is an array of ⌈lg σ⌉ bitarrays of size n
2  parfor i = 0 to ⌈lg σ⌉ − 1 do
3  |  for j = n × k' to n × (k' + 1) − 1 do
4  |  |  increment(G[k'][i][S[j]/2^(⌈lg σ⌉−i)])
5  |  prefixSum(G,L)
6  |  for j = n × k' to n × (k' + 1) − 1 do
7  |  |  if (S[j] & 2^(⌈lg σ⌉−i−1)) == 1 then
8  |  |  |  bitmapSetBit(B, G[k'][i][S[j]/2^(⌈lg σ⌉−i)], 1)
9  |  |  else
10 |  |  |  bitmapSetBit(B, G[k'][i][S[j]/2^(⌈lg σ⌉−i)], 0)
11 |  |  increment(G[k'][i][S[j]/2^(⌈lg σ⌉−i)])
12 return B
```

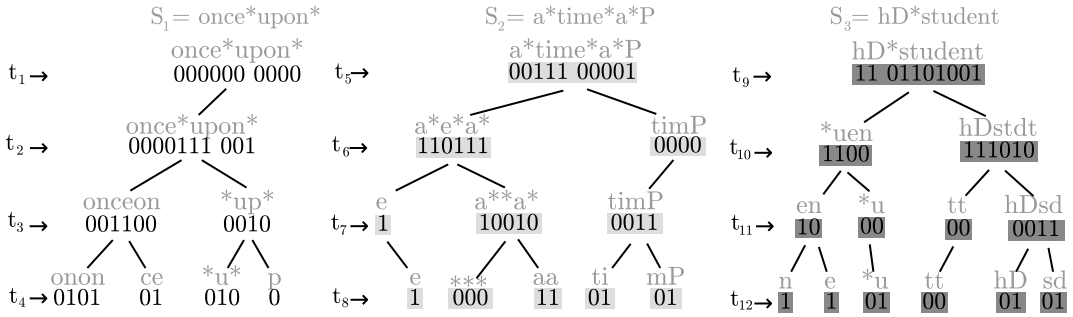
Function createPartialBA

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*.

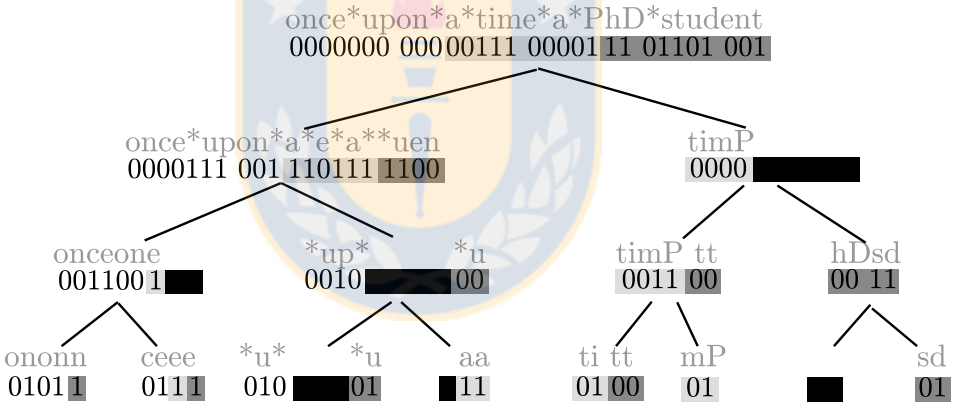
For an example of the algorithm, see Figure 4.2. Figure 4.2a shows a snapshot of the function createPartialBA and Figure 4.2b shows a snapshot of mergeBA.

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 *wtree*s 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**s is $O(\lg p + \lg \sigma \lg \lg \sigma)$. For $p = \infty$, the span of the construction of the partial *wtree*s 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 *wtree*s, 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 *wtree*s 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 *wtree*s 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 *wtree*s, respectively. The positions of the partial *wtree*s 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, σ, k, pB
Output: A bitarray representation B of the input sequence S

```
1  B is an array of ⌈lg σ⌉ bitarrays of size n
2  parfor i = 0 to ⌈lg σ⌉ − 1 do
3      parfor j = 0 to k − 1 do
4          parfor m = j × 2^i to (j + 1) × 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 *wtrees*: *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 *wtree*, 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, σ,*queries*, *num_queries*, *batch_size*
Output: *results*, an array containing the results for each query

1 $num\_batches = num\_queries/batch\_size$
2 *results* is an array of size *num_queries*. It will store the result of each query
3 *nd* is a structure that store information about the node that is being processed.
4 **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 | $nd.S_l = 0$, $nd.S_r = n - 1$ `// The root of` WT
7 | $nd.\sigma_l = 0$, $nd.\sigma_r = \sigma - 1$, $nd.lvl = 0$
8 | `batchRangeCount(`WT,σ,n,*queries*,*nd*,*results*,*states*,$batch\_size * i$,*batch_size*`)`
9 **return** *results*

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

`parBQA`). The input of the algorithm corresponds to a *wtree*, the size of the sequence S represented by the *wtree*, n, the number of symbols of such sequence, σ, an array with the branch queries, *queries*, the number of queries, *num_queries*, and the number of queries on each batch, *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 Σ 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 *wtree* (lines 6 and 7) and then calls the recursive Function batchRangeCount (line 8). Each recursive call of the function corresponds to the processing of a virtual node of the *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 batchRangeCount). Since we implement the *wtree* using one bitarray per level, we need to compute an *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 *states* which stores the state of each query. If a query q was already finished in a previous call, then $stated[q]$ 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 *results*.

Observe that the **parfor** in line 4 of the Algorithm 9 exploits the spatial and temporal locality of the node's bitarrays and the **spawn** and **sync** in lines 31 and 33 of the Function 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 *range count* and *range report*. Finally, note that the algorithm can be modified to apply batch processing to path queries.

Input : WT, σ, 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].σ_l > queries[q].σ_r then
11         local_states[q] = 1, increment(finished)
12         continue
13     if queries[q].σ_r < nd.σ_l ∨ queries[q].σ_l > nd.σ_r then
14         local_states[q] = 1, increment(finished)
15         continue
       // The current node contains completely the range of interest
       // The whole range [nd.lim_l, nd.lim_r] is part of the answer
16     if queries[q].σ_l ≤ nd.σ_l ∧ queries[q].σ_r ≥ nd.σ_r then
17         local_states[q] = 1
18         increment(finished)
19         results[q]+= queries[q].S_r − queries[q].S_l + 1
20         continue
21     queries_l[q].S_l = rank_0(WT[nd.lvl], queries[q].S_l − 1) −offset + nd.lim_l
22     queries_l[q].S_r = rank_0(WT[nd.lvl], queries[q].S_r) −offset + nd.S_l − 1
23     queries_l[q].σ_l = queries[q].σ_l, queries_l[q].σ_r = queries[q].σ_r
24     queries[q].S_l = queries[q].S_l − queries_l[q].S_l + zeros + nd.S_l
       queries[q].S_r = queries[q].S_r − queries_l[q].S_r − 1 + zeros + nd.S_l
25 if finished == num_queries then
26     return
27 nd_l is a copy of nd
28 h_σ = (nd.σ_l + nd.σ_l)/2
29 nd_l.S_r = nd.S_l + zeros − 1, nd_l.σ_r = halfσ, nd_l.lvl ++
30 nd.S_l = nd.S_l + zeros, nd.σ_l = halfσ + 1, nd.lvl ++
31 spawn batchRangeCount(WT,σ,n,queries_l,nd_l,results,local_states,init,batch_size)
32 batchRangeCount(WT,σ,n,queries,nd,results,local_states,init,batch_size)
33 sync
34 release local_states
35 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 Σ of σ=633,816 symbols. For experimentation, we generated an alphabet Σ' of size 2^x, taking the top 2^x *most frequent* words in the original Σ, 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 Σ' in the original English text and, when found, appended it to S until it reached the maximum possible size given σ' (∼1.5GB, 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)

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

Table 4.1: Datasets used in the experiments of *wtree*s.

rank/select structures. We compared our algorithms against LIBCDS[5] and 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 LIBCDS does). For SDSL we use the `bit_vector` implementation with settings `rank_support_scan<1>`, `select_support_scan<1>` and `select_support_scan<0>` to skip construction time of rank/select structures. In our experiments, `shun` is the fastest of the three algorithms introduced in [116], compiled also with the -O2 optimization flag. Our `dd` algorithm was tested with $k = p$ privileging time performance over memory.

Running times and speedup. Table 4.2 shows the running times of all tested algorithms. LIBCDS and `shun` work just for $n < 2^{32}$, so we cannot report running times of these algorithms for the datasets `rna.4GB`, `rna.5GB`, `rna.6GB` and `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 LIBCDS and SDSL are the state-of-the-art in sequential implementations of *wtree*s, the best sequential running times were obtained from the parallel implementations running on one thread. The main reason for this is that SDSL implements a semi-external algorithm for *wtree* construction, involving

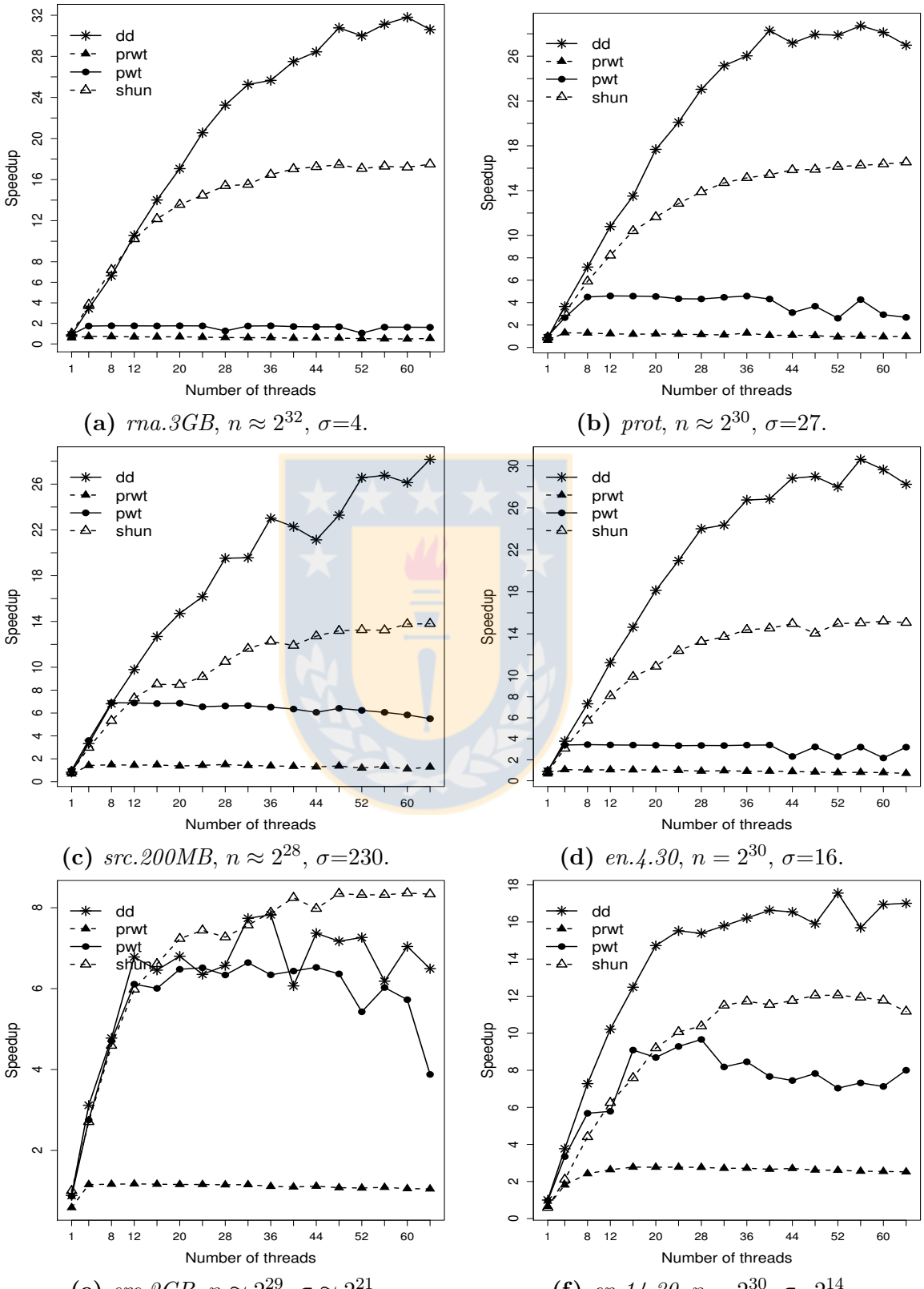
[5]We also tested a new version of LIBCDS called LIBCDS2. However the former had better running times for the construction of *wtree*s.

