Parallel Text Index Construction

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund

an der Fakultät Informatik

von

Florian Kurpicz

Dortmund

2020

Tag der mündlichen Prüfung: 26.05.2020

Dekan:	*Prof. Dr.-Ing. Gernot A. Fink* Technische Universität Dortmund Germany
Erster Gutachter:	*Prof. Dr. Johannes Fischer* Technische Universität Dortmund Germany
Zweiter Gutachter:	*Prof. Dr. Simon J. Puglisi* University of Helsinki Finland

ABSTRACT

The focus of this dissertation is the parallel construction of text indices. Text indices provide additional information about a text that allow to answer queries faster. Full-text indices for example are used to efficiently answer phrase queries, i. e., if and where a phrase occurs in a text. The research in this dissertation is focused on but not limited to parallel construction algorithms for text indices in both shared and distributed memory.

In the first part, we look at wavelet trees: a compact index that generalizes rank and select queries from binary alphabets to alphabets of size σ. Using a wavelet tree these queries can be answered in $\mathcal{O}(\lg \sigma)$ time. Being a compact index, the wavelet tree of a text of length n over an alphabet of size σ requires only $n \lg \sigma + o(n \lg \sigma)$ bits of space. We also look at wavelet matrices, which are an alternative representation of wavelet trees and can answer the same queries in the same time while requiring the same space. However, due to a slightly different structure, wavelet matrices are sometimes easier to compute.

Here, our main contribution are the fastest and most memory efficient sequential, shared memory parallel, and semi-external construction algorithms for both wavelet trees and wavelet matrices. Then, we also present the first external memory wavelet tree construction algorithms, which we also parallelize. Huffman-shaped wavelet trees are often used when the text (and wavelet tree) should be compressed. Our sequential and shared memory parallel Huffman-shaped wavelet tree construction algorithms are again the fastest and most memory efficient ones. The corresponding wavelet matrix construction algorithms are of similar speed and the first practical algorithms computing the Huffman-shaped wavelet matrix.

In the second part of this dissertation, we consider the suffix array—one of the most researched text indices. The suffix array of a text contains the starting positions of the text's lexicographically sorted suffixes, i. e., we want to sort all its suffixes. It is often accompanied by the longest common prefix (LCP) array that contains the size of the longest common prefix of two lexicographically consecutive suffixes. There exist over 24 suffix array construction algorithms that work in main memory, and more that work in shared and distributed memory or other models of computation. All these algorithms employ one of three major techniques (or a combination of those): prefix doubling, induced copying, and recursion.

We first conduct a practical evaluation of suffix sorting in main memory, identifying that the fastest algorithms are all based on induced copying and extend the fastest suffix sorting algorithm on most inputs—the DivSufSort—to also compute the LCP array. This gives us a fast LCP array construction algorithm that has a good trade-

off between running time and memory requirements, as the (still) fastest LCP array construction algorithm requires 1.4 times as much memory.

Next, we again focus on parallel algorithms. We present multiple distributed suffix sorting algorithms. First, we implement five suffix sorting algorithms using the distributed batch computation processing framework Thrill. Thrill allows for an easy implementation of distributed algorithms but limits the options to access data. Then, we implement suffix sorting algorithms using the Message Passing Interface (MPI), which is a standardized interface to distribute messages in shared memory. The resulting algorithms are the most memory efficient distributed suffix array construction algorithms and have comparable speed to previously existing algorithms. However, they require less memory than all previously existing distributed suffix sorting algorithms, which allows us to compute suffix arrays for larger inputs than before on the same hardware.

Finally, we use the distributed suffix arrays (and LCP arrays) to compute distributed Patricia tries. This allows us to answer different phrase queries more efficiently than using only the suffix array. In practice, our implementation speeds up existential, counting, and enumeration queries compared to previously existing algorithms that work directly on the suffix array. We also present succinct versions of the distributed Patricia trie.

ZUSAMMENFASSUNG (IN GERMAN)

In dieser Dissertation betrachten wir die parallele Konstruktion von Text-Indizes. Text-Indizes stellen Zusatzinformationen über Texte bereit, die Anfragen hinsichtlich dieser Texte beschleunigen können. Ein Beispiel hierfür sind Volltext-Indizes, welche für eine effiziente Phrasensuche genutzt werden, also etwa für die Frage, ob eine Phrase in einem Text vorkommt oder nicht. Diese Dissertation befasst sich hauptsächlich, aber nicht ausschließlich mit der parallelen Konstruktion von Text-Indizes im geteilten und verteilten Speicher.

Im ersten Teil der Dissertation betrachten wir Wavelet-Trees. Dabei handelt es sich um kompakte Indizes, welche Rank- und Select-Anfragen von binären Alphabeten auf Alphabete der Größe σ verallgemeinern. Ein Wavelet-Tree kann diese Anfragen dann in $\mathcal{O}(\lg \sigma)$ Zeit beantworten. Der Wavelet-Tree gilt als kompakt, da er für einen Text der Länge n über einem Alphabet der Größe σ lediglich $n \lg \sigma + o(nlg\sigma)$ Bits Platz benötigt. Zusätzlich zu den Wavelet-Trees betrachten wir auch die Wavelet-Matrix. Diese ist eine alternative Darstellung des Wavelet-Trees, kann die gleichen Anfragen in gleicher Zeit beantworten und benötigt den gleichen Platz. Allerdings ist die Konstruktion einer Wavelet-Matrix für bestimmte Eingaben einfacher.

In dieser Dissertation stellen wir zudem die schnellsten und speichersparsamsten sequenziellen, (im geteilten Speicher) parallelen und semi-externen Speicher-Konstruktionsalgorithmen für Wavelet-Trees und Wavelet-Matrizen vor. Außerdem beschreiben wir die ersten Wavelet-Tree- und Wavelet-Matrix-Konstruktionsalgorithmen für den externen Speicher. Diese Algorithmen parallelisieren wir zudem.

Huffman-shaped Wavelet-Trees werden oft genutzt, wenn der Text komprimiert werden soll. Hier erhalten wir einen Wavelet-Tree über den komprimierten Text. Unsere sequenziellen und (im geteilten Speicher) parallelen Konstruktionsalgorithmen für Huffman-shaped Wavelet-Trees sind dabei die schnellsten und speichersparsamsten. Die daraus resultierenden Huffman-shaped Wavelet-Matrix-Konstruktionsalgorithmen haben eine ähnliche praktische Laufzeit und den gleichen Speicherbedarf. Sie sind dabei die ersten Huffman-shaped Wavelet-Matrix-Konstruktionsalgorithmen.

Im zweiten Teil der Dissertation betrachten wir das Suffix-Array, den am besten erforschten Text-Index überhaupt. Das Suffix-Array enthält die Startpositionen aller lexikografisch sortierten Suffixe eines Textes, d. h., wir möchten alle Suffixe eines Textes sortieren. Oft wird das Suffix-Array um das Longest-Common-Prefix-Array (LCP-Array) erweitert. Das LCP-Array enthält die Länge der längsten gemeinsamen

Präfixe zweier lexikografisch konsekutiven Suffixe.

Es gibt über 24 verschiedene Suffix-Array-Konstruktionslagorithmen: sequenzielle, (im geteilten und verteilten Speicher) parallele und solche in weiteren Modellen. All diese Algorithmen bauen auf einer (oder einer Kombination) von drei verschiedenen Techniken auf: Prefix Doubling, Induced Copying und Rekursion. Zunächst führen wir eine praktische Evaluation von Suffix-Sortier-Algorithmen im Hauptspeicher durch. Hierbei stellen wir fest, dass die schnellsten Algorithmen auf der Technik des Induced Copying basieren. Anschließend erweitern wir den schnellsten (basierend auf den meisten Eingaben) Algorithmus – den DivSufSort –, sodass zusätzlich das LCP-Array konstruiert wird. Der daraus resultierende Algorithmus ist ein schneller LCP-Array-Konstruktionsalgorithmus, der eine gute Balance zischen Laufzeit und Speicherbedarf aufweist, da der bisher immer noch schnellste Algorithmus 1,4-mal soviel Speicher benötigt.

Als Nächstes richten wir unseren Fokus wieder auf parallele Algorithmen und betrachten verschiedene (im verteilten Speicher) parallele Suffix-Sortier-Algorithmen. Wir implementieren fünf solcher Algorithmen in dem verteilten Batch-Verarbeitungs-Framework Thrill. Dieses ermöglicht eine einfach und schnelle Implementierung verteilter Algorithmen, limitiert jedoch die Möglichkeiten, wie man auf Daten zugreifen kann. Um die Resultate besser einordnen zu können, implementieren wir diese Algorithmen auch mit dem Message Passing Interface (MPI). Hierbei handelt es sich um eine standardisierte Schnittstelle zum Nachrichtenaustausch im verteilten Speicher. Diese Algorithmen sind die speichersparsamsten verteilten Suffix-Array-Konstruktionsalgorithmen, die das Suffix-Array ähnlich schnell wie bereits existierende Algorithmen konstruieren. Durch die Speichersparsamkeit können wir das Suffix-Array auf der gleichen Hardware dabei für deutlich größere Eingaben als zuvor konstruieren.

Abschließend nutzen wir verteilte Suffix- und LCP-Arrays, um den Distributed-Patricia-Trie zu konstruieren. Dieser erlaubt es uns, verschiedene Phrase-Anfragen effizienter zu beantworten, als wenn wir nur das Suffix-Array nutzen. In der Praxis beschleunigt unsere Implementierung somit Existential-, Counting- und Enumeration-Anfragen im Vergleich zur Beantwortung direkt auf dem Suffix-Array. Zuletzt stellen wir noch Varianten des Distributed-Patricia-Trie vor, die auf platzeffizienten Datenstrukturen aufbauen.

CONTENTS

CHAPTER 1
INTRODUCTION

Stringology is a field of computer science that focuses on algorithms and data structures for texts. In this dissertation, we look at data-structures for texts that structure the inputs such that it can easily be searched—so called *text indices*. These text indices allow us to answer different queries on the text more efficiently.

Let us start with a simple and well-known example of a text index. If we want to list all occurrences of the word "efficient" in this dissertation, we have to read it at least once. However, if we have an index that lists all occurrences of all words in this dissertation, we just have to search for "efficient" in the index, without having to read the whole text. Furthermore, if all words in the index are sorted lexicographically, we can find the word faster, because we know where the word occurs relatively to all other words in the index. Similar to this example, we want to construct different text indices that allow us to speed up different queries. For example, we are not only interested in querying for simple words but also in phrases. Some text indices considered in this dissertation can answer queries asking for phrases, e. g., when we look for the phrase "efficient parallel construction."

An efficient construction of text indices is important, as the amount of data created on a daily basis is ever increasing and texts are on the forefront of this data flood, as the world wide web, digital encyclopedias, and biological data like DNA and proteins are all represented as textual data. In this dissertation we focus on the parallel construction of text indices. Parallel computation of text indices allows us to gain considerable speedups in computation speed even if the speed of single CPU cores does not increase further.

We consider two different parallel models in this dissertation: shared memory (multiple CPU cores that all share the same main memory) and distributed memory (using multiple CPUs that all have their own main memory and are connected via a network). In these parallel settings, "efficient" has multiple meanings. We are mostly interested in the following ones. First of all, our algorithms have to be fast. Even though we use potent hardware that provides lots of CPU cores we are interested in algorithms that are fast even when we only use a single CPU core. Then, our algorithm have to scale, i. e., if we double the number of CPU cores used we (ideally) would like to double the throughput, because this helps to predict the behavior of the algorithm when we use even more CPU cores in the future. Finally, our algorithms should be memory efficient as this allows us to process more input on the same hardware if we are limited by the size of the main memory.

As mentioned before, different text indices may be useful for different queries. Therefore, we do not only look at a single text index, but at different ones that we introduce in the next section. However, the main focus of this dissertation are not the queries, but their construction. While we conduct some experiments for answering queries at the end, we are mainly interested in the efficient construction.

A Short Note on this Dissertation. This document was created with its printed version in mind. Hence, it is best viewed in color and with facing pages, as most figures are placed accordingly. Our citation policy is as follows: We cite the most recent version of all papers, i. e., if there is a more recent journal version of a paper previously published at a conference, we cite the journal version.

1.1 OUR CONTRIBUTIONS

In this dissertation, we look at three different text indices—wavelet trees, suffix arrays, and Patricia tries—and present efficient construction algorithms for all three of them. As mentioned before, we focus on the parallel construction in shared and distributed memory and provide construction algorithm that scale well. Another central point of our construction algorithm are their low memory requirements, which allows us to process larger inputs on the same hardware.

The structure of this dissertation follows these three data structures, and is split into two parts. In the first part, we consider shared memory construction (with a quick side-trip to external memory) of wavelet trees. In the second part, we first look at main memory suffix sorting and longest common prefix array computation before we present different distributed memory suffix sorting algorithms. Then, we use the distributed suffix arrays to construct distributed Patricia tries.

On Wavelet Trees

In Part I, we take a detailed look at wavelet trees [Gro+03] and wavelet matrices [Cla+15], which are space efficient indices that allow us (among others) to generalize rank and select queries from the binary alphabet to alphabets of arbitrary size. Our work focuses on the efficient construction of wavelet trees and wavelet matrices—we do not answer queries using them.

There exists a a lot of work regarding the construction of wavelet trees, which we summarize with a focus on practical results in Chapter 2. We also give a detailed insight on the structure of wavelet trees and wavelet matrices in Section 2.4, focusing on the transformation from the former to the latter.

For the construction of wavelet trees and wavelet matrices, we first present a novel technique—the *bottom-up* construction—that we employ in all our algorithms. See Section 3.1. We use this technique in multiple models of computation and create the fastest and most memory efficient wavelet tree and wavelet matrix construction algorithms in practice in the following models of computation: (i) in Section 3.2, we consider sequential algorithms in the Word RAM model, (ii) in Section 3.3, we show

how we can extend the algorithms to work in shared memory, and finally (iii) in Section 3.4, we extend our algorithms to work in semi-external and external memory. In addition, we also show how to efficiently compute Huffman-shaped wavelet trees and wavelet matrices, i. e., wavelet trees and wavelet matrices for a Huffman encoded text in Section 3.5.

On Suffix Arrays

In Part II, we focus on the suffix array [Gon+92; MM93]. The task of constructing the suffix array translates to sorting all suffixes in lexicographical order. To this end, we first examine the practicality of different suffix sorting algorithms in main memory. There exists a plethora of different algorithms that can be classified into three different types (and hybrids) of algorithms. Our first result (Chapter 4) is the first practical survey of all implemented suffix sorting algorithms. We identify the fastest suffix sorting algorithm (in main memory) and extend it, such that it also computes the longest common prefix (LCP) array. The LCP array often accompanies the suffix array to speed up the search in the suffix array, which results in the fastest algorithm that computes both the suffix array and the LCP array. However, computing the LCP array using the suffix array is faster than computing both at the same time.

Next, in Chapter 5, we tackle distributed memory suffix sorting. We first implement two different suffix sorting algorithms in two and three different variants (for a total of five different algorithms) using the big data batch computation framework Thrill. Since Thrill imposes some limitations on the algorithms, we use the Message Passing Interface (MPI) to implement further distributed suffix sorting algorithms that have a similar throughput compared with existing algorithms but require less memory. Among them is the first induced copying suffix sorting algorithm in distributed memory. While not the fastest, this is the most memory efficient distributed suffix sorting algorithms, which allows us to handle much larger inputs on the same hardware as before.

On Patricia Tries

Finally, in Chapter 6, we present the distributed Patricia trie—a distributed version of the Patricia trie [Mor68]. We use the suffix array and LCP array to compute a compact index that speeds up querying the suffix array. Our distributed Patricia trie supports three different types of queries (existential, counting, and enumeration queries). It proves to be superior to querying the suffix array directly, as queries can be distributed easier and can often be answered locally without additional communication. In this chapter, we are, for the first time in this dissertation, interested in the speed we can answer queries in. This is the first sophisticated distributed index build on top of the distributed suffix array. Note that distributed indices that use the suffix array (or a permutation of the suffix array) to answer queries exist.

In addition, we present different succinct implementations of the distributed Patricia trie. To this end, we use three different tree representations that represent a tree consisting of n nodes in $2n + o(n)$ bits. The lower order term allows us to navigate in these succinct tree representation.

1.2 BASIC NOTATIONS

Most of the notations we use in this dissertation are well established in the Stringology community. Nevertheless, we briefly introduce notations we use in this dissertation in this section. Notations that we only need in specific parts are introduced right before needed. We use the *binary* logarithm and denote it by lg.

Throughout this dissertation, a text $T = [\,T[0], T[1], \ldots, T[n-1]\,]$ consists of n characters from a totally ordered alphabet Σ of size $\sigma := |\Sigma|$. Let $[\,0, n\,] := \{0, \ldots, n\}$ and $[0, n) := \{0, \ldots, n-1\}$ be ranges of integers. For any array A, we write $A[i, j]$ (or $A[i, j)$) to denote the sub-array of A ranging from i to j (or $j-1$). The sub-array is empty if $i > j$ (or $i \geq j$). In the case of texts, we call $T[i, j]$ and $T[i, j)$ *substrings* of T, $T[0..i)$ is the i-th prefix of T, and $T[j..n)$ is the j-th *suffix* of T for $i, j \in [0, n)$. We denote the empty substring by ϵ.

We often refer to the *histogram* Hist_T of a text T. We simply write Hist without indicating the corresponding text, if it is clear from context. Let T be a text of length n over an alphabet of size σ. Here, a histogram is an array $\mathsf{Hist}[0, \sigma)$ with $\mathsf{Hist}[i] = |\{j \in [0, n) \colon T[j] = i\}|$ for all $i \in [0, \sigma)$. Unless necessary, we do not explicitly mention the corresponding text.