Datasets	libcds	sdsl	prwt		pwt		dd		shun	
			1	64	1	64	1	64	1	64
rna.512MB	23.42	32.41	18.96	22.52	11.83	7.00	12.65	**0.40**	12.63	0.67
rna.1GB	47.38	65.30	38.18	44.49	23.89	16.19	25.30	**0.62**	25.36	1.32
rna.2GB	100.13	131.86	75.72	95.00	46.98	27.62	50.80	**1.20**	50.89	2.64
rna.3GB	142.90	220.11	111.77	124.73	71.09	41.00	75.37	**2.17**	66.35	3.79
rna.4GB	-	198.10	153.36	171.26	94.39	55.04	101.44	**2.84**	-	-
rna.5GB	-	329.27	192.50	213.79	117.13	68.24	126.66	**3.57**	-	-
rna.6GB	-	389.25	229.55	249.11	141.59	81.80	152.57	**4.35**	-	-
rna.13GB	-	881.41	511.89	500.94	314.86	330.44	333.14	**10.75**	-	-
prot	104.40	142.67	91.53	59.70	58.54	21.81	68.19	**2.17**	64.06	3.54
src.200MB	24.81	31.41	21.65	11.43	14.68	2.67	17.70	**0.52**	16.73	1.06
src.98MB	7.92	9.52	8.83	4.59	5.28	0.77	5.73	3.94	5.07	**0.75**
src.512MB	37.77	49.21	41.63	23.21	28.94	5.07	28.98	5.36	25.52	**3.07**
src.1GB	75.48	99.95	83.70	46.97	57.99	8.87	55.36	9.60	49.52	**6.17**
src.2GB	150.67	205.41	167.33	93.59	112.78	25.30	110.83	15.11	98.11	**11.77**
en.4.27	8.78	14.24	8.92	6.70	5.75	1.82	6.50	**0.28**	6.98	0.38
en.4.28	15.82	28.53	17.61	12.96	11.44	3.67	12.88	**0.40**	12.34	0.77
en.4.29	35.43	57.11	35.46	30.96	23.01	7.22	25.51	**0.84**	24.68	1.57
en.4.30	70.00	113.88	70.84	65.30	46.10	14.40	51.06	**1.63**	55.56	3.06
en.6.27	12.44	19.10	12.60	7.35	7.98	1.78	9.58	**0.36**	10.46	0.61
en.6.28	22.65	38.37	25.24	14.30	15.92	3.33	19.35	**0.52**	18.38	1.17
en.6.29	50.28	76.91	51.55	25.71	31.78	7.08	37.90	**1.18**	41.86	2.36
en.6.30	99.66	153.72	103.11	65.59	63.62	15.90	76.59	**2.20**	83.29	4.68
en.8.27	15.87	26.00	16.67	7.03	11.48	1.87	13.15	**0.46**	14.10	0.88
en.8.28	29.06	52.15	33.43	15.67	22.86	3.71	26.52	**0.78**	28.28	1.58
en.8.29	64.84	105.01	67.73	31.83	45.79	7.57	52.53	**1.56**	56.68	3.14
en.8.30	128.65	209.54	136.23	67.69	91.83	14.65	105.00	**3.13**	113.13	6.26
en.10.27	21.32	33.25	24.13	9.58	14.61	2.26	13.94	1.66	17.26	**1.39**
en.10.28	43.55	68.00	49.90	19.24	30.32	6.43	29.05	**2.18**	33.15	2.78
en.10.29	89.96	136.67	101.17	38.82	60.69	9.25	58.55	**4.59**	67.16	5.67
en.10.30	183.57	281.53	205.90	77.43	123.88	17.70	119.14	**8.93**	214.2	10.77
en.12.27	24.38	39.09	28.00	9.78	17.97	2.52	17.33	2.61	20.33	**1.64**
en.12.28	50.17	80.22	57.70	20.05	37.66	7.62	36.36	**2.66**	38.97	3.25
en.12.29	103.39	161.96	117.50	40.21	75.09	10.41	72.46	**5.73**	128.35	6.71
en.12.30	211.66	333.32	239.11	81.75	150.02	20.33	145.04	**9.66**	259.21	12.99
en.14.27	27.44	43.61	31.46	8.65	21.92	3.10	21.39	2.43	22.51	**1.84**
en.14.28	56.44	90.05	65.00	17.60	45.85	6.11	44.70	**2.94**	44.53	3.67
en.14.29	116.15	182.46	131.89	35.30	90.41	12.50	88.37	**6.97**	91.53	7.79
en.14.30	238.36	377.77	269.91	70.88	184.83	22.31	178.58	**10.50**	302.14	15.98

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 `pwt` algorithm is competitive until $p < \lg \sigma$. Thus, for small σ the `pwt` 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 `pwt` 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



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

(b) *prot*, $n \approx 2^{30}$, σ=27.

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

(d) *en.4.30*, $n = 2^{30}$, σ=16.

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

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

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 σ.

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 `pwt` in Figure 4.3f.

In all datasets shown in Figure 4.3, except for Figure 4.3e, the `dd` algorithm has a better speedup than both `pwt` and `shun`, especially for datasets with small alphabets, such as `rna`, `prot` and `en.4`. In the case of Figure 4.3e, `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 `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 *malloc* and released with *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 `dd` algorithm, the figure shows memory consumption for $k = 1$. LIBCDS and `shun` use more memory during construction time. In fact, `pwt` uses up to 33 and 25 times less memory than LIBCDS and `shun`, respectively. Memory usage in `libcds` is dominated by its recursive nature, while `shun` copies the input sequence S, of $O(n \lg n)$ bits, to preserve it and to maintain its permutations in each iteration. Additionally, `shun` uses an array of size $O(\sigma \lg n)$ bits to maintain some values associated to the nodes of the *wtree*, such as the number of bits, the range of the alphabet, and the offset. In our algorithms and in `sdsl`, memory consumption is dominated by the arrays which store offset values, not by the input sequence.

The main drawback of `dd` with respect to our own `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 `dd` is almost constant. For instance, memory consumption for `rna.2GB` is 1GB, plus a small overhead for each new thread. For larger alphabets, such as `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 LIBCDS and `shun` suffer a decrement in their performance. This is evident in Figure 4.5, where we

[6]The Unicode Consortium: http://www.unicode.org/

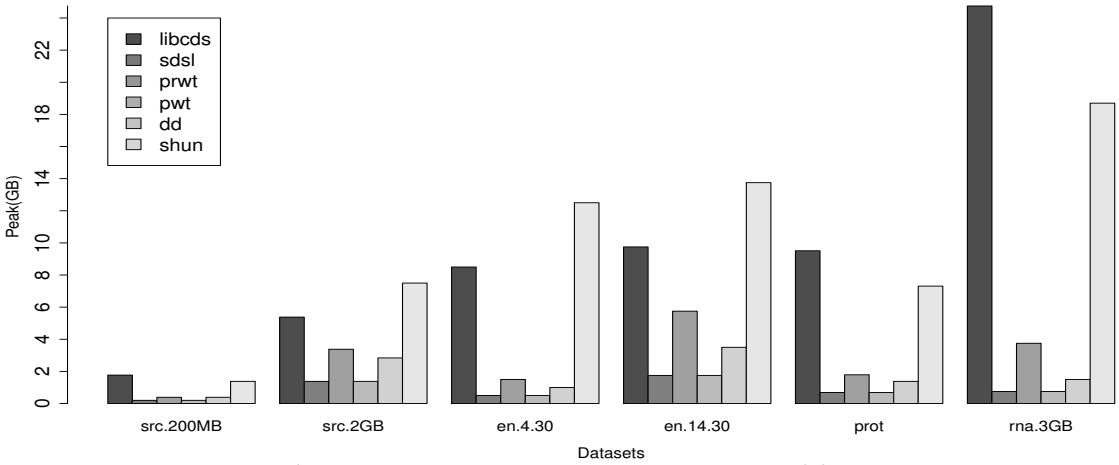


Figure 4.4: Memory consumption sorted by n.

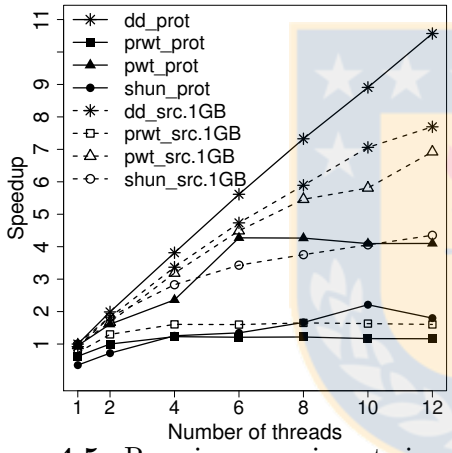


Figure 4.5: Running experiments in a machine with limited resources.

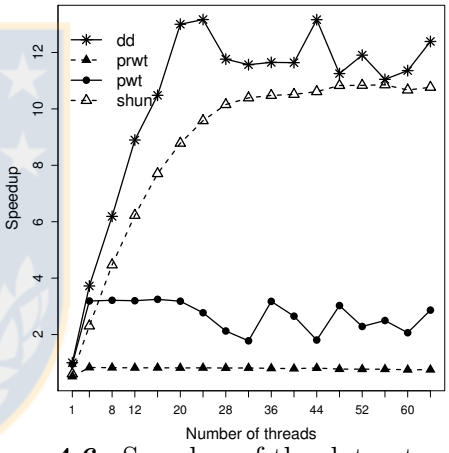


Figure 4.6: Speedup of the dataset *en.4.30* encoding each symbol with 4 bytes.

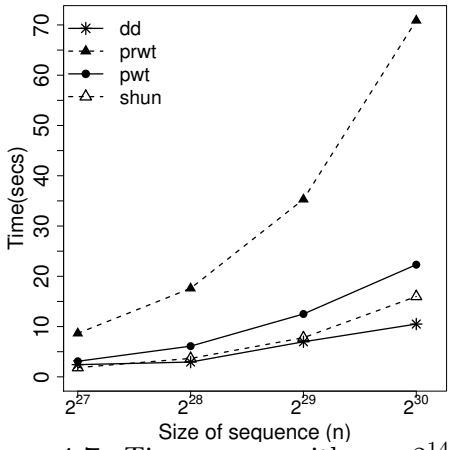


Figure 4.7: Time over n with $\sigma = 2^{14}$, 64 threads and *en.14* datasets.

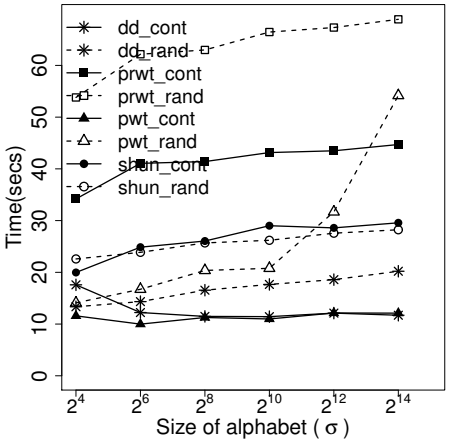


Figure 4.8: Time over σ for the best and worst cases with $n = 2^{30}$ and $p = \lg \sigma$ threads.

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 perfomance. 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 σ 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 σ with $p = \lg \sigma$ threads, the `pwt` algorithm should show a constant-time behavior, no matter the value of σ. 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 σ. 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 $n = 2^{30}$, $\Sigma = \{1 \dots 2^y\}$, 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`.

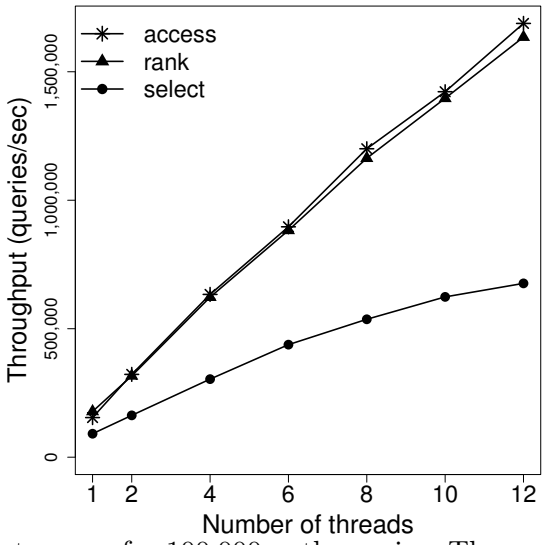


Figure 4.9: Throughput over p for 100,000 path queries. The queries were run over the dataset `en.14.29`.

`cont`, was created by writing each symbol of Σ n/σ 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[8]. 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 *wtree*s 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 *wtree*s 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

[8]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. `pwt` showed good scalability, but up to $p < \lg \sigma$. This limitation may be overcome by using `pwt` as part of `dd`, dividing the input sequence in an adequate number of subsequences.

With respect to working space, `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 `dd` algorithm increases with the number of subsequences, it can be controlled by manipulating the number of segments. In the case of `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.