Often, a text T does not contain all characters of the alphabet $\Sigma = [0, \sigma)$. In this case, we can make use of the *effective alphabet* that is a mapping of all σ' characters that actually occur in the text to a new alphabet $\Sigma' = [0, \sigma')$ that preserves the lexicographical order. To be more precise, we use the following mapping: $\Sigma'[i] = \Sigma[j]$, such that $\mathsf{Hist}[j] > 0$ and $|\{k \in [0, j) \in \colon \mathsf{Hist}[k] > 0\}| = i$, i. e., the $(i+1)$-th character (with respect to their rank) that occurs at least once in the text.

Some explanations of algorithms are enhanced using pseudo code. To this end, we require some notations that we use throughout this dissertation. First, we assume that all arrays are allocated with the right size, e. g., if we have an array that we use to store the histogram of a text over an alphabet of size σ, the array has σ positions. Initially, all array positions contain only zeros. Then, we use the equal sign ($=$) for both: comparison and assignment of values. Our syntax is close to real world programming languages, e. g., when we use $j = \mathsf{A}[i]{+}{+}$ we first get the value stored in $\mathsf{A}[i]$ and then we increase the value (stored in the array) by one.

1.3 CONSIDERED MACHINE MODELS

Whenever we analyze the running time, our considered model of computation is word RAM (random access machine) model [Hag98] with (computer) word size $w = \Omega(\lg n)$ for inputs of size n. Here, we can access a computer word in $\mathcal{O}(1)$ time. In the word RAM model, we represent texts as arrays where each character occupies one word. However, given a string over an alphabet of size σ, we only require $\lceil \lg \sigma \rceil$ bits to represent a character. This allows us to store $\lfloor w / \lceil \lg \sigma \rceil \rfloor$ characters per (computer) word. This technique is often called *word packing*.

Also, there are operations on computer words that can be computed in constant time, i. e., bitwise operations like bitwise *and*, bitwise *or*, bitwise *shift* (left and right), and access to any bit. We refer to Knuth [Knu14, p. 134ff] for a detailed list of operations.

Note that modern hardware supports even more complex operations on computer words, which we introduce whenever needed. In practice, the maximum computer word size is usually 64 bits. Admittedly, newer (high performance computing) hardware supports computer word sizes up to 512 bits, e. g., using the AVX-512 instruction set.

For our parallel algorithms, we need more sophisticated machine models to analyze running times and other costs of the algorithms. In those models, we have multiple *processing elements* (PEs) that execute algorithms in parallel. Processing element is an abstract notion that can mean different things, depending on the setting. Now, we describe different models and mention what a processing element refers to.

1.3.1 Parallel Random Access Memory Model

The first parallel model we consider is the *shared memory* parallel model. Here, all processing elements have a shared memory that they can use to communicate, by writing to designated memory addresses in the shared memory. See Figure 1.1 for a visualization. In practice, a processing element on a shared memory machine refers to a CPU core or a thread that executes an algorithm, the shared memory refers to main memory, and the local memory is the cache of the CPU.

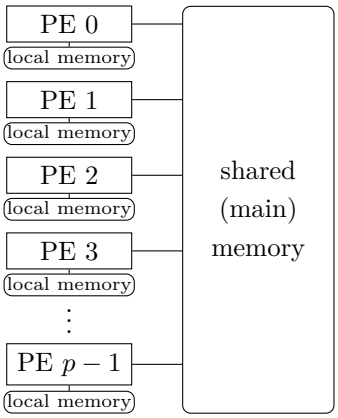


Figure 1.1. Shared memory model, where each PE has access to a local memory that can only be accessed by the corresponding PE, and all PEs can communicate over the shared memory.

We analyze parallel shared memory algorithms using JáJá's work-time model [JáJ92] for *parallel random access machines*, where we use two parameters *work* and *time* (sometimes also called *span*) to measure the performance. For any parallel algorithm the work is the total number of operations used by the algorithm, i. e., its sequential running time. The time, on the other hand, is the number of time units that are needed to execute the algorithm, when operations can be executed in parallel by different processing elements of which we have unlimited many.

On parallel random access machines, we have to consider *race conditions* and *false sharing*. Race conditions occur when the result of an algorithm depends on the timely order of processing elements executing their operations. For example, let two processing elements increase the variable *occ* which initially is 0: the first processing element increases it by 2 and second processing element increases it by 3. Now, we cannot be sure what value *occ* has afterwards; it could be 2, 3, or 5. To avoid such undefined behavior in practice we can use semaphores or mutexes, which regulate the access to memory and guarantee that only one processing element writes to the same memory position at a time. Since those have an unwanted running time overhead, we have to develop our algorithms such that race conditions do not occur by design.

False sharing occurs when two or more processing elements write to different memory positions that are in the same cache line. Then, all other processing elements that hold that cache line are forced to reload it, which results in additional and unnecessary memory access (but not undefined behavior like race conditions) and should be avoided.

1.3.2 Distributed Memory Model

In the distributed memory model, algorithms run on p distinct processing elements that are connected by a network, e. g., are distributed in a cluster on different physical hardware, such as CPUs or CPU-cores in different (compute) nodes, see Figure 1.2. In practice, the number of processing elements in this setting is often orders of magnitude greater than the number of processing elements in shared memory, because we can use multiple shared memory machines that are connected by a network. Each processing element has a unique *rank* in the range from 0 to $p - 1$. The processing elements all have access to a local memory. In a cluster environment, this local memory is usually the main memory. Here, it is possible that different processing elements communicate over the local memory. We do not consider this *hybrid* of distributed and shared memory explicitly. However, in Chapter 5, we develop algorithms in the Thrill framework that automatically makes use of the hybrid approach.

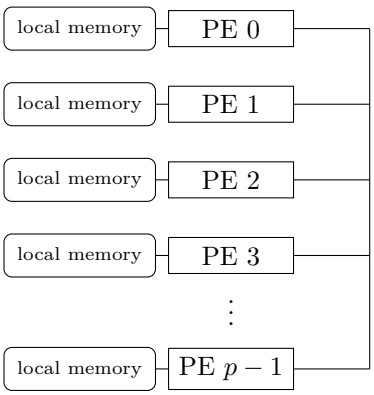


Figure 1.2. Distributed memory mode. Compared with the shared memory model (see Figure 1.1), the local memory is usually much larger and PEs can only communicate over a network.

We analyze our algorithms in the *bulk-synchronous parallel* (BSP) model [Val90]. Here, each algorithm is a sequence of *supersteps*, with each superstep being split into three phases: First, the processing elements can perform any number of operations based on local data. We use w to denote the maximum time used by a processing element. Second, the processing elements can send data to other processing elements (communication phase). Here, h is the maximum number of machine words communicated by any processing element, and G is the running time required for the communication of one word. Last, all processing elements wait until every processing element has finished the first two phases. L is the time of this barrier synchronization. There is no synchronization between the first and second phase. Processing elements can start communicating as soon as they have finished working on the local data (but data received during the communication is not available for local operations before the next barrier synchronization), see Figure 1.3. The total running time of a BSP-algorithm is the time of all its supersteps, where the time of one superstep is $w + hG + L$.

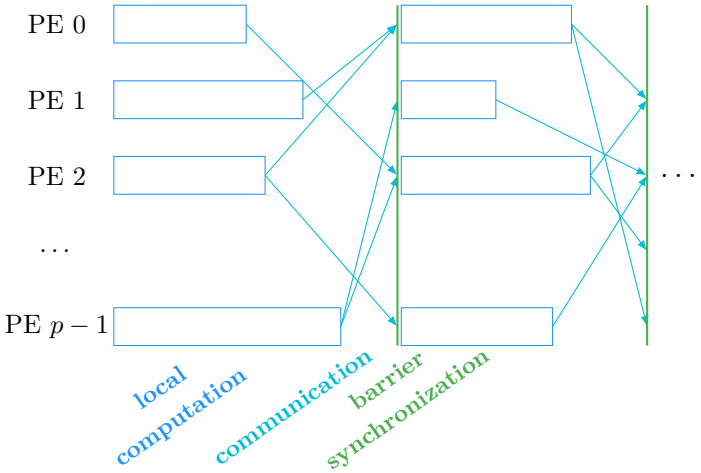


Figure 1.3. Two supersteps in the BSP model. Local computation is depicted in blue (), communication is highlighted in cyan (), and synchronization barriers are shown in green ().

1.3.3 COST of Parallelization

In addition to the models described above, we also consider the practical cost of parallelization by McSherry et al. [McS+15]. Here, the Configuration that Outperforms a Single Thread (*COST*) of the parallel algorithm is the number of processing elements that are required to process the input faster than the fastest sequential algorithm for the same problem on the same hardware. The general idea behind this measurement is to identify the overhead of the parallel implementations, as often poor baselines are the reason for reported good speedups. Hence, the lower the COST the better; the best COST is 2, as that is the least amount of processing elements needed for a parallel algorithm to be faster than any sequential algorithm on the same hardware.

The COST is similar to the *cost* (lower case) and *speedup* defined by Casanova et al. [Cas+08, page 10]. They say the *cost* is the running time of a parallel algorithm using p processing elements times p. A low cost also correlates with a well-scaling parallel algorithm, similar to the COST. However, the cost does not consider the fastest sequential algorithm and is solely based on the scalability of the parallel algorithm. The *speedup* is the running time of the fastest sequential algorithm divided by the running time of the fastest parallel algorithm using p processing elements. Often the speedup is defined with the parallel algorithm using only one processing element instead of the fastest sequential one. Since we are using the fastest sequential one, we can give the relationship between speedup [Cas+08] and COST [McS+15] of a parallel algorithm: The smallest number of processing elements that results in a speedup greater than 1 is the COST of the algorithm, because using this many processing elements results in a parallel algorithm faster than the fastest sequential algorithm.

1.3.4 External Memory Model

The *external memory model* [AV88] measures the transfer of data between the main memory of size M (also called *local memory*) and a secondary memory (also called *external memory*) that is assumed to be of unlimited size and slower in terms of memory access than the main memory. Also, data can only be transferred in *blocks* of size B between main and secondary memory. Transfers of blocks are called *I/O operations* (*I/Os* for short) and are the main cost measure of the external memory model. External memory algorithms are often analyzed using the I/Os of common operations. Later, in Section 3.4, we make use of the *scan* operation. In the external memory model, scanning N elements requires $\text{scan}(N) = \Theta(N/B)$ I/Os.

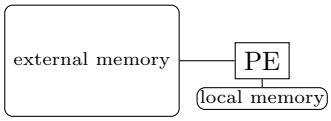


Figure 1.4. In the external memory model, data in external memory can only be accessed in blocks of size B.

For *semi-external* algorithms, we assume that we have random access on either the input or output—but not both, as then we would have an algorithm working in main memory. This relaxation allows for algorithms that cannot easily be expressed in the external memory model (due to expensive random access on either input or output). The model is used in practice, e. g., the succinct data structure library (SDSL) [Gog+14a] provides semi-external construction algorithms for many string data structures.

1.4 EXPERIMENTAL SETUP

Now, we give a detailed description of the hardware used for our experiments in Section 1.4.1, and describe the texts that we use as inputs and where to get them from in Section 1.4.2. Since we published the source code (we give links to the code whenever we evaluate our algorithms) of all our implementations, our results can be verified by the reader.

1.4.1 Hardware

We conducted most of our experiments on nodes of a cluster, which run CentOS 7.6 as operating system. There, we have access to two types of nodes:

LiDO.small nodes are equipped with 64 GB RAM and two Intel Xeon E5-2640v4 CPUs. Each CPU has 10 cores at 2.4 GHz base frequency (3.4 GHz maximum turbo frequency) and cache sizes: 32 KB L1D and L1I, 256 KB L2, 25.6 MB L3. Both L1 and the L2 caches are private, the L3 cache is shared.

LiDO.big nodes are equipped with 256 GB RAM and four Intel Xeon E5-4640v4 CPUs. Each CPU has 12 cores at 2.1 GHz base frequency (2.6 GHz maximum turbo frequency) and cache sizes: 32 KB L1D and L1I, 256 KB L2, 30 MB L3. Both L1 and the L2 caches are private, the L3 cache is shared

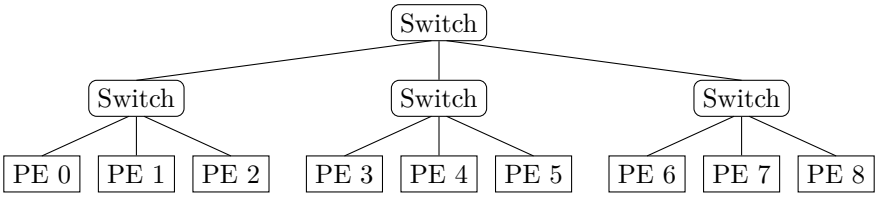


Figure 1.5. Nine processing elements that are connected 1:3 blocking.

Note that Hyperthreading is disabled on all nodes in the cluster by default and cannot be enabled by us; we only use physical cores, of which there are 20 (LiDO.small) or 48 (LiDO.big). When used in the distributed memory setting, all nodes are connected via Interconnect Infiniband QDR with a transfer rates of up to 40 GBit/s and that are connected at most 1:3 blocking, i. e., the switches are connected in a tree structure where each inner node has three children, see Figure 1.5.

For external memory our experiments (which we present in Section 3.4), we used a machine equipped with 16 GiB RAM and one Intel Xeon i7-6800K CPU (6 cores at frequencies up to 3.4 GHz and cache sizes: 32 kB L1D and L1I, 256kB L2, and 15360 kB L3). The operating system is Ubuntu 16.04 (64-bit, Linux kernel 4.4). Our external memory algorithms use the STXXL [Dem+08b] development snapshot (26-09-2017). In our external memory experiments, we consider two different settings:

Ext.hdd eight Hitachi HUA72302 HDDs each with a capacity of 1.8 TiB, or

Ext.ssd two Samsung SSD 850 EVO SSDs each with a capacity of 465.8 GiB.

1.4.2 Text Corpora

One of the most used text corpora is the *Pizza & Chili* corpus, which is available at `http://http://pizzachili.dcc.uchile.cl`. Unfortunately, the texts in it are relatively small; all files are smaller than 2.5 GiB, which is too small for most of our experiments. Since we want to use the same texts for all of our experiments in this dissertation we obtained different real world texts that have a wide variety of different properties that we look at later. Next, we give precise information on the used texts and where to obtain them. Hereafter, whenever we use a smaller text, we use a prefix of the required size of the text. We want to mention that we use the Pizza & Chili corpus for our main memory experiments in Chapter 4. Elsewhere, we make use of the following four texts.

Common Crawl. The *Common Crawl* corpus contains websites that are crawled by the Common Crawl Project. We use the *WET* files, which contain only the textual data of the crawled websites, i. e., no HTML tags. We also removed the meta information added by the Commoncrawl corpus. To be more precise, we used the following WET files: `crawl-data/CC-MAIN-2019-09/segments/1550247479101.`

30/wet/CC-MAIN-20190215183319-20190215205319-#ID.warc.wet, where #ID is in the range from 00000 to 000600. As we only care for the text, we removed the WARC meta information, i. e., each line consisting of WARC/1.0 and the following eight lines. CommonCrawl is the concatenation of all files sorted in ascending order by their ID.

DNA. We obtained our DNA data sets from the *1000 Genomes Project*. Here, we extract the DNA data from FASTQ files. A FASTQ file contains four lines for each sequence, where the second line contains the raw sequence. Since we are only interested in the raw sequence, we discarded all other lines and also cleaned the second line, such that it only contains the characters A, C, G, and T. (We simply removed all other characters.) Note that our final file consists of a single line as we want the size of the alphabet to be four. The FASTQ files are available at ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000/DRR#ID, where #ID is in the range from 000001 to 000426_1. Note that the #IDs are not continuous; not all #IDs are assigned, and some #IDs are separated in two parts, which is denoted by an _1 and _2 suffix. Throughout this dissertation, we denote this DNA sequence by DNA.

Proteins. *UniProt* (Universal Protein Resource) is a project that makes protein sequences and annotation data available. The UniProt Knowledgebase is a collection of information on proteins and their sequences. It consists of a reviewed (Swiss-Prot) and an unreviewed (TrEMBL) part. Since we are only interested in the sequences, we concatenated the files available at ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/uniprot_#ID.dat.gz. Here, #ID is either *sprot* or *trembl*. We concatenated the files in that order (first sprot then trembl), and removed all lines not containing sequences and all white spaces. We denote the protein sequences by Prot.

Wikipedia. The *Wikipedia* is an online encyclopedia available in multiple languages that makes all its (textual) content available for download. For our experiments, we used the XML data of all pages in the most current version *only*, i. e., the files available at https://dumps.wikimedia.org/#IDwiki/20190320/#IDwiki-20190320-pages-meta-current.xml.bz2, where #ID is *de*, *en*, *es*, and *fr*. We concatenated the files in the same order (first *de* then *en*, *es*, and *fr*). Throughout this dissertation, we denote the Wikipedia dump by Wiki.

Detailed Information About the Texts

Since the structure of texts can be important for the behavior of our algorithms, we now give detailed information about the texts described above. We give some commonly used characteristics in Table 1.1. Here, we see the sizes of the alphabets and the total size of the texts. Since we mostly consider byte alphabets, i. e., alphabets of size at most 256, we use four real world text with different alphabet sizes. All information about the text has been computed on the whole text. While we often use prefixes of the texts as input, this still gives a good overview of the differences of the text.

Table 1.1. Characteristics of the texts used in this dissertation: name of the text, alphabet size σ, total text size n, and empirical entropy H_k for $k \in [0, 3]$.

Name	σ	n	H_0	H_1	H_2	H_3
`CommonCrawl`	243	196,885,192,752	6.19	4.49	2.52	2.08
`DNA`	4	218,281,833,486	1.99	1.97	1.96	1.95
`Prot`	26	50,143,206,617	4.21	4.20	4.19	4.17
`Wiki`	213	246,327,201,088	5.38	4.15	3.05	2.33

Now, we want to look at slightly more advanced measures for the complexity of a text. To this end, we need additional notations. Let T be a text of length n over an alphabet $\Sigma = [0, \sigma)$. For any $k \in \mathbb{N}$, Σ^k denotes all possible strings of length k that can be constructed using any characters from Σ. Given an $S \in \Sigma^k$, we say T_S is the concatenation of all characters α in T in text order for which the substring $S\alpha$ exists. For example, if $T = [1, 2, 3, 4, 1, 2, 3, 5, 1, 2, 3, 4]$, then $T_{123} = [4, 5, 4]$.

This allows us to define the *empirical entropy*, which has been initially introduced by Kosaraju and Manzini [KM99] to better analyze Lempel-Ziv (LZ) compression algorithms. However, LZ compression algorithms also exploit repetitions that are far apart, which is not measured by the empirical entropy.

Let Hist be the histogram of the characters in T, then the 0-th empirical entropy is $H_0(T) := (1/n) \sum_{i=0}^{\sigma-1} \mathsf{Hist}[i] \lg(n/\mathsf{Hist}[i])$. The k-th empirical entropy is $H_k := (1/n) \sum_{S \in \Sigma^k} |T_S| \cdot H_0(T_S)$. In other words, the k-th empirical entropy gives us an asymptotic lower bound of the number of bits that we need to express a character when considering it preceding k characters.

We have a special interest in H_0, because our Huffman-shaped wavelet trees and wavelet matrices that we present in Section 3.5 achieves the 0-th order empirical entropy limit, i. e., it results in an optimal compression when compressing character by character. We are also interested in H_1, since our distributed suffix array construction algorithm that we present in Section 5.4 behaves differently, depending on two consecutive characters in the text, which corresponds to the 1-st order empirical entropy.

Finally, let us take a look at the histograms of the texts that we depict in Figure 1.6. Here, we can see that `CommonCrawl` and `Wiki`, and `DNA` and `Prot`, have a similar structure, respectively. Both `CommonCrawl` and `Wiki` contain natural language, hence characters that are part of the Latin alphabet occur more often. The main difference between the two texts is that `CommonCrawl` has some smaller peaks for characters with rank greater than 125. For `DNA` and `Prot` the similarities and differences are similar. Due to the lager alphabet `Prot` has lower peaks. But those peaks are for characters between ranks 65 and 90.

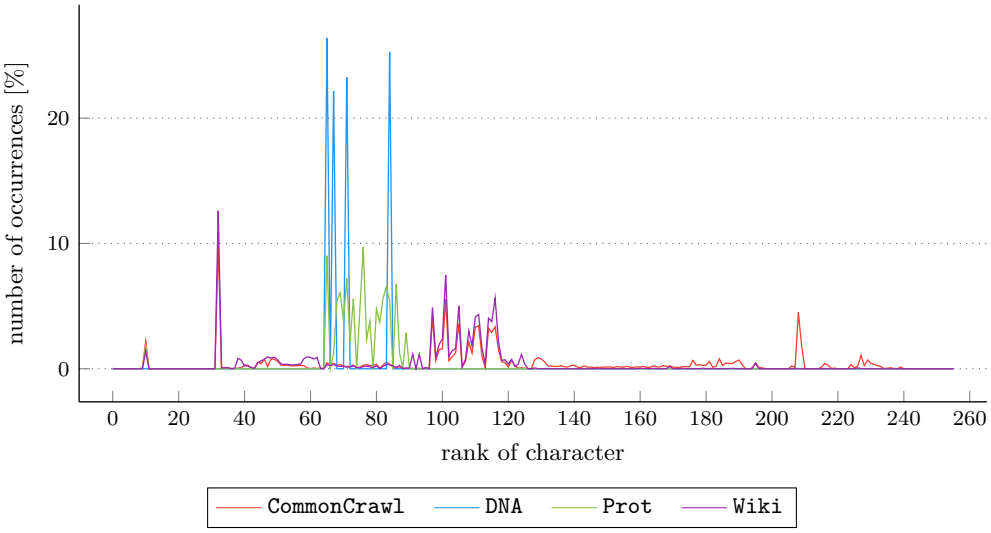


Figure 1.6. Histograms of texts used as inputs in this dissertation (over the ASCII-alphabet). We normalized the number of occurrences for better comparability.

1.5 CORRESPONDING PUBLICATIONS

A list of publications that are part of this dissertation is listed below, in Section 1.5.1, including a description of the author's contribution to them. In the list, we denote work that has been published in conference proceedings by *C*.

It should be noted that all prior work has been revised by adding and rewriting text to improve the clarity of the content, including new figures and examples to make it easier to understand, and redoing all experiments to have the same setup for all experiment throughout this dissertation.

1.5.1 Publications Contributing to this Dissertation

In Part I, we consider the wavelet tree and wavelet matrix construction. First, the transformation from the wavelet tree to the wavelet matrix and the work on wavelet tree and wavelet matrix construction described in Sections 2.4, 3.2 and 3.3 is based on

(C1) Johannes Fischer, Florian Kurpicz, and Marvin Löbel. "Simple, Fast and Lightweight Parallel Wavelet Tree Construction". In: *20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2018, pages 9–20.

The novel idea for the construction of wavelet trees and wavelet matrices has been developed by Johannes Fischer and Florian Kurpicz together. A very prototypical implementation was developed in a Bachelor's thesis [Oes16] supervised by Johannes Fischer and Florian Kurpicz. The final implementation is included in a framework for

wavelet tree and wavelet matrix construction that was coded by Florian Kurpicz, who was supported by Marvin Löbel.

Next, in Section 3.4, we extended this idea to also work in semi-external and external memory. To this end, we had to adapt the the computation of the wavelet tree to the limitations of the external memory model (see Section 1.3.4). This section is based on

(C2) Jonas Ellert and Florian Kurpicz. "Parallel External Memory Wavelet Tree and Wavelet Matrix Construction". In: *26th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. volume 11811. Lecture Notes in Computer Science. Springer, 2019, pages 407–416.

The general ideas for the external memory construction have been developed by Jonas Ellert and Florian Kurpicz. The implementation is integrated in the framework developed by Florian Kurpicz and Marvin Löbel and was coded by Jonas Ellert.

Part II is about suffix array construction. We mostly consider the distributed memory model (see Section 1.3.2) but we start with sequential algorithms in main memory, as those are the foundation for *all* work on distributed suffix array construction. In Chapter 4, the description of practical suffix array construction in main memory is based on

(C3) Johannes Fischer and Florian Kurpicz. "Dismantling DivSufSort". In: *Prague Stringology Conference (PSC)*. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017, pages 62–76.

The detailed description of the DivSufSort (implemented by Yuta Mori) has been written by Florian Kurpicz, the idea to also induce the LCP array has been developed by Johannes Fischer and Florian Kurpicz. The extension of DivSufSort—such that it also computes the LCP array—was coded by Florian Kurpicz.

We also published the results of a *project group*, which is a course where up to 12 students work for two semesters on one project, that was supervised by Johannes Fischer and Florian Kurpicz. The project was to build a framework that allows for an easy comparison of suffix array construction algorithms, and that also includes all publicly available suffix array construction algorithms. We use this framework to give an overview of existing suffix array construction algorithms in Section 4.1.

(C4) Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. "SACABench: Benchmarking Suffix Array Construction". In: *26th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. volume 11811. Lecture Notes in Computer Science. Springer, 2019, pages 392–406.

Most of the code was written by the students. The final paper was written (loosely based on a report presented by the students [Bah+19a]) by Florian Kurpicz.

The work on distributed suffix array construction in Chapter 5 is based in the following two publications

(C5) Timo Bingmann, Simon Gog, and Florian Kurpicz. "Scalable Construction of Text Indexes with Thrill". In: *2018 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, 2018, pages 634–643, and

(C6) Johannes Fischer and Florian Kurpicz. "Lightweight Distributed Suffix Array Construction". In: *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2019, pages 27–38.

The first paper contains distributed suffix array construction algorithms implemented using the distributed big data batch processing framework *Thrill*. Here, Florian Kurpicz developed all prefix doubling algorithms and also described them in the paper. The implementation and description of the recursive algorithms is the work of Timo Bingmann. This work is also presented in Timo Bingmann's [Bin18] dissertation. For this dissertation, however, we conducted more experiments to incorporate the result in our newer research, i. e., the paper (C6). Regarding the second paper, the idea to distribute the DivSufSort was developed by Florian Kurpicz who was supported by Johannes Fischer. The implementation has been coded by Florian Kurpicz.

Finally, we consider distributed full-text indices, i. e., more complex distributed indices that are often built on top of suffix arrays. The distributed index presented in Chapter 6 is based on

(C7) Johannes Fischer, Florian Kurpicz, and Peter Sanders. "Engineering a Distributed Full-Text Index". In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2017, pages 120–134.

Here, the idea to use an hierarchical distributed index based on Patricia tries, which can easily be constructed in distributed memory using suffix array and longest common prefix array, was developed by all authors together. The implementation of the distributed full-text index that we present in this part was coded by Florian Kurpicz.

Part I

Shared and External Memory Wavelet Tree Construction

CHAPTER 2

OVERVIEW OF WAVELET TREE CONSTRICTION

The *wavelet tree* is a compact data structure first presented by Grossi et al. [Gro+03] that generalizes among others $access(i)$ (character at position i), $rank(i, \alpha)$ (number of occurrences of character α up to and including position i), and $select(i, \alpha)$ (position at which the i-th α occurs) queries from a binary alphabet to alphabets of arbitrary size σ. Using a wavelet tree answering those queries requires time $\mathcal{O}(\lg \sigma)$.

We are also interested in the *wavelet matrix*, which was introduced by Claude et al. [Cla+15] as an alternative representation of wavelet trees. Wavelet matrices can answer the same queries that wavelet trees can answer in the same asymptotic time, while requiring at most as much space as the wavelet tree. Wavelet trees can have a huge space overhead, when the alphabet size is significant compared to the text length, as we will see in the next section, and realizations that require the same space as the wavelet matrix are slower than wavelet matrices in practice [Cla+15].

Wavelet trees and wavelet matrices are used for compression [Gro+11; Mak12], in computational geometry as an alternative to fractional cascading [MN06], for text indexing [Gro+03], variable length gap pattern matching [Bad+16], and to compute the Burrows-Wheeler Transform [KK19]. They are also part of FM-indices [FM05], making their efficient construction relevant for applications like DNA sequence assembly [SD10] and again compression [Kär+16]. Additional information on further applications can be found in multiple surveys [Fer+09; Gro+11; Mak12; Nav14].

Part I of this dissertation is focused is on the efficient construction of wavelet trees and wavelet matrices in shared and external memory. It is based on our papers on engineering shared memory wavelet tree and wavelet matrix construction [Fis+18], their parallel construction in external memory [EK19], and on yet unpublished research about Huffman-shaped wavelet trees and wavelet matrices. We first define the wavelet tree and wavelet matrix in Sections 2.2 and 2.3. In Section 2.4, we show how to extend wavelet tree construction algorithms such that they compute the wavelet matrix instead in the same asymptotic time. Next, in Section 2.5, we look at existing wavelet tree construction algorithms. Finally, in Chapter 3, we describe our novel wavelet tree and wavelet matrix construction algorithms and provide an extensive evaluation.

2.1 PRELIMINARIES

Let $T = T[0] \dots T[n-1]$ be a text of length n over an alphabet $\Sigma = [0, \sigma)$. Each character $T[i]$ can be represented using $\lceil \lg \sigma \rceil$ bits. The leftmost bit is the *most significant bit* (MSB), hence the *least significant bit* (LSB) is the rightmost bit. We denote the binary representation of a character $\alpha \in \Sigma$ that uses $\lceil \lg \sigma \rceil$ bits as bits(α), see Figure 2.1. Whenever we write a binary representation of a value, we indicate it by a subscript two. The k-th bit (from MSB to LSB) of a character α is denoted by $\mathrm{bit}(k, \alpha)$ for all $0 \leq k < \lceil \lg \sigma \rceil$.

α	bit(α)
0	$(000)_2$
1	$(001)_2$
2	$(010)_2$
3	$(011)_2$
4	$(100)_2$
5	$(101)_2$
6	$(110)_2$
7	$(111)_2$

MSB LSB

Figure 2.1. Binary representation of all characters in $\Sigma = [0, 8)$.

The *bit prefix* of size k of $\alpha \in \Sigma$ are the k most significant bits, i. e., $\mathrm{bit\_prefix}(k, \alpha) = (\mathrm{bit}(0, \alpha) \dots \mathrm{bit}(k-1, \alpha))_2$. We interpret sequences of bits as integer values.

Let BV be a bit vector of size n. The operation $\mathrm{rank}_0(\mathsf{BV}, i)$ returns the number of 0's in $\mathsf{BV}[0, i)$, whereas $\mathrm{select}_0(\mathsf{BV}, i)$ returns the position of the i-th 0 in BV. The operations $\mathrm{rank}_1(\mathsf{BV}, i)$ and $\mathrm{select}_1(\mathsf{BV}, i)$ are defined analogously. Both rank and select queries on a bit vector of size n can be answered in $\mathcal{O}(1)$ time using succinct dictionary data structures that requires only $o(n)$ bits space [Nav16].

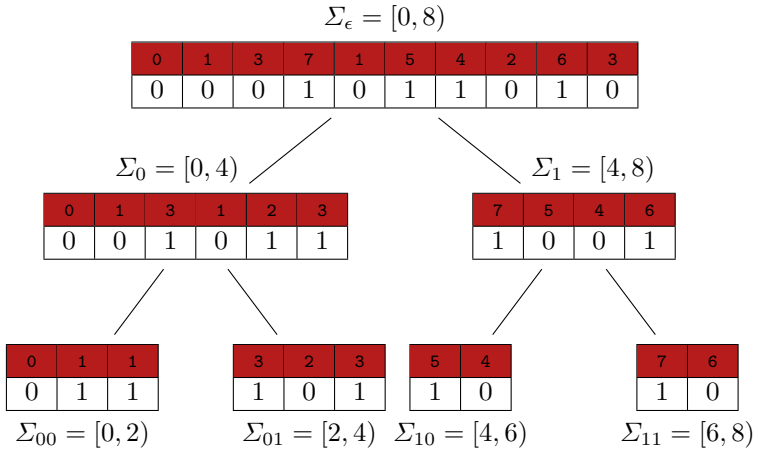
Given an array A of n integers and an associative operator $+$ (we only use addition), the zero-based *prefix sum* for A returns an array $B[0, n)$ with $B[0] = 0$ and $B[i] = A[i-1] + B[i-1]$ for all $i \in [1, n)$. If not zero-based, B is usually defined as $B[0] = A[0]$ and $B[i] = A[i-1] + B[i-1]$ for all $i \in [1, n)$.

2.2 THE WAVELET TREE

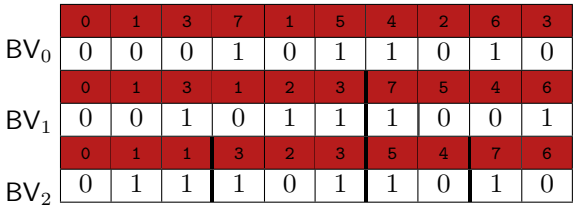
Let T be a text of length n over an alphabet $[0, \sigma)$. The *wavelet tree* [Gro+03] of T is a complete and balanced binary tree. Each node of the wavelet tree represents characters in $[\ell, r) \subseteq [0, \sigma)$. The root of the wavelet tree represents characters in $[0, \sigma)$, i. e., all characters. The left (or right) child of a node representing characters in $[\ell, r)$ represents the characters in $[\ell, (\ell + r)/2)$ (or $[(\ell + r)/2, r)$, respectively). A node is a leaf if $l + 2 \geq r$.

Characters in $[\ell, r)$ at the corresponding node v are *represented* using a bit vector BV_v such that the i-th bit in BV_v is $\mathrm{bit}(d(v), T_{[\ell,r)}[i])$, where $d(v)$ is the depth of v in the wavelet tree, i. e., the number of edges on the path from the root to v, and $T_{[\ell,r)}$ denotes the array containing the characters of T (in the same order) that are in $[\ell, r)$.

There are two variants of the wavelet tree: the *pointer-based* [Gro+03] and the *levelwise* [MN07] wavelet tree. The pointer-based wavelet tree uses pointers to represent the tree structure, see Figure 2.2a. Therefore, it requires space for $\mathcal{O}(\sigma)$ pointers in addition to the bit vectors and succinct dictionary data structures for the binary rank and select queries.



(a) Pointer-based wavelet tree.



(b) Level-wise wavelet tree.

Figure 2.2. The pointer-based (a) and the level-wise (b) wavelet tree of $T = [0, 1, 3, 7, 1, 5, 4, 2, 6, 3]$. The dark red (■) arrays contain the characters represented at the corresponding position in the bit vector and are not a part of the wavelet trees. In Figure 2.2a, Σ_α denotes the characters that are represented by the bit vector for $\alpha \in \{\epsilon, 0, 1, 00, 01, 10, 11\}$. In Figure 2.2b, thick lines represent the starting positions of the intervals. All this auxiliary information is not stored explicitly.