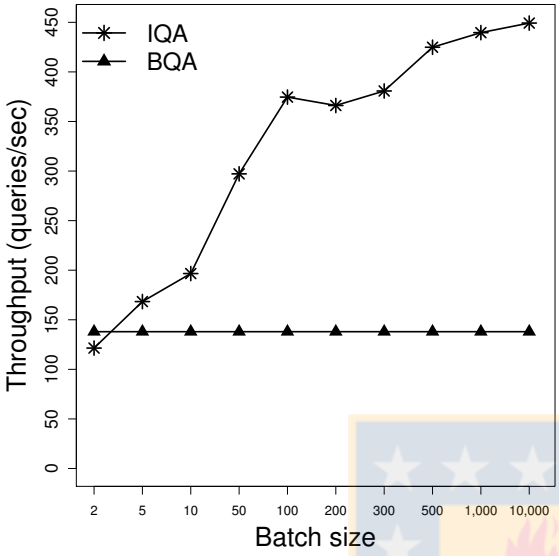
To sum up, in general, the `dd` algorithm is the best alternative for the construction of *wtree*s on multicore architectures, considering both running time and memory consumption. For domains with limited resources, `pwt`, which is a building block of `dd`, arises as a good alternative on its own.

4.3.3 Querying Experiments

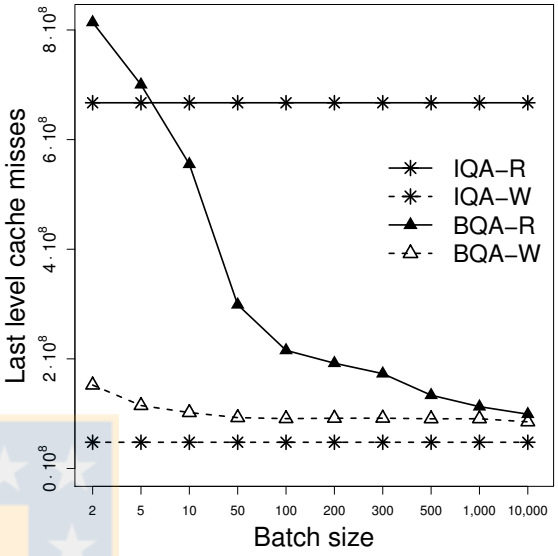
To test our querying algorithms, we generated randomly 100,000 path queries and 10,000 branch queries. For branch queries, ranges over the text were selected with random bounds, the size was fixed at 1%, and the range over the alphabet was fixed to 100%. In order to stress the `parBQA` querying algorithm,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

Figure 4.9 shows the throughput of answering 100,000 path queries in parallel using domain decomposition. We made the experiments considering `select`, `rank`, and `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 `select` is due to the fact that it takes more time because it goes down and then goes up on the levels of the *wtree*. Meanwhile `rank` and `access` only need to go down the *wtree*.

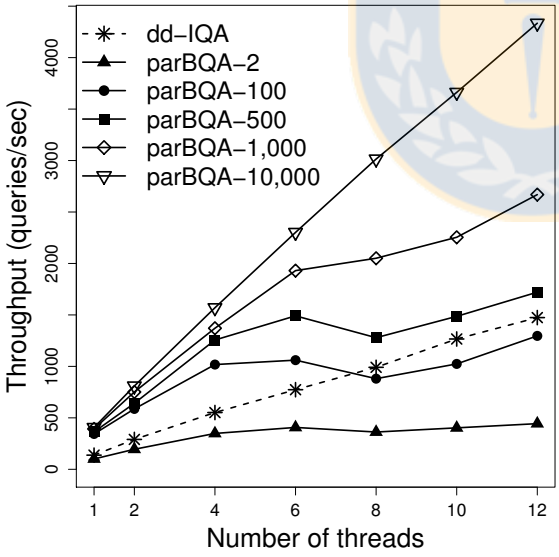
As discussed in Section 4.2, the BQA technique implies a little more programming effort but improves throughput over IQA in both sequential and parallel settings. This is shown in the Figure 4.10. Figure 4.10a shows the sequential throughput of IQA and BQA. For BQA, we created batches of 2, 5, 10, 50,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



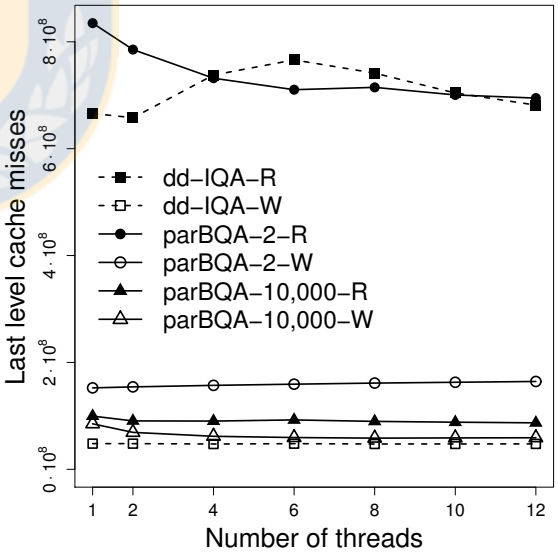
(a) Throughput comparison of IQA and BQA techniques.



(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.

Algo	Batch size		Threads						
			1	2	4	6	8	10	12
dd-IQA	-	Th	137.57	290.30	551.57	773.53	990.93	1266.12	1474.06
		RM	665860362	657818591	737767750	766095719	741319074	704190961	681491229
		WM	48081819	48041149	47328838	47987211	47378656	47449644	47528758
parBQA	2	Th	100.56	194.62	348.35	406.85	362.18	402.74	443.61
		RM	834877009	785435590	731840791	710330088	714685675	700531663	694619108
		WM	152382639	154227895	157085490	159295211	161359565	162841305	164377685
	5	Th	146.82	301.31	554.61	628.20	548.24	643.29	712.03
		RM	701741574	661159773	557769396	567348283	578432715	563580603	545671748
		WM	114809684	117015656	118332101	120191416	120199560	121362289	121932783
	10	Th	221.74	397.68	679.61	803.72	696.79	811.65	884.71
		RM	530267689	529014444	447673290	465907488	491537029	466856907	451301658
		WM	101311270	103284485	105393194	105746122	106704957	107055350	107576446
	50	Th	314.22	523.12	952.55	1013.34	926.26	967.90	1202.79
		RM	276960062	301410175	281768855	284070254	267223770	260184414	240708360
		WM	91757596	95467415	96557132	96527690	96932334	97648510	97739287
	100	Th	343.40	586.75	1017.71	1060.59	880.60	1023.97	1296.30
		RM	215546652	233695241	217395892	225892186	212776916	195072536	185470773
		WM	90939844	93539193	95889503	95359893	95995964	96441443	96722795
	200	Th	335.09	608.06	1061.12	1133.73	936.10	1212.99	1325.53
		RM	191197574	186654101	171230380	180167317	168084253	153219450	141833763
		WM	91722746	93242449	94768254	94731394	95637667	95757922	96209927
	300	Th	348.47	659.30	1118.17	1217.46	1013.05	1232.40	1443.27
		RM	172426822	167873358	162909579	155472257	157766136	140711549	129182950
		WM	91780910	92476066	94721318	94921917	95855243	96242555	96498170
	500	Th	363.48	641.26	1256.25	1491.03	1278.82	1486.17	1720.98
		RM	151292247	145254144	130744590	130601523	132549010	121846145	119806629
		WM	91881274	92461973	93689618	94788891	95593715	96078659	96722026
	1,000	Th	392.69	751.09	1370.70	1930.55	2050.69	2254.38	2669.01
		RM	113568181	124760874	113050092	109214768	111164439	102412983	108209001
		WM	90803770	81616490	77884008	76918965	76911836	76733067	76743158
	10,000	Th	406.16	809.51	1569.76	2301.69	3014.51	3662.38	4334.43
		RM	99570459	90565058	90249576	92412743	89657938	88164389	86953096
		WM	85139187	69100728	61783541	59445665	58306473	58747291	59055887

Table 4.3: Throughput (Th), last level read misses (RM) and last level write misses (WT) of the `dd-IQA` and `parBQA` parallel algorithms. The experiments were run in the `machine A` with the dataset `en.14.29`.

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 `dd-IQA` and `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 `parBQA` with a batch size of 10,000, 2.9 times better than `dd-IQA` for all the threads. For `dd-IQA`, the improvement with respect to itself was 10.7 times. For `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 `parBQA`, with a batch size less than 10,000, the $O(num\_queries/batch\_size)$ 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 `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 *wtree*s where the input sequence S and the *wtree* do not fit. Following some implementation ideas of Sdsl[51], we can read the input sequence in buffers to construct partial *wtree*s for each buffer and finally merge all of them to obtain the final *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 *wtree* as the buffer.

2. Create a partial *wtree* without rank/select structures taking the buffer as input. The partial *wtree* can be constructed in parallel using our `dd` algorithm with $O(b \lg \sigma/p)$ time and $O(1)$ span. (We could also use the `pwt` if the available memory is scarce). The starting position of each node in the partial *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 *wtree*s. These positions are computed by performing a parallel prefix sum[65] over the values of the arrays L's, similar to the `dd` algorithm. It takes $O(b\sigma/p + \lg p)$ time and $O(\lg(b\sigma))$ span.

5. The final *wtree* is constructed using Function mergeBA with $O(n \lg \sigma/pw)$ and $O(1)$ span, where w is the word size of the architecture.

The extension takes $O(n \lg \sigma/p + b\sigma/p + \lg p)$ time and $O(n/b + \lg(b\sigma))$ 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 *wtree*s. Additionally, observe that we can use the entire main memory as the buffer, storing the partial *wtree*s 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 Σ 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 σ with $O(n+\sigma)$ work and $O(\lg n + \sigma)$ span. Assume that the resulting Huffman codes and the length of each code are (temporarily) stored in a table H of size $O(\sigma \lg n)$. 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 $O(\lg n+\sigma)$ 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, $v.first$ and $v.last$, indicating the adjacency list of v, sorted counterclockwise around v and starting with vs parent edge. Notice that the number of children of v is $(v.last - v.first)$. 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 of e, e', where the $e'.src = e.tgt$ and $e'.tgt = e.src$. For $x \in \{e.src, e.tgt\}$, we use $next(e.x)$ and $first(e.x)$ 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.

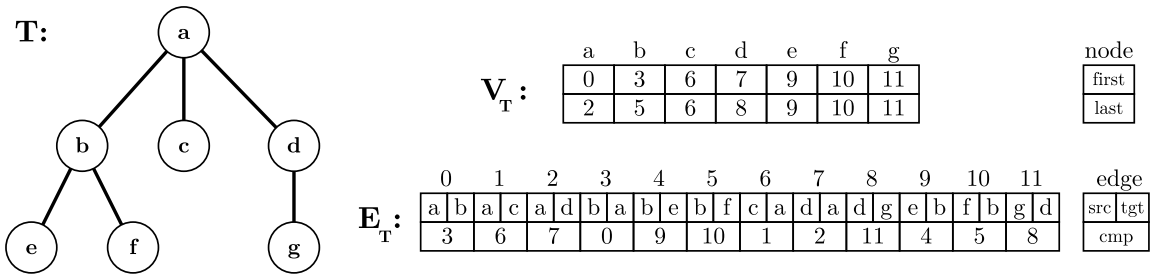


Figure 5.1: Tree representation used as input to the `PFEA` algorithm. The nodes are represented in the array V_T and the edges in the array E_T.

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, *threads*.
Output: The balanced parenthesis sequence P of T.

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

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

$\lceil 2n/s \rceil$, is divisible by *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 $\texttt{sum}(P, \pi, j, i)$, 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 $\texttt{sum}(P, \pi, 0, j-1)$. 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 : `RMMT` represented as arrays e', m', M', n' and universal lookup tables

```
1  o = ⌈2n/s⌉ − 1 // # internal nodes
2  e' = array of size ⌈2n/s⌉
3  m', M', n' = arrays of size ⌈2n/s⌉ + o
4  ct = ⌈2n/s⌉/threads
5  parfor t = 0 to threads − 1 do
6      e'_t, m'_t, M'_t, n'_t = 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         m'[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 t = 1 to threads − 1 do
24     for chk = 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 = ⌈lg threads⌉
parfor st = 0 to 2^lvl − 1 do
    for l = ⌈lg(2n/s)⌉ − 1 downto lvl do
        for d = 0 to 2^(l−lvl) − 1 do
            i = d + 2^l − 1 + st ∗ 2^(l−lvl)
            concat(i, m', M', n')
for l = lvl − 1 to 0 do
    parfor d = 0 to 2^l − 1 do
        i = d + 2^l − 1
        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 √(2^w) − 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 ≥ 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 *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 *near_fwd_pos* as an example. This table is used to support the `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 $O(\sqrt{2^w}\texttt{poly}(w))$ work and has span $O(1)$, where w is the machine word size. Thus, the total work of `PSTA` is $T_1 = O(n+\lg p+\sqrt{2^w}\texttt{poly}(w))$ and its span is $O(\lg n)$. This gives a running time of $T_p = O(T_1/p + T_\infty) = O(n/p + \lg p + \sqrt{2^w}\texttt{poly}(w)/p)$

on p cores[1]. The speedup is $T_1/T_p = O\left(\frac{p(n+\sqrt{2^w}\texttt{poly}(w))}{n+\sqrt{2^w}\texttt{poly}(w)+p\lg p}\right)$. Under the assumption that $p \ll n$, the speedup approaches $O(p)$. Moreover, the parallelism T_1/T_∞ (the maximum theoretical speedup) of `PSTA` is $\frac{n+\sqrt{2^w}\texttt{poly}(w)}{\lg n}$.

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, $O(S_1 p)$, 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 within a constant factor [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.*

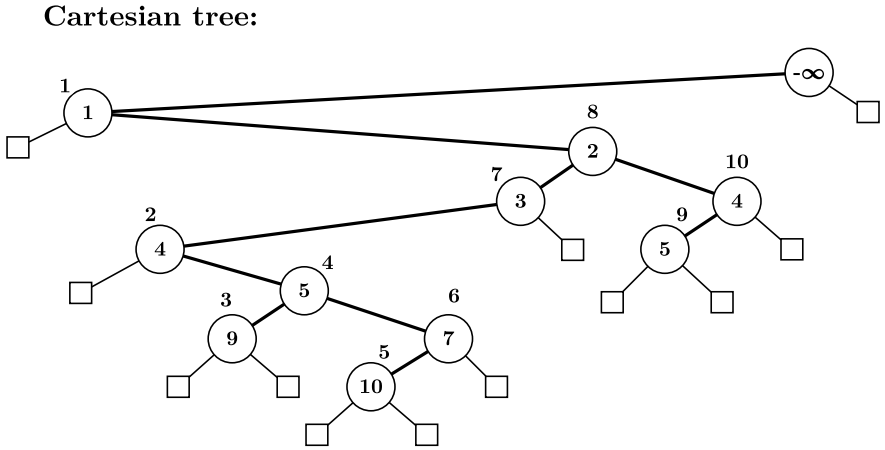
5.3 Parallel Algorithm to Support Constant-Time Queries

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 $\tau = \lceil 2n/w^c \rceil$ `RMMT`s 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 $j = min\{j' | j' > i \wedge x_{j'} < x_i\}$. 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 sucessor and the closest right ancestor are well-defined for all x_i,

[1]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 $p > n/\lg n \rightarrow \lg p = \Theta(\lg n)$.



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

indexes :	11	1	8	10	10	9	9	9	10	8	7	7	2	4	6	6	5	5	5	6	4	3	3	3	4	2	2	7	8	1	1	11		
ET :	-∞	1	2	4	4	5	5	5	4	2	3	3	4	5	7	7	10	10	10	7	5	9	9	9	5	4	4	3	2	1	1	-∞		
δ(ET) :	-∞	-∞	-∞	-∞	4	4	4	4	4	2	2	3	3	3	3	7	7	7	7	7	5	5	5	5	5	5	5	2	2	2				

(b) Euler Tour ET of the cartesian tree in Figure 5.2a and the sequence $\delta(ET)$. The first ocurrence 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 ocurrence 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 $(y_h, y_{h+1}, \ldots, y_i)$ be the subsequence of ET between y_h and y_i, and let $(k = j_h, j_{h+1}, \ldots, j_i = j)$ be the sequence of indices such that $y_t = x_{j_t}$, for all $h \leq t \leq i$, where x_{j_t} is the j_t-th element in ET.

To prove the lemma, we need to show that $j_{h+1} < j_h$ and $j_{t+1} > j_t$ 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 $\delta(z_{t-1}, y_t) = z_{t-1} = z_h = x_k$ 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,$(y_{h+1}, y_{h+2}, \ldots, y_i)$ is the sequence of nodes in this path. Since y_{t+1} is the right child of y_t for all $h < t < i$, we have that $j_{t+1} > j_t$.

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 $T^{\mathcal{B}}$ with a particular embedding $\mathcal{B}$: for each node v of $T^{\mathcal{B}}$, the children of v are ordered by their height, with the highest child in the leftmost position.

Lemma 2. *Given a tree $T^{\mathcal{B}}$ with embedding $\mathcal{B}$ and n nodes, the ladders of $T^{\mathcal{B}}$ 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,

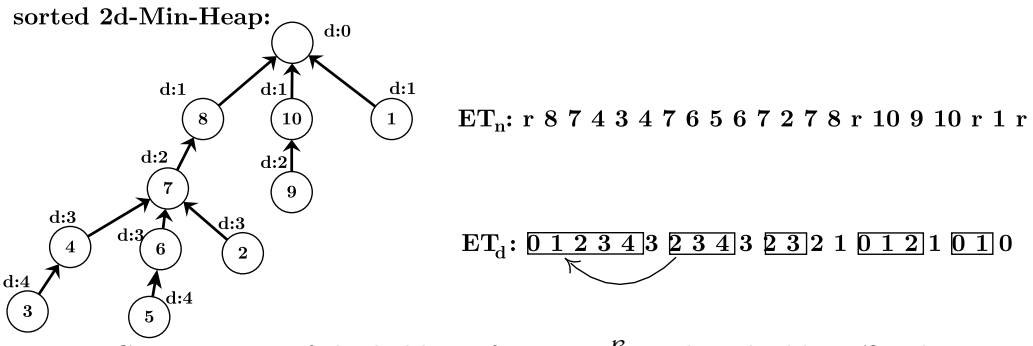


Figure 5.3: Computation of the ladders of a tree $T^{\mathcal{B}}$, with embedding $\mathcal{B}$. The tree $T^{\mathcal{B}}$ is the result of applying the embedding $\mathcal{B}$ to the tree of Figure 3.4. In the tree, the depth of each node is shown. For example, $d:3$ means that the depth of a node 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 $T^{\mathcal{B}}$ 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 $T^{\mathcal{B}}$ 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 $\mathcal{B}$, 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 `level_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 `PFEA` 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 `level_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 $O(n)$ `level_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 $\Sigma = \{0, \lceil \lg |ET_d| \rceil - 1\}$. For example, to extend the subsequence $ET_d[a..b]$ with a node with depth d', we need to perform $\texttt{select}_{d'}(ET_d, \texttt{rank}_{d'}(ET_d, a))$. 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 `pwt` algorithm, with $O(n \lg n)$ work, $O(n)$ span and $O(n \lg n)$ working space, or the `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 `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 $T^{\mathcal{A}}$ with an arbitrary embedding $\mathcal{A}$ and n nodes, the ladders of $T^{\mathcal{A}}$ 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 $\mathcal{A}$ of $T^{\mathcal{A}}$ to $\mathcal{B}$. To compute the embedding $\mathcal{B}$, we need to order all the children of the nodes of $T^{\mathcal{A}}$ by height with the highest in the leftmost position. To do this, we use the `PFEA` algorithm to compute the folklore encoding of T and then construct its simplified NS-representation to use the `height` operation to obtain the height of all the nodes. Both the folklore enconding and the simplified NS-representation can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. The `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 $T^{\mathcal{A}}$. 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 $\mathcal{B}$, we use Lemma 2 to finish the proof.

In the NS-representation, the 2d-Min-Heap has τ nodes, and therefore, the 2d-Min-Heap and its ladders can be computed with $O(\tau \lg \tau)$ work, $O(\lg^2 \tau)$ span and $O(\tau \lg \tau)$ working space.

□

Pătraşcu's bitmap: Navarro and Sadakane use the sparse bitmap of Pătraşcu [106] to represent a bitmap with 2τ 1's and $2\tau w^c$ 0's using $O(\tau \lg w^c + \frac{\tau w^c t^t}{\lg^t(\tau w^c)} + (\tau w^c)^{3/4})$ bits and supporting rank/select queries in $O(t)$ time. Pătraşcu demostrated 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 $B \lg B = \frac{\varepsilon \lg n}{t}$, and $r = B^t = (\frac{\lg n}{t})^{\Theta(t)}$. 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 segmens 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 `RMMT` 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 $O(\lg(n/r))$ 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 $O(\lg \lg(n/r))$ time. Finally, Pătraşcu's sparse bitmap can be computed in parallel, with $r = (\frac{\lg n}{t})^{\Theta(t)}$, in $O(n + \frac{nt^t}{\lg^t n})$ work, $O(t + \lg(\frac{nt^t}{\lg^t n}))$ span and $O(n)$ working space. In the context of succinct trees, the work is $O(\tau w^c + \frac{\tau w^c t^t}{\lg^t \tau w^c})$, the span is $O(t + \lg(\frac{\tau w^c t^t}{\lg^t(\tau w^c)}))$ 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 $O(\lg_w n)$ time. A fusion tree is essencially 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 $O(\lg \lg n)$ time, for a sorted list. Since the $\lg \tau$ 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 $h = \lceil \lg_m(n+1) \rceil + 1$
- 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, $O(\lg \lg n)$ span and $O(n)$ working space.

In the NS-representation, to support `fwd_search` and `bwd_search` operations, τ fusion trees are constructed over τ sorted arrays of $O(\lg \tau)$ integers. Considering the previous parallel bounds, the τ fusion trees can be constructed with $O(\tau \lg \tau)$ work, $O(\lg \lg \tau)$ span and $O(\tau \lg \tau)$ 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 $s = \lceil \frac{\lg n}{4} \rceil$. A query from i to j, $\texttt{RMQ}(A, i, j)$, 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 $B_{\lceil i/s \rceil}, \ldots, B_{\lfloor j/s \rfloor - 1}$ 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* $C^{can}_{B_x}$. 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 $P[1, C_s][1, s][1, s]$. 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 τ `RMMT`s. 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 tighest matching pairs of parentheses (i, j), with $j =$ `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 pionner (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 τ 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, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span and $O(n \lg n)$ bits of working space. This representation supports the operations in Table 3.1 in $O(c)$ time, with $c > 3/2$.*

Proof. Each of the τ `RMMT`s can be constructed with $O(\lg^c n)$ work, $O(\lg \lg^c n)$ span and $O(\lg^c n \lg \lg^c n)$ bits of working space using Theorem 2. All the τ `RMMT`s 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 constructred with $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$ work, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span and $O(n + \frac{n}{\lg^{c-1} n})$ working space. Thus, total work is $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$, the maximun span is $O(c + \lg(\frac{nc^c}{\lg^c n}))$ 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 `sdsl`, the state-of-the-art implementations of the `RMMT`, which were written in C++.

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

Table 5.1: Datasets used in the experiments of succinct trees.

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 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 libcds [23] and sdsl [51], which are state-of-the-art implementations of the 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 libcds and 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 psta, the sequential version of psta, called seq, libcds, and 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] http://pizzachili.dcc.uchile.cl

[3] http://dumps.wikimedia.org/enwiki/20150112/enwiki-20150112-pages-articles.xml.bz2 (January 12, 2015)

[4] http://wiki.openstreetmap.org/wiki/Planet.osm (January 10, 2015)

p	wiki	prot	dna	ctree	osm
libcds	33.17	44.27	75.93	140.71	339.43
sdsl	1.94	2.67	4.57	8.35	18.10
seq	2.81	4.10	7.25	12.14	28.00
1	2.81	4.10	7.15	12.17	28.05
4	.72	1.05	1.86	3.05	7.07
8	.40	.58	.95	1.57	3.55
12	.31	.43	.72	1.12	2.55
16	.24	.32	.55	.85	1.89
20	.19	.29	.49	.74	1.58
24	.19	.26	.42	.68	1.45
28	.16	.25	.43	.62	1.30
32	.18	.25	.38	.62	1.16
36	.20	.21	.36	.52	1.08
40	.21	.23	.35	.50	1.04
44	.22	.25	.34	.51	.97
48	.21	.26	.37	.49	.99
52	.27	.30	.36	.50	.93
56	.30	.36	.42	.50	.93
60	.27	.40	.39	.50	.93
64	.30	.33	.38	.54	.90

Table 5.2: Running times of the algorithms `libcds`, `sdsl`, and `PSTA`, in seconds. `seq` corresponds to the sequential execution of `PSTA`.

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`.

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

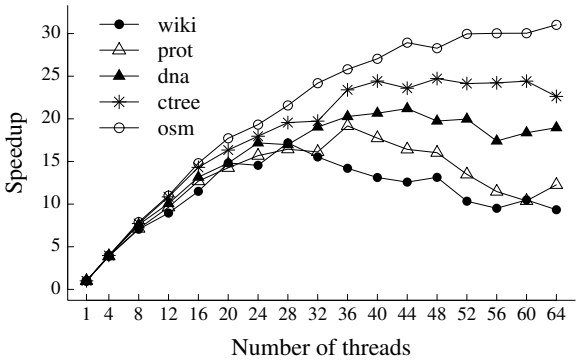


Figure 5.4: Speed-up of `PSTA` compared to `seq`.

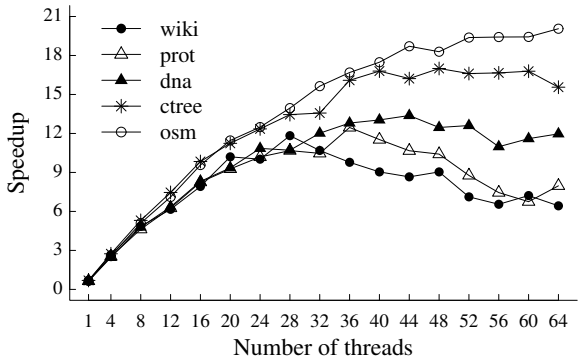


Figure 5.5: Speed-up of `PSTA` compared to `sdsl`.

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 `sdsl`. 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 `sdsl`,

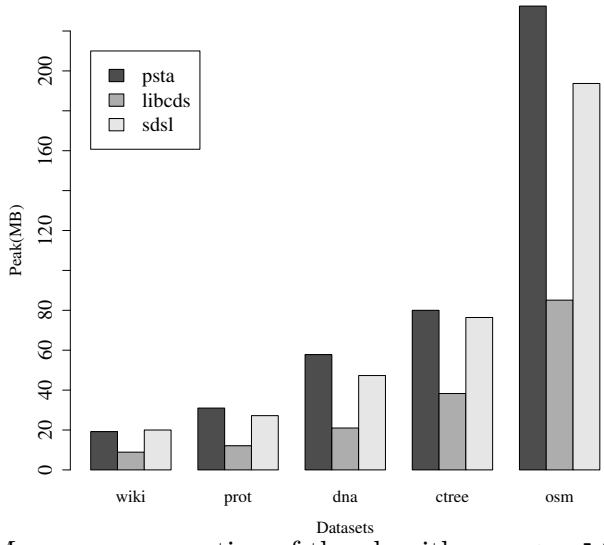


Figure 5.6: Memory consumption of the algorithms `psta`, `libcds` and `sdsl`.

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` achieves 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 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

p	ctree25	prot	dna	ctree25^{+16}	prot^{+16}	dna^{+16}	ctree25^{+32}	prot^{+32}	dna^{+32}
seq	5.67	52.87	31.33	55.04	473.92	275.06	102.94	887.61	514.73
1	6.07	34.83	57.29	54.99	275.23	474.07	102.80	514.09	886.01
4	1.68	9.35	15.62	14.22	70.84	121.92	26.15	130.30	224.86
8	1.05	5.77	9.80	7.33	36.39	62.79	13.32	66.35	114.42
12	0.87	3.99	6.90	5.02	25.18	43.23	8.99	45.02	77.49
16	0.73	3.66	5.57	3.91	19.59	33.50	6.86	34.37	59.15
20	0.76	3.42	5.53	3.21	16.06	27.61	5.58	28.16	48.26
24	0.76	3.41	5.25	2.77	13.91	24.03	4.73	23.96	41.04
28	0.66	3.28	5.36	2.47	12.40	21.23	4.16	20.87	36.05
32	0.67	3.37	5.71	2.30	12.52	19.49	4.08	18.62	35.22
36	0.68	3.23	5.57	2.27	11.61	19.95	3.71	18.70	32.29
40	0.68	3.15	5.48	2.16	10.91	18.75	3.38	17.17	29.59
44	0.68	3.16	5.44	2.04	10.20	17.70	3.19	15.98	27.56
48	0.69	**2.97**	5.19	1.92	9.67	16.79	3.00	15.02	25.82
52	0.64	3.05	5.37	1.84	9.23	16.01	2.83	14.12	24.33
56	0.65	3.01	5.33	1.79	8.83	15.18	2.71	13.37	22.87
60	**0.61**	3.19	5.35	**1.72**	8.46	**14.75**	2.58	12.63	21.82
64	0.71	3.05	**5.18**	1.80	**8.29**	15.13	**2.55**	**12.32**	**20.90**

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.

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 demostrate 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 comunication 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

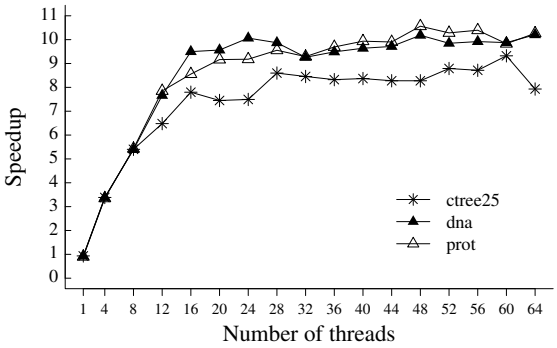


Figure 5.7: Speedup of the `PFEA` algorithm with datasets `ctree25`, `dna` and `prot`.

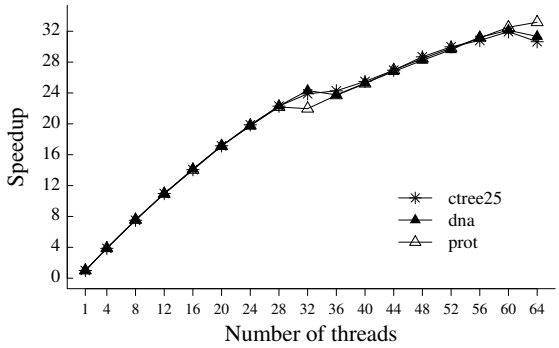


Figure 5.8: Speedup of the `PFEA` algorithm with datasets `ctree25`, `dna` and `prot`, artificially increasing the workload with 16 `CAS` operations per edge.

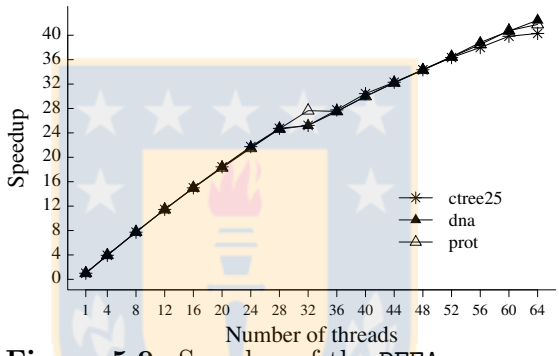


Figure 5.9: Speedup of the `PFEA` algorithm with datasets `ctree25`, `dna` and `prot`, artificially increasing the workload with 32 `CAS` operations per edge.

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 `sdsl` 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 `sdsl` 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 $|V| = n \geqslant 3$, $|E| = m$, with a canonical ordering Π 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 V_T = V_G
2 parfor i = 0 to |V_G| − 1 do
3     parfor j = first(V_G[i]) + 1 to last(V_G[i]) do
4         n1 = E_g[j − 1].tgt                          // neighbor 1
5         n2 = E_g[j].tgt                              // neighbor 2
6         if co(V_G[i]) < co(V_G[n1])) AND co(V_G[i]) > co(V_G[n2])) then
7             addEdge(E_T, V_G[i], n2)
```