In the level-wise wavelet tree, we concatenate the bit vectors of all nodes at the same depth in a pointer-based wavelet tree. Since we lose the tree topology, the resulting bit vectors correspond to a *level* that is equal to the depth of the concatenated nodes. We store only a single bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$, see Figure 2.2b. This retains the functionality from the pointer-based wavelet tree [MN06; MN07], but reduces the redundancy for the succinct dictionaries needed to answer rank and select queries on the bit vectors in constant time. The bit vectors that we concatenated to obtain the level-wise wavelet tree form intervals within the resulting bit vector (of the level-wise wavelet tree). The interval in a bit vector of a wavelet tree in which a character is represented at level ℓ is encoded by its length-ℓ bit prefix:

Observation 2.1 (Fuentes-Sepúlveda et al. [FS+17]). *Given a character $T[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the wavelet tree, the interval in which $T[i]$ is represented in BV_ℓ can be computed by* bit_prefix$(\ell, T[i])$.

There is also a variant of the level-wise wavelet tree, where in addition to the bit vectors, we also store the starting positions of the intervals in the last level of the wavelet tree. This variant is called the *extended* variant [CN08]. This version requires $\sigma \lceil \lg n \rceil$ bits more space than the level-wise wavelet tree, but is also faster in practice.

The wavelet tree (both variants) can be used to generalize the operations access, rank, and select from binary alphabets to alphabets of size σ. Answering these queries then requires $\mathcal{O}(\lg \sigma)$ time. To do so, the bit vectors of the wavelet tree are augmented by binary rank and select data structures. For further information on queries we point to [Cla+15; Nav16]. Throughout this dissertation, we refer to the *level-wise* wavelet tree, whenever we speak about wavelet trees. All algorithms can be easily adopted to compute the pointer-based wavelet tree or the extended version of the level-wise wavelet tree instead.

2.3 THE WAVELET MATRIX

A variant of the wavelet tree, the *wavelet matrix*, was introduced in 2011 by Claude et al. [Cla+15]. It requires the same space as a wavelet tree and has the same asymptotic running time for access, rank, and select queries. But in practice it is often faster than a wavelet tree for rank and select queries [Cla+15], as it needs fewer calls to binary rank and select data structures. However, the fact that the wavelet matrix loses some nice structural properties of wavelet trees—the tree structure to be precise—makes it harder to compute, as *divide-and-conquer* wavelet tree construction algorithms, e. g. [Lab+17], cannot simply be transformed to wavelet matrix construction algorithms.

For the definition of the wavelet matrix, we need additional notations: *Reversing* the significance of the bits is denoted by reverse, e. g., $\text{reverse}((001)_2) = (100)_2$. The *bit-reversal* permutation of order k (denoted by ρ_k) is a permutation of $[0, 2^k)$ with $\rho_k(i) = (\text{reverse}(\text{bits}(i)))_2$. For example, $\rho_2 = (0, 2, 1, 3) = ((00)_2, (10)_2, (01)_2, (11)_2)$. ρ_k and ρ_{k+1} can be computed from another, as $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k-1), 2\rho_k(0)+1, \ldots, 2\rho_k(2^k-1)+1)$ and $\rho_k = (\rho_{k+1}(0)/2, \ldots, \rho_{k+1}(2^k-1)/2)$. In practice, we can realize the division by a single bit shift.

The wavelet matrix has only a single bit vector BV_ℓ per level $\ell \in [0, \lceil \lg \sigma \rceil)$ like the level-wise wavelet tree, but the tree structure is discarded completely in the sense that we do not require each character to be represented in an interval that is covered by the character's interval on the previous level. In addition, we use the array $\mathsf{Z}[0, \lceil \lg \sigma \rceil)$ to store the number of zeros at each level ℓ in $\mathsf{Z}[\ell]$. Therefore, the wavelet matrix requires $\lceil \lg \sigma \rceil \lceil \lg n \rceil$ bits in addition to the space required for the bit vectors and rank and select data structure.

BV_0 of the wavelet matrix contains the MSBs of each character in T in text order (this is the same as the first level of a wavelet tree). For $\ell \geq 1$, BV_ℓ is defined as follows. Assume that a character α is represented at position i in $\mathsf{BV}_{\ell-1}$. Then the position

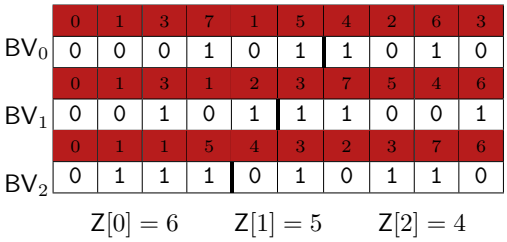


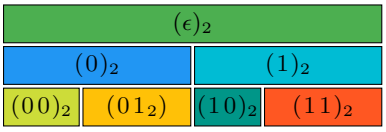
Figure 2.3. The wavelet matrix of our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$. Again, the dark red (●) arrays contain the characters represented at the corresponding position in the bit vector and are not a part of the wavelet matrix.

of its ℓ-th MSB in BV_ℓ depends on $\mathsf{BV}_{\ell-1}[i]$ in the following way: if $\mathsf{BV}_{\ell-1}[i] = 0$, $\mathrm{bit}(\ell, \alpha)$ is stored at position $\mathrm{rank}_0(\mathsf{BV}_{\ell-1}, i)$, and otherwise ($\mathsf{BV}_{\ell-1}[i] = 1$), it is stored at position $\mathsf{Z}[\ell - 1] + \mathrm{rank}_1(\mathsf{BV}_{\ell-1}, i)$. For an example of a wavelet matrix, see Figure 2.3.

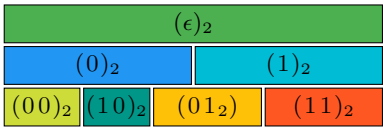
The intervals that occur in the bit vectors of a wavelet tree also occur in the bit vectors of a wavelet matrix for the same text, see Figure 2.4. The bits and also the characters represented by these bits are the same within these intervals—only the order of the intervals differs between the wavelet tree and the wavelet matrix, the content of the intervals is the same. Hence, the first two levels of a wavelet tree and wavelet matrix are the same. To be more precise:

Observation 2.2. *Given a character $T[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the wavelet matrix, the interval pertinent to $T[i]$ in BV_ℓ can be computed by reverse*(bit_prefix$(\ell, T[i])$).

This leads to a naive wavelet matrix construction algorithm, where $\mathsf{BV}_\ell[i] = \mathrm{bit}(\ell, T'[i])$, where T' is T stably sorted using the reversed bit prefixes of length ℓ of the characters as key [Cla+15].



(a) Intervals of a wavelet tree.



(b) Intervals of a wavelet matrix.

Figure 2.4. Positions of the intervals in a level-wise wavelet tree (a) and in a wavelet matrix (b). The intervals are identified by the bit prefix that all characters represented by bits in the interval have in common. The first two levels are the same; the difference is in the third level, where the order of $(01)_2$ and $(10)_2$ is interchanged.

2.4 FROM THE WAVELET TREE TO THE WAVELET MATRIX

We can make use of the similarities between wavelet trees and wavelet matrices, compare Observations 2.1 and 2.2, which we also depict in Figure 2.4, by showing that *every* algorithm that can compute a wavelet tree can also compute a wavelet matrix in the same asymptotic time.

Lemma 2.1. *We can compute in-place an array* X *and a bit vector* U *with rank and select data structures in time* $\mathcal{O}(n+\sigma)$ *and space* $(n+\sigma)(1+o(1)) + (\sigma+2)\lceil \lg n \rceil$ *bits, such that* $\mathsf{BV}_\ell^{WT}[i] = \mathsf{BV}_\ell^{WM}[j]$ *with*

$$j = \begin{cases} i, & \text{if } \ell \leq 1 \\ \mathsf{X}[2^{\ell-1} - 2 + bp] + off, & \text{otherwise} \end{cases}$$

where BV_ℓ^{WT} *and* BV_ℓ^{WM} *denote the bit vector of the wavelet tree and wavelet matrix, respectively. Also,* $bp = \operatorname{prefix}(\ell, \operatorname{rank}_0(\mathsf{U}, \operatorname{select}_1(\mathsf{U}, i+1)))$ *and* $off = i - \operatorname{rank}_1(\mathsf{U}, \operatorname{select}_0(\mathsf{U}, bp \ll (\lceil \lg \sigma \rceil - \ell)))$*, with* $\ll k$ *denoting a left bit shift by* k *bits, i. e., affixing* k *zeros on the right hand side and then removing the* k *MSBs.*

Proof. We require two auxiliary data structures for the transformation. The first one is the bit vector U of length $n+\sigma$ that stores the unary representation of the histogram of all characters in T. The second one is an array X of size $(\sigma+2)\lceil \lg n \rceil$ bits, which at first is used for counting, and later on stores the starting positions of all intervals in the wavelet matrix.

To compute U we first count the number of occurrences of all characters and store them in X such that $\mathsf{X}[i] = |\{j \in [0,n) \colon T[j] = i\}|$ for all $i \in [0,\sigma)$. Then, the unary histogram is given by $\mathsf{U} = 1^{\mathsf{X}[0]}0 1^{\mathsf{X}[1]}0 \ldots 1^{\mathsf{X}[\sigma-1]}$. In addition, we augment U with a rank and select data structure. All this requires $\mathcal{O}(n+\sigma)$ time and $o(n+\sigma)$ bits space in addition to U and X.

Next, we want to compute the starting positions of the intervals in the wavelet matrix; we fill the array X with its final content. To this end, we must compute the starting positions for intervals corresponding to bit prefixes of size ℓ with $\ell \in [2, \lceil \lg \sigma \rceil)$, i. e., for all but the first level of the wavelet matrix. To this end, we compute the number of occurrences of characters that share a bit prefix of size $\lceil \lg \sigma \rceil - 1$ in the first $\lceil \sigma/2 \rceil - 1$ positions of X. With the histogram information still in X, this can be done by setting $\mathsf{X}[i] = \mathsf{X}[2i] + \mathsf{X}[2i+1]$ for all $i \in [0, \lceil \sigma/2 \rceil)$ in increasing order. We set all other positions of X to zero.

Then, we compute the zero-based prefix sum with respect to $\rho_{\lceil \lg \sigma \rceil - 1}$ of the first $\lceil \sigma/2 \rceil - 1$ entries of X and store them in the last $\lceil \sigma/2 \rceil - 1$ entries of X. Here, "respect to $\rho_{\lceil \lg \sigma \rceil - 1}$" means that character $\rho_{\lceil \lg \sigma \rceil - 1}(i)$ follows character $\rho_{\lceil \lg \sigma \rceil - 1}(i-1)$ for all $i \in [1, \lceil \sigma/2 \rceil)$. In the same fashion, we compute the starting positions of the intervals in all other levels: by first computing the number of occurrences of bit prefixes of size k using the ones of size $k+1$ and storing the zero-based prefix sum with respect to $\rho_{\lceil \lg k \rceil}$ in the rightmost free entries of X. The $\sigma+2$ entries (of size $\lceil \lg n \rceil$) in X are sufficient to store the result at the back, where X still has unused entries.

Since the first entries of X can be empty (depending on σ), as we do not require the histogram any more, we finally move the starting positions that we have stored at the beck to the front, such that the first starting position is stored in $\mathsf{X}[0]$. All this can be done in $\mathcal{O}(\sigma)$ time without any additional space. Therefore, the construction of U, building its augmenting rank and select data structure, and computing X requires $\mathcal{O}(n+\sigma)$ time and $(n+\sigma)(1+o(1))+(\sigma+2)\lceil\lg n\rceil$ bits of space.

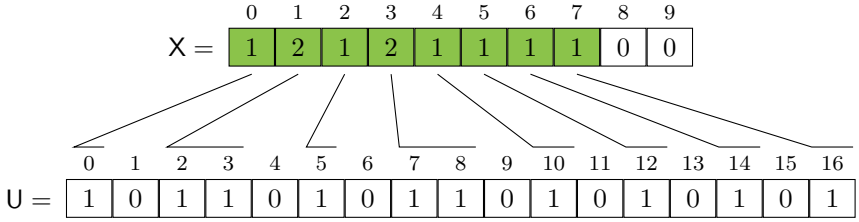
Now, we need to answer queries asking for a position $j \in [0,n)$ in $\mathsf{BV}_\ell^{\mathsf{WM}}$ given a position $i \in [0,n)$ in $\mathsf{BV}_\ell^{\mathsf{WT}}$ for $\ell \in [0, \lceil\lg\sigma\rceil)$ in constant time, i. e., the position j in the wavelet matrix corresponding to the position i in the wavelet tree. If $\ell \leq 1$ we know that $j = i$, because the bit vectors of the wavelet tree and wavelet matrix are the same for the first two levels. Otherwise ($\ell > 1$), the computation of the position j consists of two steps. First, we determine the starting position of the interval in the wavelet matrix (using X). Second, we compute the number of entries in the interval existing before i (which is the same for wavelet tree and wavelet matrix, as the intervals are the same):

1. We first need to identify the bit prefix of length ℓ corresponding to the interval containing i. Note that we are only interested in the bit prefix and not in the character c corresponding to position i. There are at least $i-1$ (or none, if $i=0$) characters occurring in T whose bit prefix of length ℓ is at most $\mathrm{bit\_prefix}(\ell, c)$. There are more than $i-1$ characters if at least one character with bit prefix $\mathrm{bit\_prefix}(\ell, c)$ occurs after c in T. Therefore, $c' = \mathrm{rank}_0(\mathsf{U}, \mathrm{select}_1(\mathsf{U}, i+1))$ has the same bit prefix of length ℓ as c. To be more formal, $bp = \mathrm{bit\_prefix}(\ell, c') = \mathrm{bit\_prefix}(\ell, c)$. Since we have stored all starting positions of the intervals on level ℓ in the wavelet matrix in $\mathsf{X}[2^\ell-2, 2^{\ell+1})$, the starting position is $\mathsf{X}[2^\ell-2+bp]$.
2. Now we need to compute the offset of the position from the starting position of the interval. To do so, we compute the smallest character contained in the interval by padding the bit prefix with $\lceil\lg\sigma\rceil - \ell$ 0's, giving us a value $r = \mathrm{select}_0(\mathsf{U}, bp \ll \lceil\lg\sigma\rceil - \ell)$. Next, we determine the number of 1's occurring before the r-th 0 in U to compute the offset, i. e., $\mathit{off} = i - \mathrm{rank}_1(\mathsf{U}, r)$.

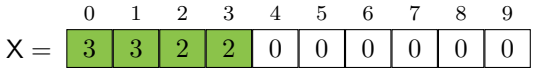
Since all operations used for querying require constant time and there is only a constant number of operations, the query can be answered in constant time. □

Before we give a detailed example of the construction and content of U and X during all phases of the proof, and describe how we the use U and X to get from a position in a wavelet tree to the position in the corresponding wavelet matrix on the next page, we want to mention a recently published related result by Dinklage [Din19] that extends our result. First, they show how to get *from the wavelet matrix to the wavelet tree*, i. e., the other direction of Lemma 2.1. However, even though they are able to do this in the same asymptotic time by using another array C that contains the histogram of the text, which helps to identify the characters in the wavelet matrix, they requires additional $\mathcal{O}(\sigma \lg n)$ bits of space for the histogram. Then, they provide another version of Lemma 2.1 where they require only $\mathcal{O}(n)$ time and require σ bits less space, which however only works for effective alphabets.

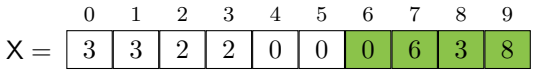
Example. Given our running example of $T = [0, 1, 3, 7, 1, 5, 4, 2, 6, 3]$, we compute the bit vector U and the array X. Whenever an entry in X changes, we highlight it using a green background (🟢). First, we compute the histogram of the characters in the text, and also fill U with the histogram encoded in unary. We also compute the rank and select data structure that augments U, which is not depicted.



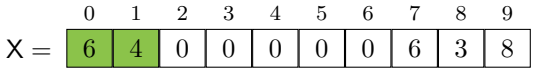
Next, we compute the histogram for all bit prefixes of size two ($\lceil \lg \sigma \rceil - 1$ in general), i. e., $(00)_2$, $(01)_2$, $(10)_2$, and $(11)_2$. We store the histogram at the front of X.



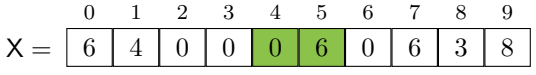
Then, we compute the starting positions for the intervals in the third level of the wavelet matrix using the histogram of bit prefixes that we have just computed. We store the starting positions of the intervals at the back of X.



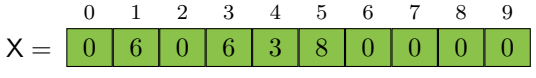
We repeat the last two steps, but this time, we consider the bit prefixes of size one, which correspond to the second level of the wavelet matrix, and store them at the front of X. Note that this is not necessary for the algorithm, as the first two levels of the wavelet tree and wavelet matrix are the same, but we compute it here for the sake of this example to show the general idea of the construction of X for multiple levels.



Then, we compute the starting positions of the corresponding intervals in the wavelet matrix and store them at the rightmost free entries in X.



Finally, we move the starting positions of the intervals, which we currently have stored on the back of the X, to the front and fill all unused entries with zeros.