Algorithm 14: Parallel canonical spanning tree algorithm (`PCoST`)

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 $n_1^v, n_2^v, \ldots$ be the neighbors of v in counterclockwise order. The parent of v is its neighbor at the j-th position, such that $co(v) < co(n_{j-1}^v)$ and $co(v) > co(n_j^v)$. 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 auxiliar array C to store the lower-numbered and higher-numbered neighbors of each vertex of G, an auxiliar 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 = an array of length |V_G|
2  LE = an array of length |E_T|
3  S_co = an array of length |E_G| + 2
4  chk = |E_T|/threads
5  parfor i = 0 to |V_G| − 1 do
6      [C[i].l, C[i].h] = parallelCount(V_G, E_G, V_T, i)
7  parfor t = 0 to threads − 1 do
8      for i = 0 to chk − 1 do
9          j = t ∗ chk + i
10         if co(E_T[j].src) < co(E_T[j].tgt) then // forward edge
11             LE[j].value = ⊕("(","]"∗C[E_T[j].tgt].l)
12             LE[j].rank = C[E_T[j].tgt].l + 1
13             if E_T[j].tgt is a leaf then
14                 LE[j].succ = E_T[j].cmp
15             else
16                 LE[j].succ = first(E_T[j].tgt) + 1
17         else // backward edge
18             LE[j].value = ⊕(")","["∗C[E_T[j].src].h)
19             LE[j].rank = C[E_T[j].src].h + 1
20             if E_T[j] is the last edge in the adjacency list of E_T[j].src then
21                 LE[j].succ = first(E_T[j].tgt)
22             else
23                 LE[j].succ = next(E_T[j].tgt)
24 parallel_list_ranking(LE)
25 parfor t = 0 to threads − 1 do
26     for i = 0 to chk − 1 do
27         j = t ∗ chk + i
28         S_co[LE[j].rank ... LE[j + 1].rank − 1] = LE[j].value
29 S_co[0] ="("
30 S_co[|E_G| + 1] =")"
```

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 $co(e.src)$ is lower than $co(e.tgt)$, 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 $S[LE[i].rank \dots LE[i+1].rank - 1]$. 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 $O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c)$ work, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ 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:

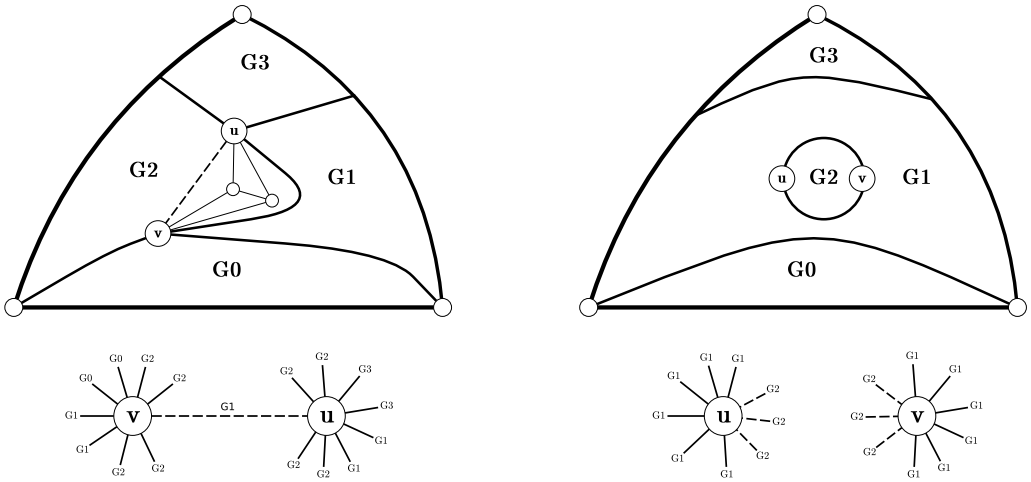
(a) $\forall i \in \{0, \ldots, k\}, |V_i| = O(n/p)$

(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 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 Π_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 Π_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

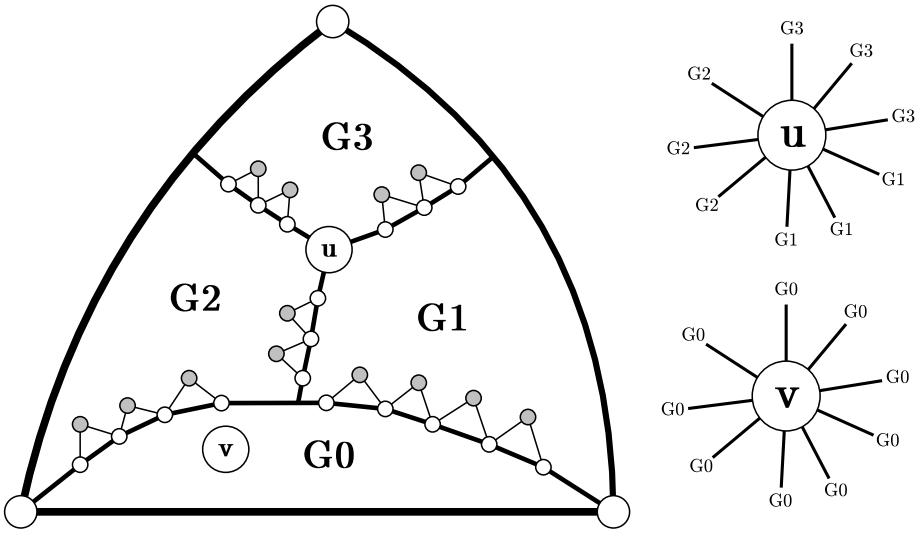


Figure 6.2: Decomposition of a triangulated plane graph.

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, Π, as follows:

- Π 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.

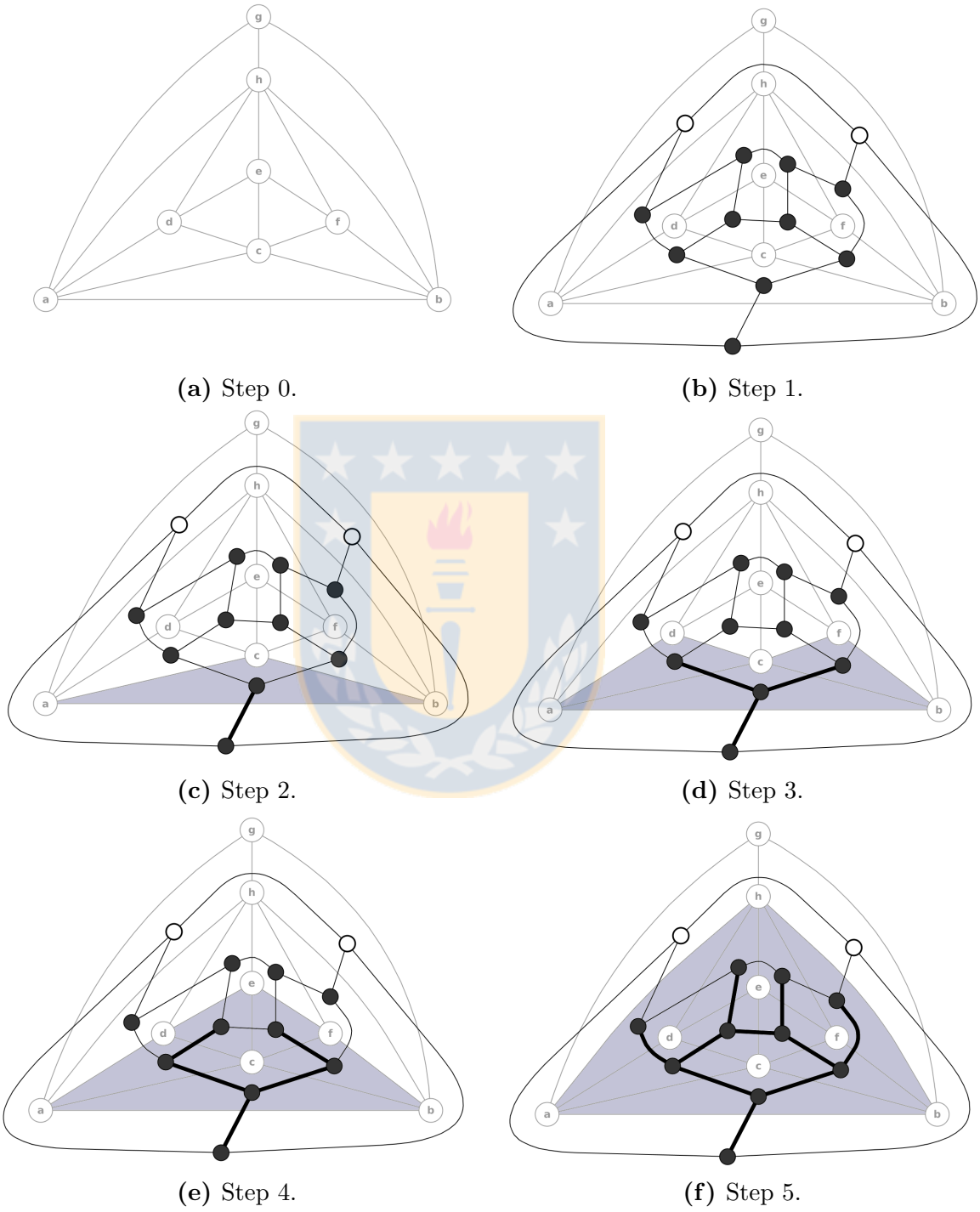
As Π_0 is a canonical ordering and the first vertex in Π_1, v, is incident to Π_0, $\Pi_0 \cup v$ is a canonical ordering of $G_0 \cup v$. At the same time, v is part of the canonical ordering Π_1, then $(\Pi_0 \cup \Pi_1)$ is a canonical ordering of G_1. Therefore, following this procedure, we can obtain 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



(a) Step 0. **(b)** Step 1.

(c) Step 2. **(d)** Step 3.

(e) Step 4. **(f)** Step 5.

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 *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 $f : \langle e_1, e_2, e_3 \rangle$ corresponds to three edges, ordered in counterclockwise order, where $e_1.tgt = e_2.src$, $e_2.tgt = e_3.src$, $e_3.tgt = e_1.src$, $e_1.src < e_2.src$ and $e_1.src < e_3.src$. An edge e of G is the *owner* of a face $f = \langle e_1, e_2, e_3 \rangle$ 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 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_i^d 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 tradicional 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 Δ 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, $l_i^{T_2}$, and higher, $h_i^{T_2}$, 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 $l_i^{T_2}$ "]"s and $l_i^{T_3}$ "}"s. Each time when we visit a backward edge, write $h_i^{T_3}$ "{"s followed by $h_i^{T_2}$ "["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`$(E_G, c)]$
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.