As mentioned above, the first two levels of the wavelet tree and wavelet matrix are the same, hence we give an example for the last level. We want to set the 8-th bit ($i = 7$) in $\mathsf{BV}_2^{\mathsf{WT}}$ to 0. Now, we need to compute the corresponding position j in $\mathsf{BV}_2^{\mathsf{WM}}$. To do so, we first identify the position of the $(i+1) = (7+1)$-th 1 in U, i. e., $p = \text{select}_1(8) = 12$. The value represented by this position ($\text{rank}_0(12) = 5 = (101)_2$) may not correspond to the value of the considered character, but it has the same bit prefix of length 2 as the character. The length-2 bit prefix is $bp = \text{bit\_prefix}(2, \text{rank}_0(14)) = (10)_2 = 2$. Below, we show U with the number of zeros up to a certain position given in red (🔴, only below zeros in U) and the number of ones up to a certain position given in blue (🔵, only below ones in U).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{U} =$	1	0	1	1	0	1	0	1	1	0	1	0	1	0	1	0	1
	1	1	2	3	2	4	3	5	6	4	7	5	8	6	9	7	10

To get the first position in the interval in level 2, we need to pad the bit prefix with $\lceil \lg \sigma \rceil - \ell = 1$ zeros to get the smallest value with the bit prefix bp, i. e., $(100)_2 = 4$. Now we can compute the offset of the position with respect to the first position of the interval. We identify the starting position of the interval containing 4, i. e., $\text{select}_0(4) = 9$. Then we get the number of 1's up to that position ($\text{rank}_1(9) = 6$) and subtract this value from i to get the offset, i. e., $off = 7 - 6 = 1$. Using the bit prefix bp and the offset off, we can get the position where we have to set the bit using $\mathsf{X}[2^2 - 2 + bp] + off = \mathsf{X}[2 + 2] + 1 = 3 + 1 = 4$. When we look at the wavelet tree in Figure 2.2b and the wavelet matrix in Figure 2.3, we see that the 8-th bit in the wavelet tree corresponds to the character 4 and the 5-th bit in the wavelet matrix (the bit $\mathsf{BV}_2^{\mathsf{WM}}[4]$) also corresponds to the character 4, which concludes this example.

2.5 RELATED WORK

While wavelet trees and wavelet matrices are easy to compute naively, there exist many more sophisticated algorithms that improve the running time compared to the trivial $\mathcal{O}(n \lg \sigma)$ time and minimize the space that is required in addition to the $n\lceil \lg \sigma \rceil$ bits for the bit vectors (ignoring space required for the supporting rank and select data structures). In this section, we mainly focus on wavelet tree construction, as there are—to the author's best knowledge—no papers solely considering the wavelet matrix, expect for its initial presentation [Cla+15] and the algorithms we present in Chapter 3. Still, some wavelet tree construction algorithms can easily be modified to compute the wavelet matrix instead of the wavelet tree.

First, in Section 2.5.1, we discuss sequential wavelet tree and wavelet matrix construction algorithms. Then, in Section 2.5.2, we present parallel wavelet tree and wavelet matrix construction algorithms that work in the parallel random-access machine. Last, in Section 2.5.3, we briefly discuss wavelet tree construction algorithms in the semi-external memory model and the bulk-synchronous parallel model.

2.5.1 Sequential Wavelet Tree Construction Algorithms

An overview of the sequential wavelet tree construction algorithms presented in this section is given in Table 2.1, which contains the running times and the required additional memory (in addition to the input and resulting wavelet tree) in bits of the wavelet tree construction algorithms that we briefly describe below. Here, we focus on practical algorithms that mostly have publicly available implementations. Some theoretically interesting algorithms, e. g., the in-place construction algorithms by Tischler [Tis11] are briefly touched in Section 2.5.3.

Grossi et al. [Gro+03] only use the wavelet tree as a building block that is part of a more complex index. Therefore, only a general description of the structure of the wavelet tree is given—no dedicated wavelet tree construction algorithm is described. Still, computing the wavelet tree for a text of size n over an alphabet of size σ *naively* in time $\mathcal{O}(n \lg \sigma)$ is simple. To compute the level ℓ of the wavelet tree, we only have to sort the text using the length-ℓ bit prefix of the characters as sort key. If we compute the wavelet tree top-down (starting with $\ell = 0$ and then consecutively increasing ℓ by one) and sort the text *stably*, we do not require an additional copy of the text. Since this preserves the order of the text with respect to the required intervals on the next level on each level. To obtain the same running time of $\mathcal{O}(n \lg \sigma)$, we must sort the text in linear time. To this end, we can simply use Counting sort. However, *stable* Counting sort requires additional $n\lceil \lg \sigma \rceil$ bits to store the sorted text [Sed98, p. 300].

This is similar to the naive wavelet tree construction algorithm, which we briefly discuss due to its similarity. As we also consider the ℓ-th significant bit on level ℓ of the wavelet matrix, we only have to compute the different order of the intervals at each level. Claude et al. [Cla+15] describe the idea behind constructing the wavelet matrix *naively* as moving all zeros of a given level to the left, and all ones of the level to the right, i. e., after the computation of the current bit vector we append all characters represented by a zero to T_0 and all other characters to T_1, without changing the order of the characters. Then, we continue with the text $T = T_0T_1$ as input for the next level. This behavior can also be achieved by stably sorting the text using the ℓ-th significant bit as key (on level ℓ). The running time and the required additional memory is the same as for the naive wavelet tree construction algorithm: $\mathcal{O}(n \lg \sigma)$ time and $n\lceil \lg \sigma \rceil$ bits in addition to the input and output to store the sorted text.

Shun [Shu15] presents a more sophisticated version of the naive wavelet tree construction algorithm that provides practical improvements with respect to the naive wavelet tree construction algorithm. Again, the wavelet tree is constructed top-down. Each level ℓ (except the first) is computed based on the previous level $\ell - 1$, using the fact that we do not need to sort the whole text at once, but only the part of the text covered by the interval that is split into two intervals on the current level. To this end, the nodes are annotated with an offset and a length. The offset is the total number of characters represented by preceding nodes on the same level and the length is the number of characters represented by the current node. This information must be computed on each level with an additional scan of the text. Then, the text can be sorted while computing the bit vector of the level.

Table 2.1. Sequential wavelet tree and wavelet matrix construction algorithms in the Word RAM Model [Hag98]. We use dashes ("—") to mark algorithms for which no analysis of the required additional space is conducted by the authors.

Reference	Time Complexity	Additional Space (bit)
naive	$\mathcal{O}(n \lg \sigma)$	$n\lceil \lg \sigma \rceil$
seq.serial [Shu15]	$\mathcal{O}(n \lg \sigma)$	—
[Cla+11]	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(\lg n \lg \sigma)$
[Bab+15]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$	—
[Mun+16]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$	—
[FS17][†]	$\mathcal{O}(n \lg \sigma)$	$\alpha n \lceil \lg \sigma \rceil + \mathcal{O}(1)$
[Kan18]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$	—[‡]
Section 3.2	$\mathcal{O}(n \lg \sigma)$	$2\sigma \lceil \lg n \rceil$

[†] This is an online wavelet tree construction algorithm that does not need to know the alphabet size σ in advance.
[‡] The author states that in experiments the algorithm requires twice as much memory (in total) as the algorithm that we present in Section 3.2.1. No further analysis of the required space was conducted by the author.

Babenko et al. [Bab+15] and Munro et al. [Mun+16] independently improved the running time for wavelet tree construction algorithms to $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$ by using broadword programming techniques, i. e., applying operations to multiple integers that all fit into one computer word at the same time. They describe the algorithm in two phases:

(P1) Babenko et al. [Bab+15] first compute every τ-th level of the wavelet tree (those levels are called *big*), for a constant $\tau \geq 1$, which requires $\mathcal{O}(n \lg \sigma / \tau)$ time. (The first phase is not required, if $\tau > \lg \sigma$, as the root is the only big node.)

(P2) In the second phase, the *small* levels between the big ones are computed. Assume that we want to compute the bit vectors for the levels between the i-th and $i+1$-th big level (or the ones after the last big level). To this end, we only need τ bits starting at the $(i\tau+1)$-th bit for every character of the text. Therefore, for each big level, it suffices to store the corresponding τ bits in τ-bit integers, which allows us to represent $\lfloor \lg n / \tau \rfloor$ characters in one computer word and process them at the same time. Here, processing means that we write the currently considered bit to the bit vector and split the words into two lists, depending on this bit. Using these two lists, we can compute the next level, where we proceed recursively. By choosing $\tau = \sqrt{\lg n}$ we minimize the running time and get the targeted running time of $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$.

Munro et al. [Mun+16] describe the first phase based on L-ary wavelet trees. Here,

we get the following alternative to Babenko et al.'s [Bab+15] algorithm:

(P1′) To get an L-ary wavelet tree, we do not split the alphabet into two parts for each interval on each level, but into L-parts. Since each character could be pertinent to one of L intervals in the next level, we use integers in [0,L) instead of bits to represent the characters. Those integers are stored packed, i. e., $\lfloor \lg n / \lg L \rfloor$ integers can be stored in one computer word.

(P2′) In (P1′), we compute a different (implicit) representation of the big levels mentioned above in (P1). Therefore, the result of the first phase is the same, and the second phase of this algorithm is the same as (P2).

In the context of these two theoretically fastest wavelet tree construction algorithms, we also mention Kaneta's [Kan18] recently presented wavelet tree construction algorithms that bridge the gap between theory and practice and are based on the ones by Babenko et al. [Bab+15] and Munro et al. [Mun+16]. Their main contribution is a practical implementation of the broadword programming techniques using new CPU instructions. Unfortunately, their implementation is not publicly available.

2.5.2 Parallel Wavelet Tree Construction Algorithms

Now, we have a look at parallel wavelet tree construction algorithms. We give an overview of the algorithms, which we describe below, in Table 2.2. In addition to the practical wavelet tree construction algorithms, we also briefly describe three primarily theoretical algorithms by Shun [Shu17], as they are currently the best ones regarding work and time complexity.

Shun [Shu15] presents two parallel wavelet tree construction algorithms that both compute the wavelet tree top-down, i. e., from the first level to the last level. This is the main difference compared to our parallel wavelet tree construction algorithms that compute the wavelet tree bottom-up as we describe in Section 3.1.

1. Shun's [Shu15] first parallel wavelet tree construction algorithm (par.level) computes the histogram of each level in parallel. To this end, each processing element first computes a local histogram for the text scanned by it. Then, a prefix sum over all those local histogram yields the positions where the characters of each processing element are represented in the level. Then, the text is scanned once more—again in parallel—and the bit vector is created. Note that this approach requires two scans of the text.

2. Shun's [Shu15] second practical parallel construction algorithm is based on *sorting* (par.sort). Again, the general idea is that we construct the wavelet tree top down, i. e., from the first to the last level. For each level ℓ the text has to be scanned three times: once for creating the histogram of the length-ℓ bit prefixes and their intervals, then a second time for sorting using the starting positions, and a third time—now the sorted text has to be scanned—for the computation of the bit vector. Note that we can sort the text in parallel.

Table 2.2. Shared memory parallel wavelet tree and wavelet matrix construction algorithms in the Work-Time Model [JáJ92]. We use dashes ("—") to mark algorithms for which no analysis of the required additional space is conducted by the authors.

Algorithm	Work (left) and Time (right) Complexity		Additional Space (Bit)
par.level [Shu15]	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(\lg n \lg \sigma)$	$\mathcal{O}(n \lg n)$
par.sort [Shu15]	$\mathcal{O}(n \lg \lg n \lg \sigma)$	$\mathcal{O}(\lg n \lg \sigma)$	$\mathcal{O}(n \lg n \lg \sigma)$[¶]
par.rec [Lab+17]	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(\lg n \lg \sigma)$	$\mathcal{O}(p \lg n \lg \sigma)$
par.dd [FS+17]	$\mathcal{O}(\sigma n / \lg n + n \lg \sigma)$	$\mathcal{O}(\lg n \lg \sigma)$	$\mathcal{O}(n \lg \sigma)$
Section 3.3.1	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(n)$[†]	$\sigma \lceil \lg n \rceil$
Section 3.3.2	$\mathcal{O}(\lg \sigma (n + p\sigma))$[‡]	$\mathcal{O}(\lg \sigma (\frac{n}{p} + \lg p + \sigma))$[‡]	$n \lceil \lg \sigma \rceil + p\sigma \lceil \lg n \rceil$
Section 3.3.3	$\mathcal{O}(n \lg \sigma + p\sigma)$	$\mathcal{O}(\frac{n}{p} \lg \sigma + \lg p + \sigma)$	$n \lceil \lg \sigma \rceil + p\sigma \lceil \lg n \rceil$
[Shu17][*]	$\mathcal{O}(\frac{n \lg \lg n \lg \sigma}{\sqrt{\lg n \lg \lg n}})$	$\mathcal{O}(\lg n \lg \sigma)$	—
[Shu17][*]	$\mathcal{O}(\frac{n \lg \sigma}{\delta \sqrt{\lg n}})$[§]	$\mathcal{O}(\frac{n^\delta \lg \sigma}{\delta \sqrt{\lg n}})$[§]	—
[Shu17]	$\mathcal{O}(\frac{n \lg \sigma}{\sqrt{\lg n}})$	$\mathcal{O}(\sigma + \lg n)$	—

[¶] By computing the wavelet tree level-by-level, instead of all levels in parallel, the additional space can be reduced to $\mathcal{O}(n \lg n)$ bits. However, this increases the time to $\mathcal{O}(\lg n \lg \sigma)$. The work remains the same in both cases.
[†] We cannot use more than $\lceil \lg n \rceil$ processing elements. See Section 3.3.1 for the reasons.
[‡] This algorithm can efficiently use up to $p \leq n/\sigma$ processing elements. Using n/σ processing elements yields $\mathcal{O}(n \lg \sigma)$ work and $\mathcal{O}(\lg \sigma (\lg n + \sigma))$ time. When we employ more processing elements we only increase the required work, without achieving a better running time.
[*] Both algorithms are based on parallel stable integer sorting. The difference in work and time is a trade-off introduced by the used sorting algorithm.
[§] For a constant $\delta \in (0, 1)$.

Fuentes-Sepúlveda et al. [FS+17] present a wavelet tree construction algorithm using a meta-approach they call *domain decomposition* that we denote by *par.dd*. Here, the general idea is that each processing element computes a partial wavelet tree for a slice of the text. The slices are non-overlapping consecutive slices, such that by concatenating them we obtain the text. We can compute the partial wavelet trees in parallel. Then, we merge the partial wavelet trees—also in parallel—to obtain the final wavelet tree. We describe this approach in detail in Section 3.3.3. There, we present our domain decomposition framework that allows us to compute both wavelet trees and wavelet matrices. For the computation of the partial wavelet trees, any sequential wavelet tree construction algorithm can be used. Fuentes-Sepúlveda et al.'s [FS+17], par.dd uses a slightly adopted version of Shun's [Shu15] wavelet tree construction algorithm seq.serial, see Section 2.5.1. Labeit et al. [Lab+17], too, present a domain decomposition wavelet tree construction algorithm that uses a different merge function compared with Fuentes-Sepúlveda et al.'s [FS+17] one.

In addition to their domain decomposition wavelet tree construction algorithm, Labeit et al. [Lab+17] present the previously fastest parallel wavelet tree construction algorithm *par.rec.* It uses the operation *split* that, given a text T of length n and a *splitter* function $s\colon \Sigma \to \texttt{bool}$, generates two texts $T_{\texttt{true}}$ and $T_{\texttt{false}}$ such that for all $i \in [0..|\{k \in [0..n)\colon s(k) = \alpha\}|)$ and $\alpha \in \{\texttt{true}, \texttt{false}\}$ we have $T_\alpha[i] = T[j]$ where j is the only position that fulfills (i) $s(T[j]) = \alpha$ and (ii) $|\{k \in [0..j)\colon s(T[k]) = \alpha\}| = i$. Now, we split the text for each node (or interval) of the wavelet tree, such that the resulting two texts correspond to the text that is represented at the nodes children. To this end, we use

$$s_\ell(\alpha) = \begin{cases} \texttt{true}, & \mathrm{bit}(\ell, \alpha) = (0)_2 \\ \texttt{false}, & \mathrm{bit}(\ell, \alpha) = (1)_2 \end{cases}$$

as splitter function on level ℓ. Now, the input text is always the text that is represented at the corresponding node. With a scaling implementation of the split operation, we can always employ all processing elements as follows: at the first level, we use all p processing elements to split the text and compute the bit vector; then, on the second level, we use a number of processing elements proportional to the sizes of the two results of the previous split operations (but at least one processing element) to compute the bit vectors of the intervals *and* split the considered text into two. In total, all processing elements are used. This continues in the same fashion for each following level, always proportionally to the sizes dividing the number of processing elements available to split the text and compute the bit vector of the interval. Hence, all processing elements are used throughout the computation.

Last, we discuss Shun's [Shu17] work on parallel wavelet tree constuction, which are the theoretically best ones. Also, they are based on ideas that have already been described in Section 2.5.

1. The first algorithm is based on parallel integer sorting similar to par.sort. In general, this (parallel wavelet tree construction) algorithm is a parallelization of Babenko et al.'s [Bab+15] wavelet tree construction algorithm, which we described in Section 2.5.1. The main idea is to use parallel integer sorting for the computation of the big nodes in (P1). The parallel wavelet tree construction algorithm has two variants that have a trade-off between work and time, as it can achieve either $\mathcal{O}(\lg n \lg \sigma)$ time or $\mathcal{O}(n^\delta \lg \sigma / \delta \sqrt{\lg n})$ time. The work of those algorithms is either $\mathcal{O}(n \lg \lg n \lg \sigma / \sqrt{\lg n \lg \lg n})$ or $\mathcal{O}(n \lg \sigma / \delta \sqrt{\lg n})$. Here, the parameter $\delta \in (0, 1)$ in the work and time complexity of the work-efficient-variant comes from the use of a more work-efficient stable integer sorting algorithm with worse time complexity [Vis10, p. 38]. The used (stable and parallel) integer sorting algorithm is also the reason for the work-time-trade-off between the two variants of this wavelet tree construction algorithm. As either a work-inefficient algorithm, e. g., [Bha+91; Ram90], or an algorithm that is work-efficient but does not parallelize as well, e. g., [Vis10, p. 38] can be used. For the second phase (P2), we precompute lookup tables in parallel, which allow us to split the big levels in parallel.

2. The second algorithm is based on domain decomposition, a technique that we described ealier in this section. For the computation of the partial wavelet trees they use a $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$ time sequential wavelet tree construction algorithm, e. g., [Bab+15; Kan18; Mun+16]. Then, they introduce *boundary words* that allow for more independent computations during merging. Using these they can lower the work to $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$, which is lower than all other domain decomposition approaches, i. e., [FS+17; Lab+17] and the one that we present in Section 3.3.3.

2.5.3 Further Wavelet Tree Construction Algorithms

There also exist wavelet tree construction algorithms in other settings and models of computation. In particular, we are aware of the following four settings and models. We include all of these algorithms in our experiments if they have a publicly available implementation. Only the distributed memory algorithms are parallel wavelet tree construction algorithms. All other algorithms mentioned below are sequential.

Sequential In-Place Construction. While we can restore the text from the wavelet tree, the access operation requires $\mathcal{O}(\lg \sigma)$ time instead of the constant access time we have if we store the text in addition to the wavelet tree. For many applications of the wavelet tree, for example the FM-index [Gro+03], we would like to have constant time access operation to the text after the construction of the wavlete tree. Still, Claude et al. [Cla+11] and Tischler [Tis11] developed in-place wavelet tree construction algorithms that overwrite the text with the wavelet tree. This reduces the total amount of memory, as we do not need additional space for the wavelet tree anymore. The space for the auxiliary binary rank and select data structures is still needed.

1. Claude et al. [Cla+11] present two algorithms, a very space efficient one with running time $\mathcal{O}(n \lg n \lg^2 \sigma)$ and additional space requirements of $\mathcal{O}(\lg \sigma \lg n)$ bits, and another one with faster running time $\mathcal{O}(n \lg^2 \sigma)$ that, however, requires $n + \mathcal{O}(\lg \sigma \lg n)$ additional bits, and

2. Tischler [Tis11] presents two algorithms with running times $\mathcal{O}(\sigma \lambda^2 n + \lambda n \lg c)$ and $\mathcal{O}(\lambda n \lg c)$ that have space requirements of $\mathcal{O}(\sqrt{n}(\lambda + \lg n)/c) + \mathcal{O}(\lg n)$ and $\mathcal{O}(\sqrt{n}(\lambda + \lg n)/c) + \mathcal{O}(\lambda(\lg n + \lg \lambda))$ for constants $\lambda = \lceil \lg(\sigma + 1) \rceil$ and $c > 0$.

The author is not aware of any implementation of either of these sequential in-place wavelet tree construction algorithms.

Sequential Semi-External Memory. For the semi-external memory model, we mention the sequential wavelet tree and wavelet matrix construction algorithms that are part of the SDSL [Gog+14a], as these algorithms in the SDSL work in semi-external memory. Here, we also include seq.sdsl in our evaluation in Section 3.2.4.

Parallel Distributed Memory. Recently, our wavelet tree construction algorithms (which we describe in detail in Section 3.2) have been transformed to distributed memory algorithms by Dinklage et al. [Din+20]. While they can use more processing elements due to the model of computation, our parallel algorithms (Section 3.3) achieve a higher throughput on the same number of processing elements, as reported by the authors. Another improvement by Dinklage et al. [Din+20] is that they present the first parallel (shared memory and distributed memory) wavelet matrix construction algorithm whose memory requirements do not depend linearly on the alphabet size.

Sequential Online Construction. All wavelet tree construction algorithms that we described up to this point heavily depend on the size of the alphabet during construction, i. e., some in their running time and all in their memory requirements. Fonseca and Silva [FS17] present an online wavelet tree construction algorithm that does not need to know the alphabet size or the effective alphabet in advance. To this end, they compute the effective alphabet and build the tree structure dynamically such that both (effective alphabet and tree structure) can easily be extended. Finally, the dynamic tree structure is transformed to the final wavelet tree.

CHAPTER 3

ENGINEERING WAVELET TREE CONSTRUCTION

In this chapter, we present different wavelet tree and wavelet matrix construction algorithms that are all based on a novel technique: the *bottom-up* computation of wavelet trees and wavelet matrices (Section 3.1). With the bottom-up computation, we can reduce the number of text accesses during the computation of the wavelet tree or wavelet matrix, which significantly improves construction time in practice.

We then present three sequential (Section 3.2) and three shared memory parallel (Section 3.3) wavelet tree construction algorithms based on this technique. All these algorithms can easily be adapted to compute the wavelet matrix instead, making the parallel ones the first practical parallel wavelet matrix construction algorithms. We also present the first external memory wavelet tree and wavelet matrix construction algorithms (Section 3.4). Here, we use the basic idea of the bottom-up computation to reduce the number of I/Os, as we compute all information required for the computation in one scan of the text. Last, in Section 3.5, we discuss Huffman-shaped wavelet trees and wavelet matrices, which are built on the Huffman encoded text. This results in wavelet trees and wavelet matrices that can have bit vectors that are shorter than the uncompressed input text. To retain functionality, we use special Huffman codes.

3.1 BOTTOM-UP COMPUTATION OF HISTOGRAMS

As mentioned before, the idea of our construction algorithms is to compute the wavelet tree *bottom-up*. To this end, we first compute the histogram of characters of the text. Using this histogram, we can compute the histograms for all levels without another text access, and thus the starting positions of the intervals for any level ℓ of the wavelet tree (or wavelet matrix) using a zero-based prefix sum (with respect to ρ_ℓ for wavelet matrices) on the corresponding histogram. Generally speaking, if we have a histogram of length-ℓ bit prefixes, we can compute the histogram of the length-$\ell - 1$ bit prefixes, as each entry in this new histogram is the sum of the two entries that have a common length-$\ell - 1$ bit prefix in the old histogram. This does not require any text access, as the bit prefix corresponds to the position of the entry in the histogram.

For example, the number of characters with bit prefix $(01)_2$ is the total number of characters with bit prefixes $(010)_2$ and $(011)_2$, since only these characters have the common bit prefix $(01)_2$. We give an extended example in Figure 3.1 below.

Now, we briefly discuss the additional space requirements of this technique. Given a text of length n over an alphabet of size σ, the histogram of all characters requires $\sigma\lceil\lg n\rceil$ bits of space. We can reuse that space for all histograms at previous levels ℓ, which need $\sigma\lceil\lg n\rceil/2^{\lceil\lg\sigma\rceil-\ell}$ bits of space and storing the starting positions requires the same space as storing the histogram. Note that we do not need a histogram for the first level, and we also do not need the starting positions resulting from the histogram of all characters. Since we need at most $\lceil\sigma\lg n\rceil/2$ bits of space for the histogram (of the last level) and the starting positions, and we can reuse the space when computing both for the following level, we need $\sigma\lceil\lg n\rceil$ bits of space for both the histogram and the starting positions in total.

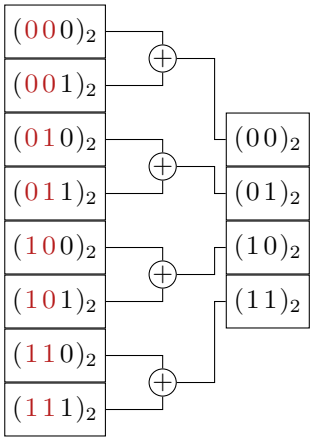


Figure 3.1. Computing histogram of bit prefixes of length two from histogram of length three bit prefixes. Common prefixes are highlighted in dark red (●).

In the sequential setting, the computation of all histograms and starting positions of the intervals requires $\mathcal{O}(\sigma\lg\sigma)$ time, which is dominated by the $\mathcal{O}(n\lg\sigma)$ running time of all sequential wavelet tree and wavelet matrix construction algorithms that we present in the next section. While this idea yields no theoretical improvement of the running time, in practice, it saves up to one scan of the text for each level compared to other wavelet tree construction algorithms, e. g., [FS+17; Shu15].

3.2 SEQUENTIAL CONSTRUCTION

Now, we employ the bottom-up construction in two different wavelet tree construction algorithms. First, in Section 3.2.1, for every level, we compute the starting positions of the intervals in a bottom-up fashion as described above, and fill the bit vector accordingly. This results in a lot of random access to the bit vector, which is of no major concern—except for cache misses—in the sequential setting. We also present a variant of this algorithm, where we first compute all histograms and then fill all bit vectors during a single scan of the text. In preparation for our parallel shared memory algorithms, which are presented in Section 3.3, where random access on bits is problematic, we also describe a second variant in Section 3.2.2, where the access pattern to the bit vectors is a scan from left to right. However, we need random read and write access to the text, which is easier to handle in the parallel setting. All wavelet tree construction algorithms that we present in this section can easily be adopted to compute the wavelet matrix instead (see Section 3.2.3).

Algorithm 3.1. Wavelet tree construction with prefix counting (seq.pc)

Input : Text T of length n and the alphabet size σ.
Output : A bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$ of the wavelet tree.

```
1  for i = 0 to n − 1 do
2  |  Hist[T[i]]++  // Compute histogram of the text (as basis for all other histograms).
3  |  BV_0[i] = bit(0, T[i])  // Fill first level's bit vector (characters' MSB in text order).
4  for ℓ = ⌈lg σ⌉ − 1 to 1 do  // Construct other levels of the wavelet tree bottom-up.
5  |  for i = 0 to 2^ℓ − 1 do  // Compute new histogram based the previous level's one.
6  |  |  Hist[i] = Hist[2i] + Hist[2i + 1]  // Update the histogram in-place.
7  |  for i = 1 to 2^ℓ − 1 do  // Get starting positions of intervals from new histogram.
8  |  |  Borders[i] = Borders[i − 1] + Hist[i − 1]  // Update the positions in-place.
9  |  for i = 0 to n − 1 do  // Fill the bit vector of the current level.
10 |  |  p = Borders[bit_prefix(ℓ, T[i])]++  // Get and update position for bit.
11 |  |  BV_ℓ[p] = bit(ℓ, T[i])  // Set the bit in the bit vector.
```

3.2.1 Prefix Counting

Our first wavelet tree construction algorithm (*seq.pc*, see Algorithm 3.1) starts with the computation of the initial histogram $\mathsf{Hist}[0, \sigma)$ of the text (line 2). In addition, the first level of the wavelet tree is computed, as it contains the most significat bits of all characters in text order (line 3). While this way of contruction is not truly bottom-up, we save an additional scan of the text to compute the bit vector for the first level. After this, all other levels are computed bottom-up. This requires $\mathcal{O}(n)$ and $\sigma\lceil \lg n \rceil/2$ bits space for the histogram.

Initially, we have a histogram for all characters in the text. During each iteration, say at level ℓ, we want to compute the histogram for all bit prefixes of length $\ell - 1$ of the characters in the text. We compute these histograms as described in Section 3.1 by ignoring the last bit of the considered bit prefixes. As described before, we can do so in $\mathcal{O}(\sigma)$ time using no additional space (lines 5 and 6).

Using the updated histogram that occupies $\sigma\lceil \lg n \rceil/2^{\lceil \lg \sigma \rceil - \ell}$ bits, we compute the starting positions of the intervals of the characters that can by identified by their bit prefix of size $\ell - 1$ for level ℓ with a zero-based prefix sum. We also require $\sigma\lceil \lg n \rceil/2^{\lceil \lg \sigma \rceil - \ell}$ bits to store these starting positions (array $\mathsf{Borders}$ in line 8). Again, this requires $\mathcal{O}(\sigma)$ time and only $\sigma\lceil \lg n \rceil$ bits in total, as we can reuse the space used during this step for the previouly considered level.

Last, we compute the bit vector for the current level ℓ. To do so, we scan the text once from left to right and consider the bit prefix of length $\ell - 1$ of each character. Now, we have stored the starting positions of all length-ℓ bit prefixes in array $\mathsf{Borders}$. Thus, when we consider the characters in text order, we know where we have to set the bit in BV_ℓ. We set it accordingly and update the starting position for characters with the same bit prefix (lines 10 and 11). This requires no additional space and $\mathcal{O}(n)$ time for each of the $\lceil \lg \sigma \rceil$ levels.

Single Scan. Right now, we scan the text once for each level of the wavelet tree. For each level, we scan the text from left to right and set bits at the corresponding positions in the bit vector of the level. We can reduce the number of accesses to the text if we compute all levels during a single scan of the text. To this end, we first compute the histograms of all levels at the same time. Then, we compute the starting positions for all intervals on all levels. Using these starting positions, we can fill the bit vectors by scanning the text once again and considering all bit prefixes of each character we read. This increases the required space to $2\sigma\lceil \lg n \rceil$ bits compared to seq.pc, as we must store all histograms at the same time. However, the asymptotic running time remains the same. This variant reduces the number of scans of the text but also increases the number of cache misses, as we access one bit in each bit vector. We denote this extended version of seq.pc by *seq.pc.ss*.

Lemma 3.1. *Algorithm seq.pc computes the wavelet tree of a text of length n over an alphabet of size σ in $\mathcal{O}(n \lg \sigma)$ time using $\sigma\lceil \lg n \rceil$ bits of space in addition to the input and output. Algorithm seq.pc.ss computes the wavelet tree in the same asymptotic time, requiring $2\sigma\lceil \lg n \rceil$ bits of space in addition to the input and output.*

3.2.2 Prefix Sorting

Our next wavelet tree construction algorithm (*seq.ps*, Algorithm 3.2) is very similar to seq.pc, as we also compute the wavelet tree bottom-up and compute the histogram of the characters first. The algorithms only differ in lines 9–11. Before, we scanned the text and set this bits in the bit vector according to the Borders array. Now, for each level ℓ, we use Counting sort (line 9) with the length-ℓ bit prefixes as keys to sort the text, such that we can fill the bit vector from left to right (line 11). Since Counting sort requires $\mathcal{O}(n)$ time, given the Borders array, the running time does not differ with respect to seq.pc or seq.pc.ss. However, we cannot overwrite the text, as we compute the wavelet tree bottom-up and would lose information of the text order otherwise. Therefore, we must always keep the original text. We compute the bit vectors from the sorted text and *stable* Counting sort requires additional $n\lceil \lg \sigma \rceil$ bits to store the sorted text [Sed98, p. 300]. This leads to the following running time and memory requirements:

Lemma 3.2. *Algorithm seq.ps computes the wavelet tree of a text of length n over an alphabet of size σ in $\mathcal{O}(n \lg \sigma)$ time using $n\lceil \lg \sigma \rceil + \sigma\lceil \lg n \rceil$ bits of space in addition to the input and output.*

Our first algorithms seq.pc and seq.pc.ss compute the bits of each level of the wavelet tree in text order, which results in random access on the bit vectors. With seq.ps, we have the sequential variant of a wavelet tree construction algorithm that is easier to parallelize than seq.pc, because the random access happens during the sorting of the text (line 9). There, we access bytes, not bits. Thus, we can only cause false sharing, but not race conditions, as each character is written exactly once by one processing element. We describe the parallel version of seq.ps in Section 3.3.2.

Algorithm 3.2. Wavelet tree construction with prefix sorting (seq.ps)

Input : Text T of length n and the alphabet size σ.
Output : A bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$ of the wavelet tree.

```
1  for i = 0 to n − 1 do
2  |  Hist[T[i]]++  // Compute histogram of the text (as basis for all other histograms).
3  |  BV_0[i] = bit(0, T[i])  // Fill first level's bit vector (characters' MSB in text order).
4  for ℓ = ⌈lg σ⌉ − 1 to 1 do  // Construct other levels of the wavelet tree bottom-up.
5  |  for i = 0 to 2^ℓ − 1 do  // Compute new histogram based the previous level's one.
6  |  |  Hist[i] = Hist[2i] + Hist[2i + 1]  // Update the histogram in-place.
7  |  for i = 1 to 2^ℓ − 1 do  // Get starting positions of intervals from new histogram.
8  |  |  Borders[i] = Borders[i − 1] + Hist[i − 1]  // Update the positions in-place.
9  |  T' = CountingSort(T, Borders, ℓ)  // Sort T using length-ℓ bit prefixes as keys.
10 |  for i = 0 to n − 1 do  // Scan the sorted text from left to right.
11 |  |  BV_ℓ[i] = bit(ℓ, T'[i])  // Set the bits in the bit vector from left to right.
```

3.2.3 Adaption to the Wavelet Matrix

When comparing the bit vectors of the (level-wise) wavelet tree and the wavelet matrix at level ℓ, we see two similarities. First, both bit vectors contain the ℓ-th MSB of each character of T and second, the bits are grouped in intervals with respect to the bit prefix of size $\ell - 1$ of the corresponding character. Within those intervals, the represented characters appear in the same order. Thus, the number, the sizes, and the content of the intervals are the same for the wavelet tree and matrix. Hence, the only difference is the *position* of the intervals within each level, see Figure 2.4.

At level ℓ, the intervals in BV_ℓ of a wavelet tree occur in increasing order with respect to the bit prefixes of size ℓ of the characters in T, i. e., the first interval corresponds to characters with bit prefix $(0^\ell)_2$, the second one to characters with bit prefix $(0^{\ell-1}1)_2$, and so on. On the other hand, the intervals in BV_ℓ of a wavelet matrix occur in increasing order with respect to the bit-reversal permutation ρ_ℓ of the characters in T. The first interval still corresponds to characters with bit prefix $(0^\ell)_2$, but the interval corresponding to characters with bit prefix $(0^{\ell-1}1)_2$ is the $(2^{\ell-1}+1)$-th interval.