```
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

Input : An array of edges E_G and the index of an edge in E_G, i.
Output : The position of the edge $E_G[i]$ in the face where it is involved, starting from the owner of the face.

```
1 owner = ownership(E_G, i)
2 if i = owner then
3 |  return 0
4 next = prev(E_G[i].cmp)
5 if next = owner then
6 |  return 2
7 return 1
```

Function newIndex

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 ocurrence 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 $C_{T_3}[v].h$ "{"

Input : An adjacency list representation of the plane graph G consisting of arrays V_G and E_G and the number of threads, *threads*. Vertices in graph G have their position in the canonical ordering of G.

Output: A three-type parentheses sequence S'_{rz} induced by the realizer T_1, T_2 and T_3 of G.

```
1  C_{T_2} = an array of length |V_G|
2  C_{T_3} = an array of length |V_G|
3  S'_{rz} = an array of length |E_G| + 2
4  [T_1, T_2, T_3] = buildRealizers(V_G, E_G, threads)
5  newOrder(V_G, E_G)
6  parfor i = 0 to |V_G| do
7      [C_{T_2}[i].l, C_{T_2}[i].h, C_{T_3}[i].l, C_{T_3}[i].h] = parallelCount(V_G, E_G, i)
8  LE = an array of length |E_{T_1}|        // E_{T_1} is the array of edges of realizer T_1
9  chk = |E_{T_1}|/threads
10 parfor t = 0 to threads − 1 do
11     for i = 0 to chk − 1 do
12         j = t * chk + i
13         if co(E_{T_1}[j].src) < co(E_{T_1}[j].tgt) then // forward edge
14             LE[j].value = ⊕("(","]"*C_{T_2}[E_{T_1}[j].tgt].l,"}"*C_{T_3}[E_{T_1}[j].tgt].l)
15             LE[j].rank = C_{T_2}[E_{T_1}[j].tgt].l + C_{T_3}[E_{T_1}[j].tgt].l + 1
16             if E_{T_1}[j].tgt is a leaf then
17                 LE[j].succ = E_{T_1}[j].cmp
18             else
19                 LE[j].succ = first(E_{T_1}[j].tgt) + 1
20         else // backward edge
21             LE[j].value = ⊕("{"*C_{T_3}[E_{T_1}[j].tgt].h,"["*C_{T_2}[E_{T_1}[j].tgt].h,")")
22             LE[j].rank = C_{T_2}[E_{T_1}[j].src].h + C_{T_3}[E_{T_1}[j].src].h + 1
23             if E_{T_1}[j] is the last edge in the adjacency list of E_{T_1}[j].src then
24                 LE[j].succ = first(E_{T_1}[j].tgt)
25             else
26                 LE[j].succ = next(E_{T_1}[j].tgt)
27 parallel_list_ranking(LE)
28 parfor t = 0 to threads − 1 do
29     for i = 0 to chk − 1 do
30         j = t * chk + i
31         S'_{rz}[LE[j].rank ... LE[j + 1].rank − 1] = LE[j].value
32 S'_{rz}[0] = "("
33 S'_{rz}[|E_G| + 1] = ")"
```

Algorithm 17: Parallel graph encoding algorithm - realizers version(`PGEA-rz`)

and $C_{T_2}[v].h$ "[", depending on whether the edge is a forward or a backward edge; *succ* is the index in LE of the next edge in the Euler tour; and *rank* is the number

Input : An adjacency list representation of a tree T consisting of arrays V and E and the number of threads, *threads*.
Output: The tree T with a new ordering.

```
1  ET = an array of length |E|
2  chk = |E|/threads
3  parfor t = 0 to threads − 1 do
4      for i = 0 to chk − 1 do
5          j = t ∗ chk + i
6          if co(E_T[j].src) < co(E_T[j].tgt) then // forward edge
7              ET[j].rank = 1
8              if E[j].chld is a leaf then
9                  ET[j].succ = ET[j].cmp
10             else
11                 ET[j].succ = first(E[j].tgt) + 1
12         else
13             ET[j].rank = 0
14             if E[j] is the last edge in the adjacency list of E[j].src then
15                 ET[j].succ = first(E[j].tgt)
16             else
17                 ET[j].succ = next(E[j].tgt)
18 parallel_list_ranking(ET)
19 parfor t = 0 to threads − 1 do
20     for i = 0 to 2 ∗ chk − 1 do
21         j = t ∗ chk + i
22         if first(E[j].src) = j then // First ocurrence of the node E[j].src
23             V[E[j].src].co = ET[j].rank
```

Algorithm 18: Parallel algorithm `newOrder` to define a new order of the nodes in a maximal plane graph

of parentheses in *value*, used to compute the rank of each symbol in S'_{rz}.

The rest of the explanation of the `PGEA-rz` algorithm is the same as the `PGEA` algorithm.

The theoretical analysis of the `PGEA-rz` algorithm is given by the complexity of the functions `buildRealizers` and `newOrder`, and the complexity of the rest of the algorithm, which is, essentially, the complexity of the `PGEA` algorithm. Algorithm `newOrder` has the same complexity of the `PFEA` algorithm, $O(n)$ work, $O(\lg n)$ span and $T_p = O(n/p + \lg p)$. The complexity of the function `buildRealizers` is also $O(n)$ work, $O(\lg n)$ span and $T_p = O(n/p + \lg p)$. Thus, the complexity of the `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 `PGEA`, we can use Algorithm `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 `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 `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 `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 `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 `rank`, `select`, `match`, `first` and `last` operations and bit-vectors B_1 and B_2 must support `rank` and `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 $O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c)$ work, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ 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.

```
1  V_{T_1} = V_G
2  V_{T_2} = V_G \ {v_1}              // v_1 is the vertex with canonical ordering 1
3  V_{T_3} = V_G \ {v_1, v_2}         // v_2 is the vertex with canonical ordering 2
4  L = an array of length |V_G|
   // Stage 1
5  parfor i = 0 to |V_G| do
6      parfor j = first(V_G[i]) to last(V_G[i]) − 1 do
7          n1 = E_g[j].tgt                          // neighbor 1
8          n2 = E_g[j + 1].tgt                      // neighbor 2
9          if co(V_G[i]) < co(V_G[n1])) AND co(V_G[i]) > co(V_G[n2])) then
10             L[i].p = j + 1
11         else if co(V_G[i]) > co(V_G[n1])) AND co(V_G[i]) < co(V_G[n2])) then
12             L[i].q = j
   // Stage 2
13 parfor i = 0 to |V_G| do
14     addEdge(E_{T_1}, E_G[L[i].p].tgt, E_G[L[i].p].src)
15     addEdge(E_{T_2}, E_G[L[i].q].tgt, E_G[L[i].q].src)
16     parfor j = L[i].p + 1 to L[i].q − 1 do
17         addEdge(E_{T_3}, E_G[j].src, E_G[j].tgt)
```

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, $3 \leqslant k \leqslant n$. 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, $O(\lg n)$ span and $T_p = O(n/p + \lg p)$ time, using a parallel prefix sum algorithm over the lengths of the adyacency list of the tree spanning trees. Therefore, the complexity of the Algorithm 19 is $O(n)$ work, $O(\lg n)$ span and $T_p = O(n/p + \lg p)$ time.

	Dataset	Nodes (n)	Edges (m)	Min fan-out	Max fan-out
1	worldcities	2,243,467	6,730,395	3	36
2	rand-1M	1,000,000	2,999,994	3	20
3	rand-2M	2,000,000	5,999,994	3	14
4	rand-4M	4,000,000	11,999,994	3	18
5	rand-8M	8,000,000	23,999,994	3	17
6	rand-10M	10,000,000	29,999,994	3	24

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

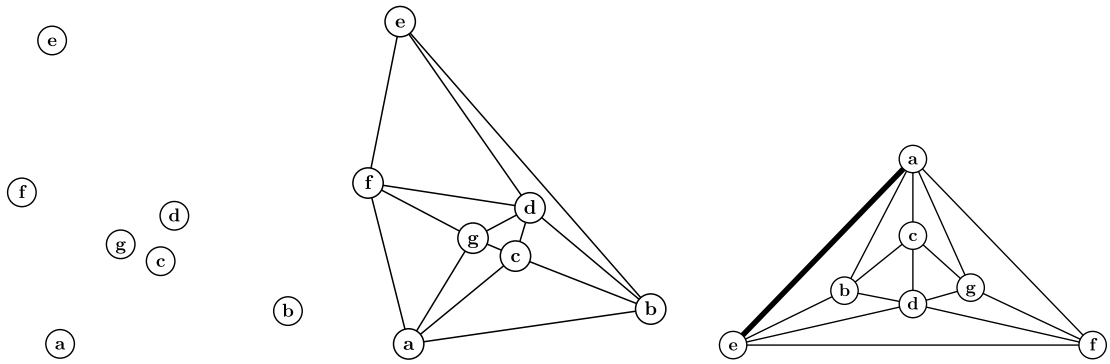
We implemented the `PGEA` algorithm to compute the bit-vectors S_1, S_2 and S_3 and the `PSTA` algorithm of Chapter 5 to construct the succinct representations of the three bit-vectors. All the algorithm were implemented in C and compiled using GCC 4.9 with optimization level -O2 and using the -ffast-math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

[2]The dataset containing the coordinates was created by MaxMind, available from 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. Our triangulations were generated using the options `-cezCBVPNE`.