All our previously described wavelet tree construction algorithms (seq.pc, seq.pc.ss, and seq.ps) can easily be adjusted to compute the wavelet matrix instead of the wavelet tree. To this end, we only have to change the computation of $\mathsf{Borders}$, see line 8 in Algorithm 3.1 (seq.pc and seq.pc.ss) and line 8 in Algorithm 3.2 (seq.ps), since we store the starting positions of the intervals in $\mathsf{Borders}$. The change is also minor: we compute the starting positions of the intervals using the bit-reversal permutation, i. e., the lines mentioned above are changed to $\mathsf{Borders}[\rho_\ell[i]] = \mathsf{Borders}[\rho_\ell[i-1]] + \mathsf{Hist}[\rho_\ell[i-1]]$. Then, the resulting starting positions of the intervals for bit prefixes are in bit reversal permutation order, i. e., the starting positions of the intervals for a wavelet matrix.

Therefore, all our wavelet matrix construction algorithms have the same running time and memory requirements as their wavelet-tree-constructing counterparts.

3.2.4 Experimental Evaluation

We implemented all our sequential wavelet tree and wavelet matrix construction algorithms. The code is available at `www.kurpicz.org/wavelet`. We use the hardware setup described in Section 1.4.1 and conducted the experiments on LiDO.big nodes. The code was compiled using GCC 7.3.0 with flags `-O3` and `-march=native`. While more recent compiler versions would be available, some parallel algorithms that we compare our algorithms with in Section 3.3.5 require Cilk Plus, which was removed from GCC starting with version 8.0.0, and we wanted to compile all code using the same compiler. Our inputs are prefixes of the texts described in Section 1.4.2.

We also include the following wavelet tree construction algorithms, which we also described in more detail in Section 2.5.1, in our experiments: (i) seq.naive is the naive wavelet tree construction algorithm based on sorting, (ii) seq.serial was the previously fastest sequential wavelet tree construction algorithm [Shu15], and (iii) seq.sdsl is part of the frequently used Succinct Data Structure Library (SDSL) [Gog+14a]. One wavelet tree (and wavelet matrix) construction algorithm that we could not include here is that of Kaneta [Kan18], because the code is neither publicly available, nor could it be provided by the author due to licensing issues. However, they only report small improvements (on some inputs) while requiring more memory.

To be consistent with the already existing and evaluated wavelet tree construction algorithms listed above, we first compute the effective alphabet from the input text, start the timer, compute the wavelet tree (or wavelet matrix) for the text over the effective alphabet, and stop the timer after all bit vectors have been computed. For easier distinction between our sequential wavelet tree and wavelet matrix construction algorithms, we mark the wavelet matrix construction algorithms with a *.wm*-suffix in our plots. The rest of the name is identical.

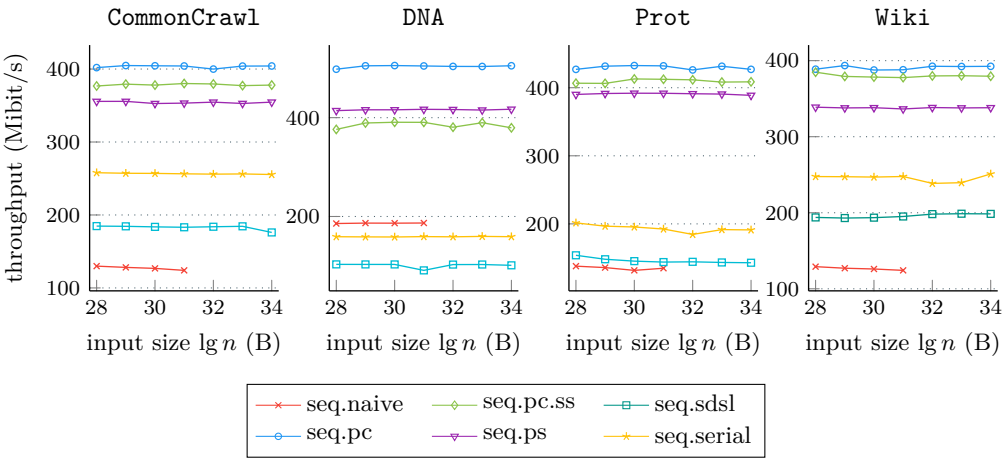


Figure 3.2. Throughput of the sequential wavelet *tree* construction algorithms.

Construction Time. We measured the construction time of our wavelet tree and wavelet matrix construction algorithms on different input sizes ranging from 256 MiB to 16 GiB, and show the resulting throughput (number of bits computed per second) in Figures 3.2 and 3.3. (Remember that timing starts as soon as the effective alphabet has been loaded into main memory and ends as soon as all bit vectors have been computed.) The time used to compute the throughput is the median running time of five executions. In addition, we set a time limit of 2 hours for the five executions in addition to another one to compute the memory peak. Missing data is either due to exceeding the time limit or exhausting the available main memory.

First, we mention that the throughput does not differs a lot for the different inputs, as we show the number of bits computed per second, which is independent of the number of levels. On all inputs seq.pc is the fastest wavelet tree construction algorithm on all inputs. The algorithm seq.pc.ss, which is based on seq.pc, is the second fastest on all inputs but `DNA`. It is slower than seq.pc as it results in more cache misses than seq.pc during the construction when inserting bits in all levels for each character—instead of doing one level at a time. On `DNA`, seq.ps is the second fastest algorithm and seq.pc.ss is the third fastest algorithm, and on all inputs but `DNA`, seq.ps is the third fastest wavelet tree construction algorithm. The previously fastest algorithm seq.serial is slower than our new three algorithms and also slower than seq.naive on `DNA`. On inputs of size 16 GiB, seq.pc is 1.56 times (`Wiki`), 1.58 times (`CommonCrawl`), 2.23 times (`Prot`), and 3.17 times (`DNA`) faster than seq.serial, the previously fastest wavelet tree construction algorithm.

Our results are similar for the wavelet-matrix-computing counterparts seq.pc.wm, seq.pc.ss.wm, and seq.ps.wm of the algorithms analyzed above. On average, their running time is only 0.0061 % slower. This is due to the computation of the histograms that we have to compute in bit reversal permutation order.

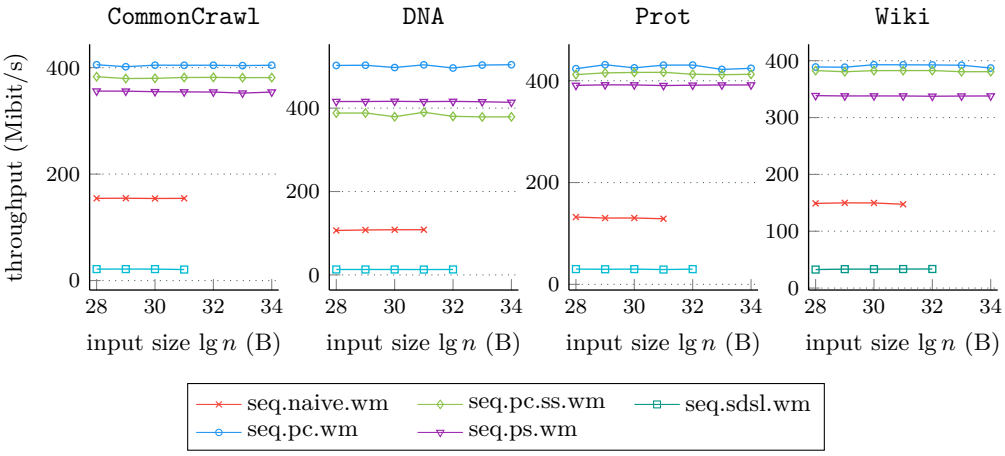


Figure 3.3. Throughput of the sequential wavelet *matrix* construction algorithms.

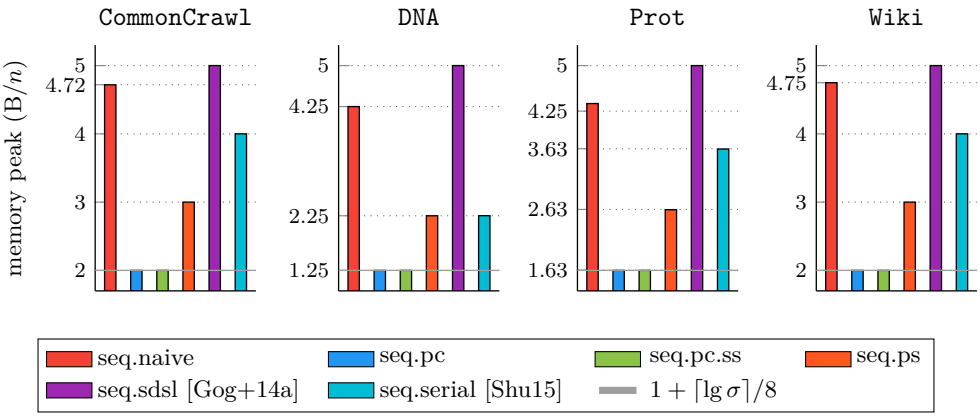


Figure 3.4. Snapshot of the memory peaks of sequential wavelet *tree* construction algorithms for $n = 2^{31}$. We also depict is the memory required to store the text and the wavelet tree ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character).

Memory Peak. The measured memory peaks of our wavelet tree and wavelet matrix construction algorithms that we present in Figures 3.4 and 3.5 is normalized by the input size. We only give the memory peaks for inputs of size 2 GiB, because (i) since we normalize the memory peaks, they are independent of the input size and (ii) this is the maximum input size that all algorithms can process given the time and memory constraints described earlier. We do not consider texts with large alphabets as none of the algorithms can handle those as implemented.

Our algorithms seq.pc and seq.pc.ss have the smallest memory peak—they only require the space for the input, output, and histograms—matching the theoretical analysis. Due to the alphabet size, the histograms only require up to 2 KiB *in total*, which is $9.54 \cdot 10^{-7}$ Bytes per character of the 2 GiB input that is used here. Matching its theoretical analysis, seq.ps requires exactly $n \lg \sigma$ bits more than seq.pc and seq.pc.ss. Hence, it requires 1.5 times (`CommonCrawl` and `Wiki`), 1.6 times (`Prot`), and 1.8 times (`DNA`) more memory than our other two wavelet tree construction algorithms.

The previously fastest sequential wavelet tree construction algorithm seq.serial requires 1.8 times (`DNA`), 2 times (`CommonCrawl` and `Wiki`), and 2.23 times (`Prot`) as much memory as seq.pc and seq.pc.ss. This makes seq.pc not only the fastest but also the most memory efficient wavelet tree and wavelet matrix construction algorithm.

Above, we only analyze the memory peak of the wavelet tree construction algorithms, as *our* wavelet matrix construction algorithms have exactly the same memory peaks as the corresponding wavelet tree construction algorithms. The naive construction algorithm seq.naive.wm requires around 5 % more memory than seq.naive on all instances but `DNA` where they have the same memory peak. The other algorithm, seq.sdsl.wm, requires 1.4 times (`DNA`, `Prot`, and `Wiki`) and 1.6 times (`CommonCrawl`) more memory than its wavelet-tree-constructing counterpart.

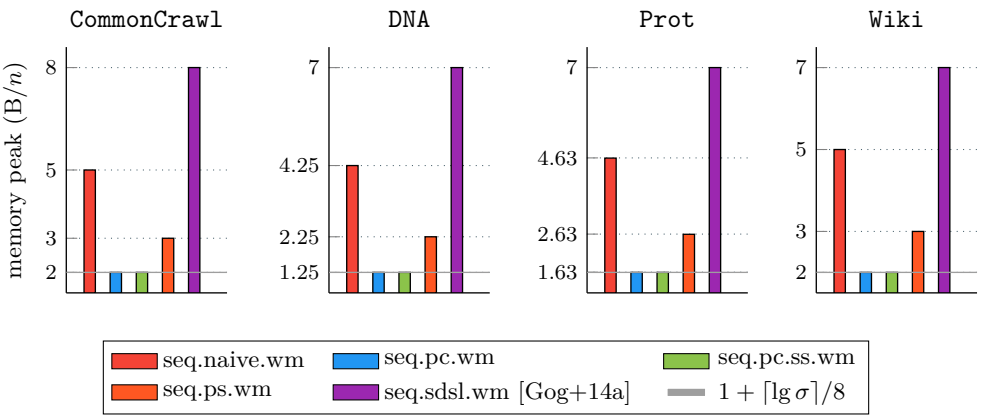


Figure 3.5. Snapshot of the memory peaks of sequential wavelet *matrix* construction algorithms for $n = 2^{31}$. We also depict is the memory required to store the text and the wavelet matrix ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character).

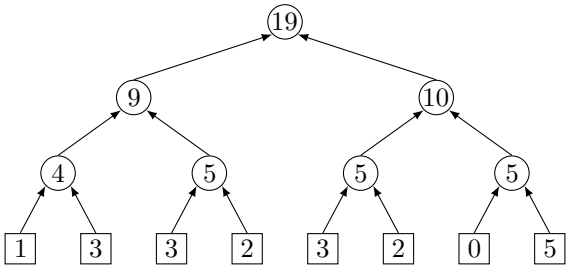
3.3 SHARED MEMORY CONSTRUCTION

Now, we look at shared memory parallel wavelet tree construction algorithms. First, in Section 3.3.1, we describe a naive parallelization of seq.pc that only scales up to $\lceil \lg \sigma \rceil$ processing elements. Therefore, we also parallelize seq.ps in Section 3.3.2, which scales better. In Section 3.3.3, we look at the meta-approach *domain decomposition* [FS+17; Lab+17], where we first construct partial wavelet trees for consecutive slices of the text. We obtain the wavelet tree by merging the partial ones. Similarly to the last section, we show that our parallel wavelet tree construction algorithms can be adapted to compute the wavelet matrix instead (Section 3.3.4). Finally, in Section 3.3.5, we give an experimental evaluation of these algorithms.

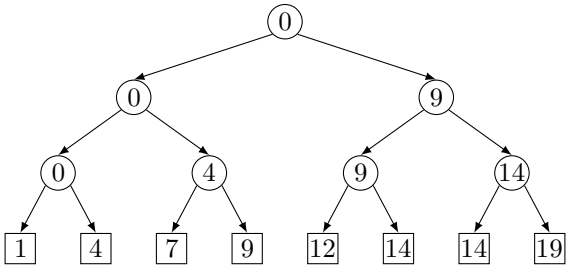
Parallel Histogram Computation

Before we look into the parallelization of our wavelet tree and wavelet matrix construction algorithms, we first take a look at the *parallel* computation of histograms and border positions, which heavily rely on computing prefix sums. It is well known that we can compute the prefix sum of σ elements in parallel in time $\mathcal{O}(\lg \sigma)$ with $\mathcal{O}(\sigma)$ work, e. g., [JáJ92, p. 45].

This running time can be achieved using a two-phase algorithm based on a merge tree. For simplicity let us assume that σ is a power of two. Each of the n elements is represented by one leaf. Then, each other node contains the sum of its children. The values of all parents that have children that contain a value can be computed in parallel—going from the leaves to the root. This happens $\mathcal{O}(\lg \sigma)$ times, as this is the height of the tree. Hence, it is easy to see that we can fill all nodes in $\mathcal{O}(\lg \sigma)$ time and $\mathcal{O}(\sigma)$ work. We give an example of this step in Figure 3.6a.



(a) Result of the first phase of parallel prefix sum computation.



(b) Result of the second phase of parallel prefix sum computation, including updated leaves.

Figure 3.6. Two phases that we use to describe the computation of the parallel prefix sum for $[\,1, 3, 3, 2, 3, 2, 0, 5\,]$.

Next, we set the value of the root to 0. Then, the value of each right child is set to the sum of its parent and its left sibling, and each left child's value becomes the value of its parent. In both cases, we add the value if the child is a leaf, see Figure 3.6b. As the previous phase, this requires $\mathcal{O}(\lg \sigma)$ time and $\mathcal{O}(\sigma)$ work.

However, in the following, we often want to compute a prefix sum of $n = p\sigma$ elements, where p is the number of processing elements and σ is the size of the alphabet of a text (but we could use any other value, too). Now, the running time and work of parallel prefix sum depends on the number of processing elements, because increasing the number of processing elements also increases the number of elements that we want to compute the prefix sum for. In this case, we can compute the parallel prefix sum of $p\sigma$ elements using p processing elements as follows.

First, we split the problem into p sub-problems. We denote these elements by $\mathsf{X}_i[0..\sigma - 1]$ for processing element $i \in [0, p)$. We then compute the prefix sums from X_i. As the computation can be done in parallel, this first step requires $\mathcal{O}(\sigma)$ time and $\mathcal{O}(p\sigma)$ work. Since all these prefix sums are computed independently, we need another step to compute the final prefix sum over all elements. To this end, we consider the values $\mathsf{X}_i[\sigma - 1]$, i. e., the rightmost elements of each sub-problem. There are p such elements (one for each sub-problem) and now compute the prefix sum for

$\mathsf{X}' \coloneqq \mathsf{X}_0[\sigma-1]\ldots\mathsf{X}_{p-1}[\sigma-1]$. Using a simple parallel prefix sum, we can do so in $\mathcal{O}(\lg p)$ time and $\mathcal{O}(p)$ work. Finally, we have to add the result of this prefix sum to all elements in X_i, i. e., for all $i \in [1,p)$ and $j \in [0,\sigma)$ we add $\mathsf{X}'[i-1]$ to $\mathsf{X}_i[j]$. All in all, we achieve the following time and work for parallel prefix sums, which occurs multiple times in the analysis of the following algorithms.

Observation 3.1. *The parallel prefix sum over $p\sigma$ elements requires $\mathcal{O}(\sigma + \lg p)$ time and $\mathcal{O}(p\sigma)$ work using p processing elements.*

Note that Shun [Shu17, Theorem 4.2] give an analysis for parallel prefix sum over $\mathcal{O}(P\sigma)$ elements for some P (which does not have to be the number of processing elements) that requires $\mathcal{O}(\lg P)$ time and $\mathcal{O}(P\sigma)$ work. This is because they can independently compute $\mathcal{O}(\sigma)$ prefix sums for P elements, as they are interested in pointer-based wavelet trees. Thus they do not have to compute the starting positions of the intervals in the bit vectors.