(a) Random coordinates. **(b)** Delaunay triangulation. **(c)** Maximal plane graph.

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 demostrate 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. 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

p	rand-1M	rand-2M	worldcities	rand-4M	rand-8M	rand-10M
seq	1.80	4.29	4.89	9.63	20.79	25.82
1	1.35	4.46	3.46	6.63	19.61	17.99
4	0.57	1.23	1.45	2.61	5.59	7.24
8	0.34	0.71	0.82	1.51	3.17	3.94
12	0.26	0.51	0.59	1.07	2.11	2.76
16	0.20	0.41	0.48	0.82	1.69	2.18
20	0.20	0.39	0.37	0.76	1.54	1.78
24	0.17	0.36	0.39	0.67	1.19	1.58
28	0.15	0.37	0.41	0.64	1.24	1.47
32	0.18	0.32	0.34	0.64	1.17	1.51
36	0.17	0.32	0.33	0.54	1.09	1.53
40	0.16	0.31	0.35	0.52	1.04	1.29
44	0.18	0.30	0.32	0.57	1.12	1.14
48	**0.14**	**0.28**	0.33	0.56	1.02	1.30
52	0.15	0.30	**0.30**	**0.46**	0.99	1.19
56	0.26	0.35	0.34	0.47	1.10	1.27
60	0.19	0.36	0.32	0.58	0.99	1.10
64	0.27	0.33	0.34	0.52	**0.92**	**1.06**

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.

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.

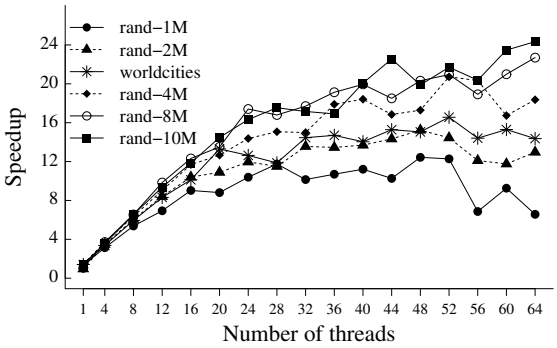


Figure 6.5: Speedup of the `PGEA` and `PSTA` algorithms with all the datasets.

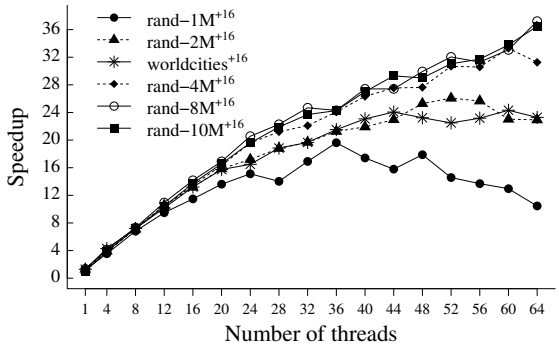


Figure 6.6: Speedup of the `PGEA` and `PSTA` algorithms with all the datasets, artificially increasing the workload with 16 `CAS` operations per edge.

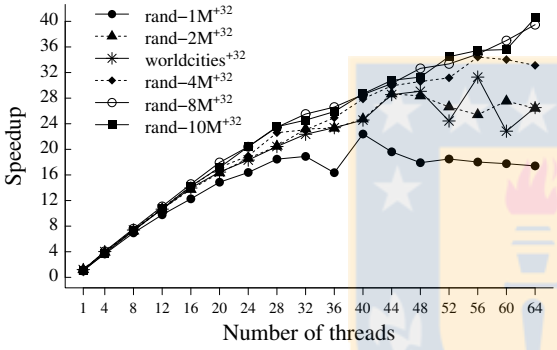


Figure 6.7: Speedup of the `PGEA` and `PSTA` algorithms with all the datasets, artificially increasing the workload with 32 `CAS` operations per edge.

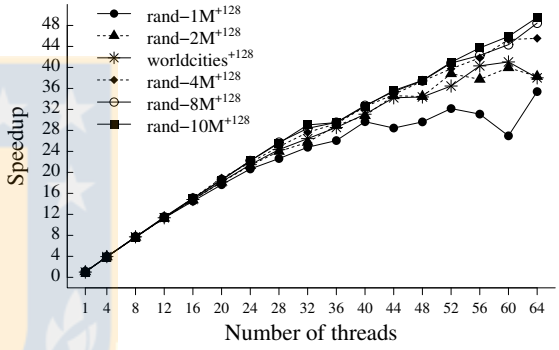


Figure 6.8: Speedup of the `PGEA` and `PSTA` algorithms with all the datasets, artificially increasing the workload with 128 `CAS` operations per edge.

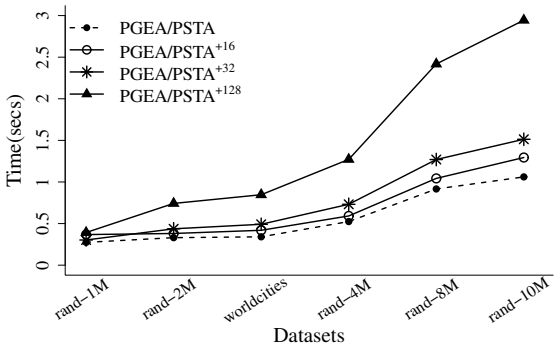


Figure 6.9: Time over the number of vertices (n), with 64 threads. `PGEA/PSTA` corresponds to the execution of `PGEA` and `PSTA` without extra operations. For the other lines, the superscript indicates the number of extra `CAS` operations.

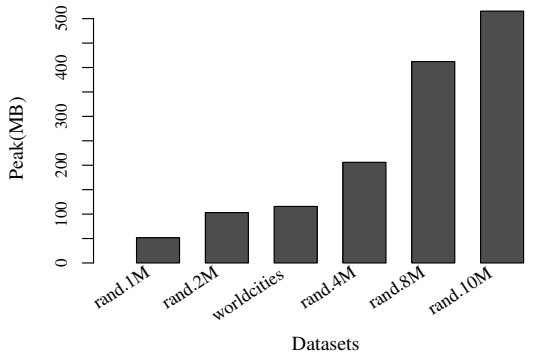


Figure 6.10: Memory consumption sorted by the number of vertices (n).

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_{co}. 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 *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 `PFEA` and `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_e: 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 desgined 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 σ 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 $O(c + \lg(\frac{nc^c}{\lg^c n}))$ 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 $O(c + \lg(\frac{nc^c}{\lg^c n}))$ 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 introducction 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.

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

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

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

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

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

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

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"

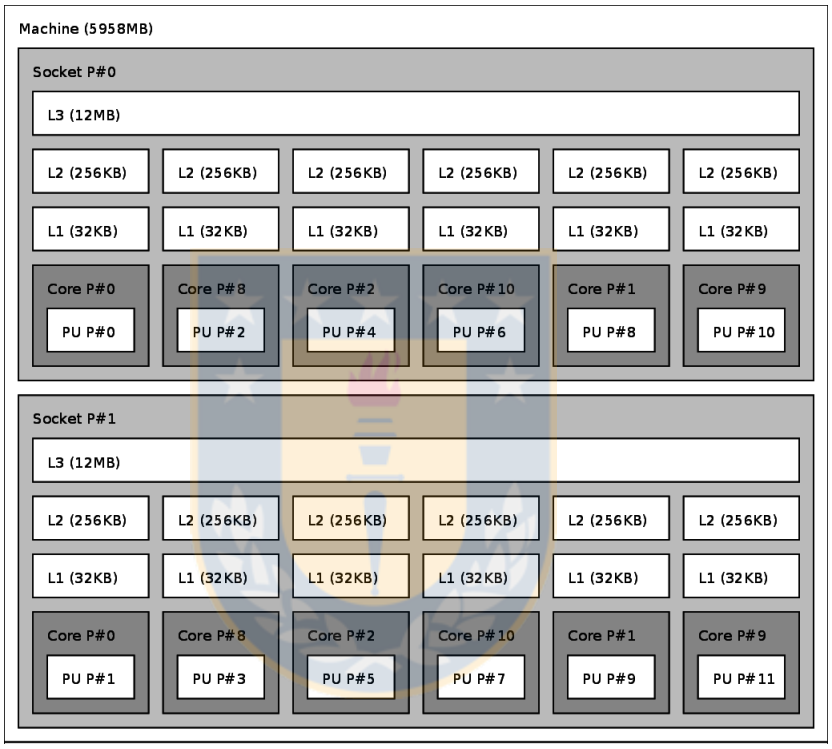


Figure B.1: Topology of `machine A`.

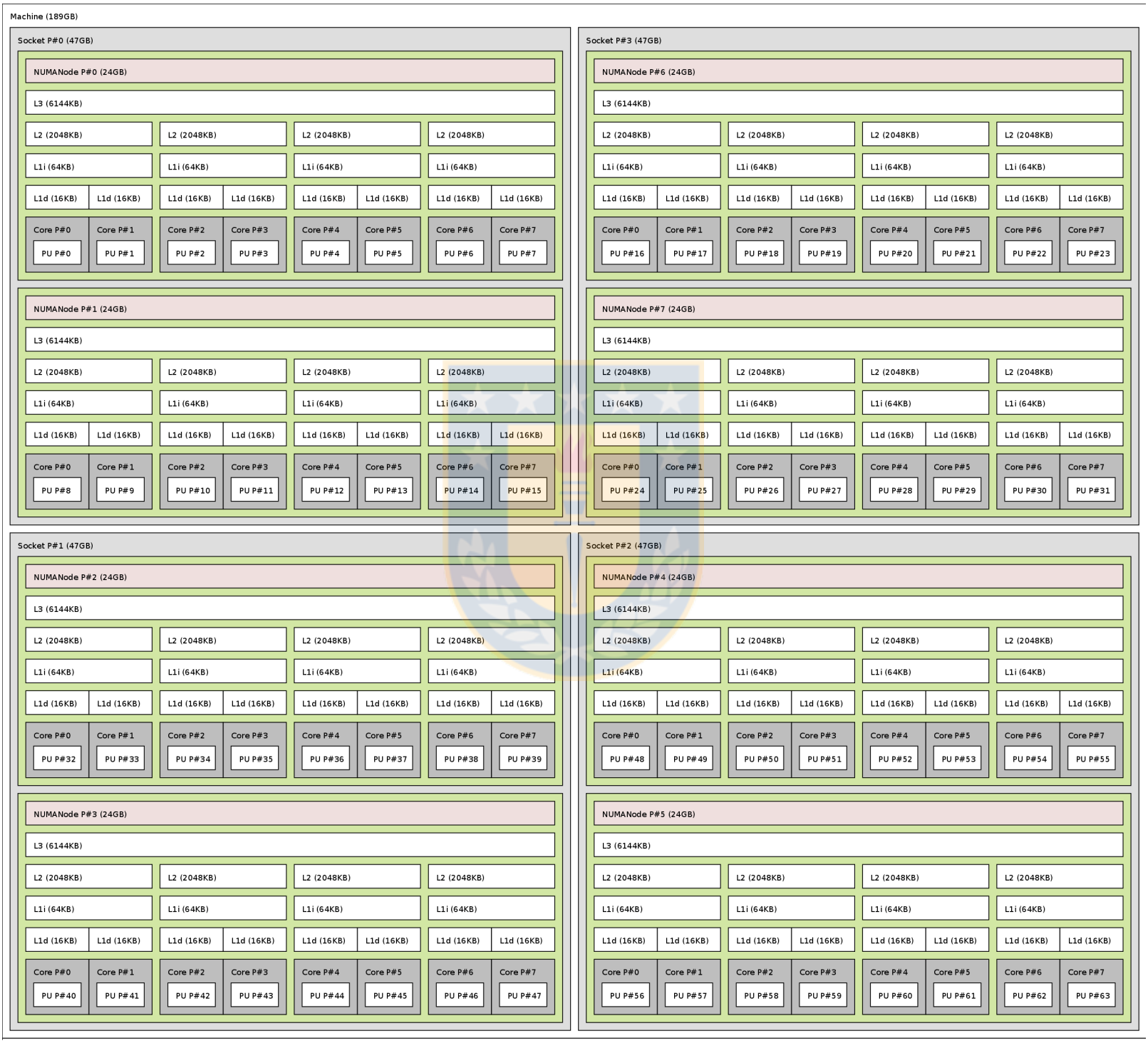


Figure B.2: Topology of `machine B`.

Bibliography

[1] Samy Al Bahra. Nonblocking Algorithms and Scalable Multicore Programming. *Commun. ACM*, 56(7):50–61, July 2013.

[2] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[3] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct Trees in Practice. In *ALENEX*, pages 84–97, 2010.

[4] Diego Arroyuelo, Veronica Gil Costa, Senén González, Mauricio Marín, and Mauricio Oyarzún. Distributed search based on self-indexed compressed text. *Inf. Process. Manag.*, 48(5):819–827, 2012.

[5] Melanie Badent, Ulrik Brandes, and Sabine Cornelsen. More canonical ordering. *Journal of Graph Algorithms and Applications*, 15(1):97–126, 2011.

[6] David A Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, September 2005.

[7] Jrmy Barbay, Luca Castelli Aleardi, Meng He, and J.Ian Munro. Succinct representation of labeled graphs. *Algorithmica*, 62(1-2):224–257, 2012.

[8] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *WADS*, pages 169–180. Springer-Verlag LNCS 1663, 1999.

[9] Timo Bingmann. malloc_count - tools for runtime memory usage analysis and profiling. http://panthema.net/2013/malloc_count/. Last accessed: January 17, 2015.

[10] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, New York, NY, USA, 2011. ACM.

[11] Robert D Blumofe and Charles E Leiserson. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.*, 27(1):202–229, February 1998.

[12] Robert D Blumofe and Charles E Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999.

[13] Hans-J. Boehm and Sarita V Adve. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Not.*, 43(6):68–78, June 2008.

[14] Alex Bowe. *Multiary Wavelet Trees in Practice*. PhD thesis, School of Computer Science and Information Technology, RMIT University, 2010. Honours Thesis.

[15] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. *k2-TreesforCompactWebGraphRepresentation*, pages 18–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] Nieves R Brisaboa, Miguel R. Luaces, Gonzalo Navarro, and Diego Seco. Indexación mediante arrays de sufijos para recuperación de información geográfica. In *Actas del II Congreso Español de Recuperación de Información (CERI)*, 2012.

[17] Nieves R. Brisaboa, Miguel R. Luaces, Gonzalo Navarro, and Diego Seco. Space-efficient representations of rectangle datasets supporting orthogonal range querying. *Inf. Syst.*, 38(5):635–655, 2013.

[18] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications. *SIAM J. Comput.*, 34(4):924–945, April 2005.

[19] Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact Encodings of Planar Graphs via Canonical Orderings and Multiple Parentheses. *CoRR*, cs.DS/0102005, 2001.

[20] RichieChih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In KimG. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 1998.

[21] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *SODA*, pages 383–391, 1996.

[22] David Clark. *Compact PAT trees*. PhD thesis, Cheriton School of Computer Science, Waterloo, Ontario, Canada, 1996.

[23] Francisco Claude. A compressed data structure library. https://github.com/fclaude/libcds. Last accessed: January 17, 2015.

[24] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 176–187, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Francisco Claude and Gonzalo Navarro. The Wavelet Matrix. In Liliana Calderón-Benavides, Cristina González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval*, volume 7608 of *Lecture Notes in Computer Science*, pages 167–179. Springer Berlin Heidelberg, 2012.

[26] Francisco Claude, Patrick K Nicholson, and Diego Seco. Space Efficient Wavelet Tree Construction. In *Proceedings of the 18th International Conference on String Processing and Information Retrieval*, SPIRE'11, pages 185–196, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] Richard Cole and Vijaya Ramachandran. Analysis of Randomized Work Stealing with False Sharing. *Parallel and Distributed Processing Symposium, International*, 0:985–998, 2013.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Multithreaded algorithms. In *Introduction to Algorithms*, pages 772–812. The MIT Press, third edition, 2009.

[29] Beman Dawes and David Abrahams. Boost: C++ libraries. http://www.boost.org/. Last accessed: November 01, 2015.

[30] Hubert de Fraysseix, János Pach, and Richard Pollack. Small sets supporting fary embeddings of planar graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 426–433, New York, NY, USA, 1988. ACM.

[31] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free Dynamically Resizable Arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, OPODIS'06, pages 142–156, Berlin, Heidelberg, 2006. Springer-Verlag.

[32] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. *Engineering the LOUDS Succinct Tree Representation*, pages 134–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[33] Ulrich Drepper. What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf, 2007. Last accessed: May 24, 2014.

[34] James A. Edwards and Uzi Vishkin. Parallel algorithms for burrowswheeler compression and decompression. *Theoretical Computer Science*, 525:10 – 22, 2014. Advances in Stringology.

[35] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[36] Simone Faro and M. Oğuzhan Külekci. Fast multiple string matching using streaming SIMD extensions technology. In *SPIRE*, pages 217–228, Berlin, Heidelberg, 2012. Springer.

[37] Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.

[38] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.

[39] Johannes Fischer and Volker Heun. *A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array*, pages 459–470. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[40] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, April 2011.

[41] Steve Fortune and James Wyllie. Parallelism in Random Access Machines. Technical report, Cornell University, Ithaca, NY 14853, 1978.

[42] Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, 2(4):611–639, October 2006.

[43] Ulrich Fößmeier, Goos Kant, and Michael Kaufmann. *2-Visibility drawings of planar graphs*, pages 155–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[44] H. Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[45] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[46] José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient Wavelet Tree Construction and Querying for Multicore Architectures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 150–161. Springer International Publishing, 2014.

[47] Martin Fürer, Xin He, Ming-Yang Kao, and Balaji Raghavachari. O(nlog log n)-work parallel algorithms for straight-line grid embeddings of planar graphs. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 410–419, New York, NY, USA, 1992. ACM.

[48] T. Gagie, J. I. Gonzlez-Nova, S. Ladra, G. Navarro, and D. Seco. Faster compressed quadtrees. In *2015 Data Compression Conference*, pages 93–102, April 2015.

[49] Travis Gagie, Gonzalo Navarro, and Simon J Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426427(0):25–41, 2012.

[50] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *SODA*, pages 1–10, 2004.

[51] Simon Gog. Succinct data structure library 2.0. https://github.com/simongog/sdsl-lite. Last accessed: January 17, 2015.

[52] Leslie M Goldschlager. A Universal Interconnection Pattern for Parallel Computers. *J. ACM*, 29(4):1073–1086, October 1982.

[53] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 368–373, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics.

[54] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, Greece, 2005. CTI Press. Poster.

[55] Rodrigo González and Gonzalo Navarro. Rank/Select on Dynamic Compressed Sequences and Applications. *Theor. Comput. Sci.*, 410(43):4414–4422, October 2009.

[56] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., New York, NY, USA, 1995.

[57] Roberto Grossi, Ankur Gupta, and Jeffrey Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, Philadelphia, PA, USA, 2003. Soc. Ind. Appl. Math.

[58] Roberto Grossi and Giuseppe Ottaviano. The Wavelet Trie: Maintaining an Indexed Sequence of Strings in Compressed Space. In *Proceedings of the 31st Symposium on Principles of Database Systems*, PODS '12, pages 203–214, New York, NY, USA, 2012. ACM.

[59] D. Harel and M. Sardas. An algorithm for straight-line drawing of planar graphs. *Algorithmica*, 20(2):119–135, 1998.

[60] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[61] Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct ordinal trees based on tree covering. *ACM Trans. Algorithms*, 8(4):42, 2012.

[62] Xin He. On floor-plan of plane graphs. *SIAM Journal on Computing*, 28(6):2150–2167, 1999.

[63] Xin He and Ming-Yang Kao. Parallel construction of canonical ordering and convex drawing of triconnected planar graphs. In K W Ng, P Raghavan, N V Balasubramanian, and F Y L Chin, editors, *Algorithms and Computation*, volume 762 of *Lecture Notes in Computer Science*, pages 303–312. Springer Berlin Heidelberg, 1993.

[64] Xin He, Ming-Yang Kao, and Hsueh-I Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM Journal on Discrete Mathematics*, 12(3):317–325, 1999.

[65] David R. Helman and Joseph JáJá. Prefix computations on symmetric multiprocessors. *J. Par. Dist. Comput.*, 61(2):265 – 278, 2001.

[66] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[67] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[68] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[69] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[70] Maurice P Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[71] Shane V Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

[72] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[73] Facts Hunt. Total number of websites & size of the internet as of 2013. Last accessed: March 04, 2015.

[74] Intel Developer Zone. Transactional synchronization in haswell. https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/. Last accessed: July 24, 2016.

[75] G Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.

[76] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, School of computer science, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.

[77] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[78] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *SODA*, 2007.

[79] Peter Graham Jong Ho Kim Helen Cameron. Lock-Free Red-Black Trees Using CAS. Technical report, University of Manitoba, October 2011.

[80] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[81] Goos Kant. A more compact visibility representation. *International Journal of Computational Geometry & Applications*, 07(03):197–210, 1997.

[82] Roberto Konow and Gonzalo Navarro. Dual-sorted inverted lists in practice. In *SPIRE*, pages 295–306, Berlin, Heidelberg, 2012. Springer.

[83] Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. In *Proceedings of the 2016 Data Compression Conference*, DCC '16, 2016. To appear.

[84] Susana Ladra, Oscar Pedreira, Jose Duato, and Nieves R. Brisaboa. Exploiting SIMD Instructions in Current Processors to Improve Classical String Algorithms. In *ADBIS*, pages 254–267, Berlin, Heidelberg, 2012. Springer.

[85] Thuy T. Le and Jalel Rejeb. A detailed {MPI} communication model for distributed systems. *Future Generation Computer Systems*, 22(3):269 – 278, 2006.

[86] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[87] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[88] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Trans. Algorithms*, 4:28:1–28:13, July 2008.

[89] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, November 2007.

[90] Christos Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012.

[91] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[92] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*, chapter Succinct representation of data structures. Chapman & Hall/CRC, 2004.

[93] Kazuyuki Miura, Machiko Azuma, and Takao Nishizeki. Canonical decomposition, realizer, schnyder labeling and orderly spanning trees of plane graphs. *International Journal of Foundations of Computer Science*, 16(01):117–141, 2005.

[94] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[95] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao S. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74 – 88, 2012.

[96] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *FOCS*, pages 118–126, 1997.

[97] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, March 2002.

[98] J. Ian Munro and S. Srinivasa Rao. *Succinct Representations of Functions*, pages 1006–1015. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[99] J.Ian Munro. Tables. In V Chandru and V Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer Berlin Heidelberg, 1996.

[100] J.Ian Munro, Venkatesh Raman, and S.Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205 – 222, 2001.

[101] Shin-Ichi Nakano. Planar drawings of plane graphs. In *EICE Trans. Fundamentals: Special Issue on Algorithm Engineering*, 2000.

[102] G Navarro and K Sadakane. Fully-Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[103] Gonzalo Navarro. Wavelet Trees for All. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 2–26. Springer Berlin Heidelberg, 2012.

[104] Gonzalo Navarro, Yakov Nekrich, and Luís Russo. Space-efficient data-analysis queries on grids. *Theoret. Comput. Sci.*, 482:60–72, 2013.

[105] Paul Otellini. Keynote speech at intel developer forum, 2003. Last accessed: March 05, 2015.

[106] Mihai Patrascu. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, pages 305–313, Washington, DC, USA, 2008. IEEE Computer Society.

[107] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A Comparison of some recent Task-based Parallel Programming Models. In *3rd Workshop on Programmability Issues for Multi-Core Computers*, Pisa, Italy, January 2010.

[108] Rajeev Raman. *The power of collision: Randomized parallel algorithms for chaining and integer sorting*, pages 161–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[109] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), 2007.

[110] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), November 2007.

[111] Rajeev Raman and S. Srinivasa Rao. Succinct representations of ordinal trees. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 319–332, 2013.

[112] Rajeev Raman and S.Srinivasa Rao. Succinct Representations of Ordinal Trees. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 319–332. Springer Berlin Heidelberg, 2013.

[113] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 225–232, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[114] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, December 2007.

[115] Walter Schnyder. Embedding planar graphs on the grid. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 138–148, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[116] Julian Shun. Parallel wavelet tree construction. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, pages 63–72, Washington, DC, USA, 2015. IEEE Computer Society.

[117] Julian Shun and Guy E. Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Trans. Parallel Comput.*, 1(1):8:1–8:20, October 2014.

[118] Matthias Petri Simon Gog. Optimized succinct data structures for massive data, 2013.

[119] Arwed Starke. Locking in OS Kernels for SMP Systems, 2006.

[120] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software, 2005. Last accessed: March 05, 2015.

[121] German Tischler. On wavelet tree construction. In *CPM*, pages 208–218, Berlin, Heidelberg, 2011. Springer.

[122] Sid-Ahmed-Ali Touati, Julien Worms, and Sbastien Briais. The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426, 2013.

[123] Leslie G Valiant. A Bridging Model for Multi-core Computing. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[124] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *CPM*, volume 4580 of *LNCS*, pages 205–215, Berlin, Heidelberg, 2007. Springer.

[125] Biing-Feng Wang and Gen-Huey Chen. Cost-optimal parallel algorithms for constructing b-trees. *Information Sciences*, 81(1):55 – 72, 1994.

[126] David R. Wood. Degree constrained book embeddings. *Journal of Algorithms*, 45(2):144 – 154, 2002.

[127] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, May 1977.