3.3.1 Parallel Prefix Counting

It is embarrassingly easy to parallelize seq.pc (Algorithm 3.1) such that each processing element computes one level of the wavelet tree, see Figure 3.7. To this end, we first compute the histogram that is then used to compute the starting positions of the intervals of each level (both in parallel). On the ℓ-th level, the starting positions require $2^\ell \lceil \lg n \rceil$ bits of space. With the starting positions, we can compute all bit vectors in parallel. We denote the resulting parallel algorithm by *par.pc*.

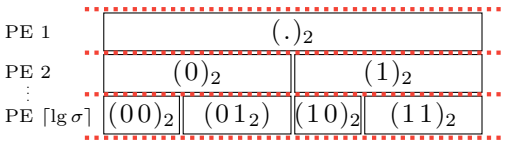


Figure 3.7. Each PE computes one level.

Lemma 3.3. *Algorithm par.pc computes the wavelet tree in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg \sigma)$ work requiring $\sigma\lceil \lg n \rceil$ bits of space in addition to the input and output.*

The disadvantage of par.pc is that it cannot efficiently use more than $\lceil \lg \sigma \rceil$ processing elements. To use more processing elements, instead of parallelizing level-wise, we could do the following. Each of the p processing elements gets a slice of the text of size $\Theta(n/p)$ and computes the corresponding bits in the bit vectors on *all* levels. On level ℓ, each procssing element c first computes its *local* histogram $\mathsf{Hist}_c[0,\sigma)$ according to the length-ℓ bit-prefixes of the input characters. Using a parallel zero-based prefix sum, these local histograms are then combined such that in the end, each processing element knows where to write its bits (arrays $\mathsf{Borders}_c[0,\sigma)$ for $c \in [0,p)$). As in the sequential algorithm, the final writing is then accomplished by scanning the local slice of the text from left to right, writing the bits to their correct places in BV_ℓ, and incrementing the corresponding value in $\mathsf{Borders}_c$. This comes with the problem that two or more processing elements may want to concurrently write bits to the same computer word (race conditions). To avoid this, one would have to implement mechanisms for exclusive writes, which would result in unacceptably high running times.

3.3.2 Parallel Prefix Sorting

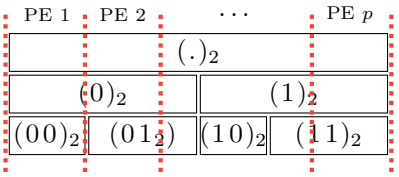


Figure 3.8. Each PE computes the same slice of the bit vector on all levels.

Instead of having each processing element write randomly to each bit vector, we want each processing element to be responsible for the same slice on each level of the wavelet tree, see Figure 3.8. Those slices have size $\Theta(n/p)$. To this end, we parallelize seq.ps (Algorithm 3.2), which has also been the main motivation for the sequential variant of the algorithm. Now, we *globally* sort the input text in parallel. The resulting sorted text T_{sorted} is then again split into parts of size $\Theta(n/p)$. Then, each processing element scans its local slice from left to right and writes the corresponding bits to the bit vector. Note that this is different from *domain decomposition*, a popular approach for parallel wavelet tree construction [FS+17; Lab+17], which we discuss in Section 3.3.3. To avoid race conditions and false sharing, we make sure that the size of each slice of the text is a common multiple of the cache lines' length and the size of a computer word.

The resulting parallel wavelet tree construction algorithm (*par.ps*, Algorithm 3.3) works as follows: First, each of the p processing elements computes the local histogram (Hist_c for $c \in [0, p)$) of its part of T and, at the same time, fills BV_0 (lines 3 and 4). Then, we compute the local starting positions ($\mathsf{Borders}_c$ for $c \in [0, p)$), using the parallel zero-based prefix sum of $\mathsf{Borders}_0[0], \mathsf{Borders}_1[0], \ldots, \mathsf{Borders}_{p-1}[0], \ldots, \mathsf{Borders}_0[\sigma - 1], \ldots, \mathsf{Borders}_{p-1}[\sigma - 1]$ (line 5). Using these starting positions, we can extract the starting positions for all other levels by choosing every $2^{\lceil \lg \sigma \rceil - \ell}$-th entry on level ℓ (line 9). All in all, this requires $\mathcal{O}(n/p + \lg(p\sigma))$ time, $\mathcal{O}(n + p\sigma)$ work and $p\sigma\lceil \lg n \rceil$ bits of space. Using this information (Hist_c and $\mathsf{Borders}_c$), we can compute the corresponding values of $\mathsf{Borders}_c$ for all levels $\ell \in [1, \lceil \lg \sigma \rceil)$ in time $\mathcal{O}(\sigma/p)$.

For each level (loop starting at line 6), the time and work required are the same as during the first step. There is no additional space required since we reuse the space used during the previous iteration. For the temporary starting positions $\mathsf{Borders}'_c$ we can use the space occupied by Hist. To sort the text, we use the local starting positions to represent the intervals in counting sort (line 10). Storing the sorted text requires additional $n\lceil \lg \sigma \rceil$ bits of space, which we reuse at each level. After sorting the text, each processing element can fill BV_ℓ accordingly (line 13).

Lemma 3.4. *Algorithm par.ps computes the wavelet tree of a text of length n over an alphabet of size σ in $\mathcal{O}((n/p)\lg\sigma + \sigma + \lg p)$ time and $\mathcal{O}(n\lg\sigma + p\sigma))$ work requiring $n\lceil\lg\sigma\rceil + p\sigma\lceil\lg n\rceil$ bits of space in addition to the input and output using $p < (n\lg\sigma)/\sigma$ processing elements.*

This algorithm can efficiently use up to $p \leq n/\sigma$ processing elements. Using that many processing elements yields $\mathcal{O}(n\lg\sigma)$ work with $\mathcal{O}(\lg\sigma\,(\sigma + \lg n))$ time. Employing more processing elements only increases the required work, without achieving a better running time. In theory, better work could be achieved by using word packing techniques, similar to [Bab+15; Kan18; Mun+16].

Algorithm 3.3. Parallel wavelet tree construction with prefix sorting (par.ps)

Input : Text T of length n and the alphabet size σ.
Output : A bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$ of the wavelet tree.

```
1  parfor c = 0 to p − 1 do
2  |  for i = c n/p to (c + 1) n/p do
3  |  |  Hist_c[T[i]]++            // Compute histogram of the local part of the text.
4  |  |  BV_0[i] = bit(0, T[i])    // Fill first level's bit vector in parallel (characters' MSB).
5  |  Borders_c = Parallel zero-based prefix sum on Hist_c   // Global starting positions.
6  for ℓ = ⌈lg σ⌉ − 1 to 1 do                  // For each level (from the last to the second).
7  |  parfor c = 0 to p − 1 do // Get histogram for local part and current level in parallel.
8  |  |  for i = 0 to 2^ℓ − 1 do
9  |  |  |  Borders'_c[i] = Borders_c[2^(⌈lg σ⌉−ℓ) i]
10 |  T_sorted = parallel CountingSort(T, Borders'_c) // Use starting positions to sort text.
11 |  parfor c = 0 to p − 1 do
12 |  |  for i = c n/p to (c + 1) n/p do   // Fill bit vector left to right using the sorted text.
13 |  |  |  BV_ℓ[i] = bit(ℓ, T_sorted[i])
```

Using sorting for the parallel construction of wavelet trees has already been considered by Shun [Shu15] (*par.sort*). We gave a short description of their approach in Section 2.5.2. The main difference between their algorithm and ours is that we construct the wavelet tree bottom-up whereas they construct it tree top-down. This allows us to save one scan per level, which does not affect the theoretical running time, but is a huge improvement in practice, as we show in the evaluation in Section 3.3.5.

3.3.3 Domain Decomposition

The *domain decomposition* [FS+17; Lab+17] is a popular meta-approach for parallel wavelet tree construction. Here, each processing element gets a consecutive slice of the text of size $\Theta(n/p)$ and computes a *partial* wavelet tree for that slice. We can use any sequential version of our wavelet tree construction algorithms, e. g., seq.pc, seq.pc.ss, or seq.ps (see Section 3.2), to compute the partial wavelet trees. The final wavelet tree is then computed by merging, which is more like a concatenation of intervals in the bit vectors, all partial wavelet trees in parallel, which is described below. We call this parallel algorithm *par.dd.pc*, *par.dd.pc.ss*, and *par.dd.ps*, depending on the sequential algorithm used to compute the partial wavelet trees.

To merge the partial wavelet trees, we only have to concatenate the intervals of all partial wavelet trees that correspond to the same bit prefix and store these concatenations (in the same order the corresponding bit prefixes occur in the partial wavelet trees) at the correct level of the merged wavelet tree, see Figure 3.9. We can do so in parallel by using the starting positions of the intervals of the partial wavelet trees that have already been computed during their construction. To this end, a zero-based prefix sum computes the starting positions of the intervals in the merged wavelet tree. Then, each processing element writes its intervals at the corresponding positions.

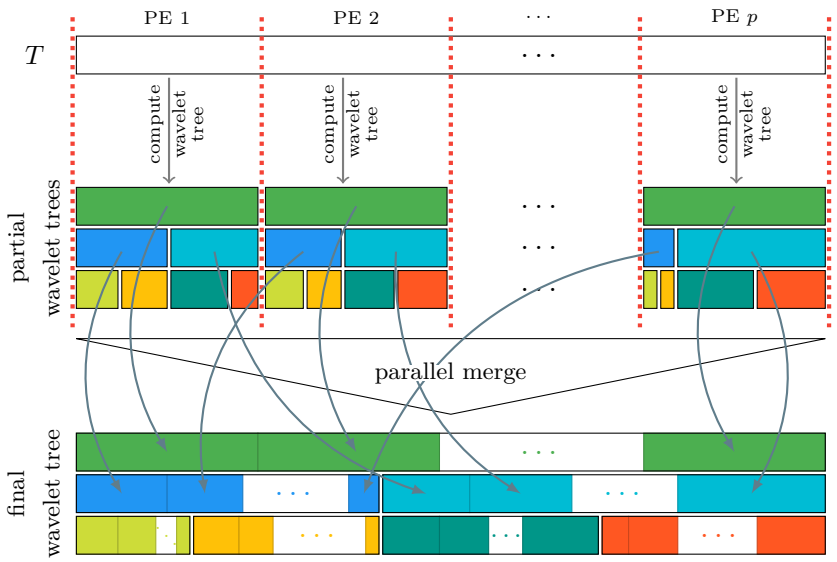


Figure 3.9. Each PE considers a part of the text and computes a *partial* wavelet tree for that tree, in parallel. Then, all partial wavelet trees are merged—again, in parallel. To merge the partial wavelet trees, we concatenate the intervals representing the same characters on each level. To highlight this, we colored the intervals and point from the partial wavelet trees to the merged results (for the first two levels).

Here, we also avoid race conditions by choosing the starting positions of the merged intervals according to the width of a computer word. As the computation of the partial wavelet trees can be parallelized perfectly, we only require one parallel prefix sum, and the merging is one parallel scan of all bit vectors. We do not merge in-place—thus we need another $n\lceil \lg \sigma \rceil$ bits for the final wavelet tree. When computing the partial wavelet trees with seq.ps, we can reuse the space required for sorting the text.

Lemma 3.5. *Algorithms par.dd.pc, par.dd.pc.ss, and par.dd.ps compute the wavelet tree of a text T of length n over an alphabet of size σ in $\mathcal{O}(n/p \lg \sigma + \sigma + \lg p)$ time and $\mathcal{O}(n \lg \sigma + p\sigma)$ work requiring $n\lceil \lg \sigma \rceil + p\sigma \lceil \lg n \rceil$ bits of space in addition to the input and output using $p < (n \lg \sigma)/\sigma$ processing elements.*

3.3.4 Adaption to the Wavelet Matrix

All of our shared memory parallel wavelet tree construction algorithms can be adapted to compute the wavelet matrix instead. We keep our naming scheme from the last section and append the *.wm* suffix to denote wavelet matrix construction algorithms.

We can adapt par.pc (Section 3.3.1) in the same fashion we adapted its sequential

Algorithm 3.4. Parallel wavelet matrix construction with prefix sorting

Input : Text T of length n and the alphabet size σ.
Output : A bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$ of the wavelet matrix.

```
1  parfor c = 0 to p − 1 do
2  |  for i = c n/p to (c + 1) n/p do
3  |  |  Hist_c[T[i]]++            // Compute histogram of the local part of the text.
4  |  |  BV_0[i] = bit(0, T[i])    // Fill first level's bit vector in parallel (characters' MSB).
5  |  Borders_c = Parallel prefix sum w.r.t. ρ_⌈lg σ⌉
6  for ℓ = ⌈lg σ⌉ − 1 to 1 do
7  |  parfor c = 0 to p − 1 do
8  |  |  for i = 0 to 2^ℓ − 1 do
9  |  |  |  Hist_c[i] = Hist_c[2i] + Hist_c[2i + 1]
10 |  Borders_c = Parallel prefix sum w.r.t. ρ_ℓ // w.r.t means in order ρ_ℓ[0], ..., ρ_ℓ[2^ℓ − 1]
   |  T_sorted = CountingSort(T, Borders) parfor c = 0 to p − 1 do
11 |  |  for i = c n/p to (c + 1) n/p do
12 |  |  |  BV_ℓ[i] = bit(ℓ, T_sorted[i])
```

counterpart in Section 3.2.3—using a zero-based prefix sum with respect to ρ_ℓ for level ℓ. This is sufficient, as each bit vector is computed by only one processing element.

To adapt par.ps (Section 3.3.2), we have to adjust the computation of the starting positions because we lose the tree structure when we compute the wavelet matrix, see Algorithm 3.4. Hence, we cannot compute the new starting positions from the old ones (line 9). Instead, we have to update the histogram Hist_c for each processing element c and for each level ℓ (loop starting in line 6). With the updated histogram, we can compute the borders using a zero-based prefix sum with respect to ρ_ℓ, locally on Hist_c. Since we have to compute the prefix sum for each level, we get:

Lemma 3.6. *The wavelet-matrix-constructing counterpart of par.ps, par.ps.wm, requires $\mathcal{O}(\lg \sigma\,(n/p + \sigma + \lg p))$ time and $\mathcal{O}(\lg \sigma(n + p\sigma))$ work to compute the wavelet matrix for a text of size n over an alphabet of size σ using $p < n/\sigma$ processing elements. The space requirements do not change compared to par.ps.*

The parallel wavelet tree construction algorithms using domain decomposition that we described in Section 3.3.3 are also easily adapted. To this end, we only have to use sequential wavelet *matrix* construction algorithms to compute the partial wavelet matrices. There is no need to change the merging of the partial wavelet matrices as long as these intervals occur in bit-reversal permutation order, since we only concatenate intervals on each level. See also Figure 3.9, where the merging is independent of the content of the interval (or the considered bit prefix), as we only need the starting positions of the intervals in the partial wavelet trees or wavelet matrices.

Thus, par.pc.wm and the domain decomposition algorithms have the same running time, work, and memory requirements as their wavelet-tree-constructing counterparts. For par.ps.wm, we give the running time in Lemma 3.6.

3.3.5 Experimental Evaluation

We implemented all of our parallel wavelet tree and wavelet matrix construction algorithms that we described in the previous sections. The code of our algorithms and the ones we compare them with is available at `www.kurpicz.org/wavelet`. We used the hardware setup described in Section 1.4.1, conducted the experiments on LiDO.big nodes, and use (prefixes of) the text described in Section 1.4.2 as inputs. As before, the code was compiled using GCC 7.3.0 with flags `-O3` and `-march=native`, because even though more recent compiler versions were available at the time we conducted these experiments *all* publicly available parallel wavelet tree construction algorithms not implemented by us require Cilk Plus, which was removed from GCC starting with version 8.0.0. Our algorithms express parallelism using OpenMP 4.5.

In addition to our algorithms that are described in this section, we also include the following algorithms in our experiments that we described in Section 2.5.2:

par.dd is a parallel wavelet tree construction algorithm using domain decomposition by Shun [Shu17],

par.level constructs the wavelet tree level-by-level in parallel and does not rely on sorting by Shun [Shu15],

par.sort is a parallel algorithm based on sorting and computes the wavelet tree top-down by Shun [Shu15], and finally

par.rec is the previously fastest parallel wavelet tree construction algorithm that splits the text by Labeit et al. [Lab+17].

As for the sequential algorithms, we want to keep our evaluation coherent with already existing experiments. Therefore, we first compute the effective alphabet from the input text, start the timer, compute the wavelet tree (or wavelet matrix) for the text over the effective alphabet, and stop the timer as soon as all bit vectors of the wavelet tree (or wavelet matrix) have been computed. Again, we do not compute any auxiliary binary rank and select data structures.

Construction Time. First, we report the construction time of the wavelet tree and wavelet matrix construction algorithms. Here, we focus on the results of our wavelet tree construction algorithms, which are depicted in Figure 3.10, as running times and scalability of our wavelet matrix construction algorithms are nearly identical, see Figure 3.11. In addition, the throughput of *all* algorithms does not depend on the size of the input, as it is nearly identical regardless of whether we use inputs of size 256 MiB, 512 MiB, or 1024 MiB per processing element. Therefore, in the following, we only discuss data from experiments where we use inputs of size 1024 MiB.

Then, we want to mention our slowest parallel algorithms par.pcand par.pc.ss. While they do not scale up to 48 processing elements, they do scale up to $\lceil \lg \sigma \rceil$ processing elements (two for `DNA`, four for `Prot`, and eight for `CommonCrawl` and `Wiki`), which is as expected and matches the theoretical analysis. The last algorithm that does not scale well is par.ps, which is due to the overhead of stable parallel sorting.

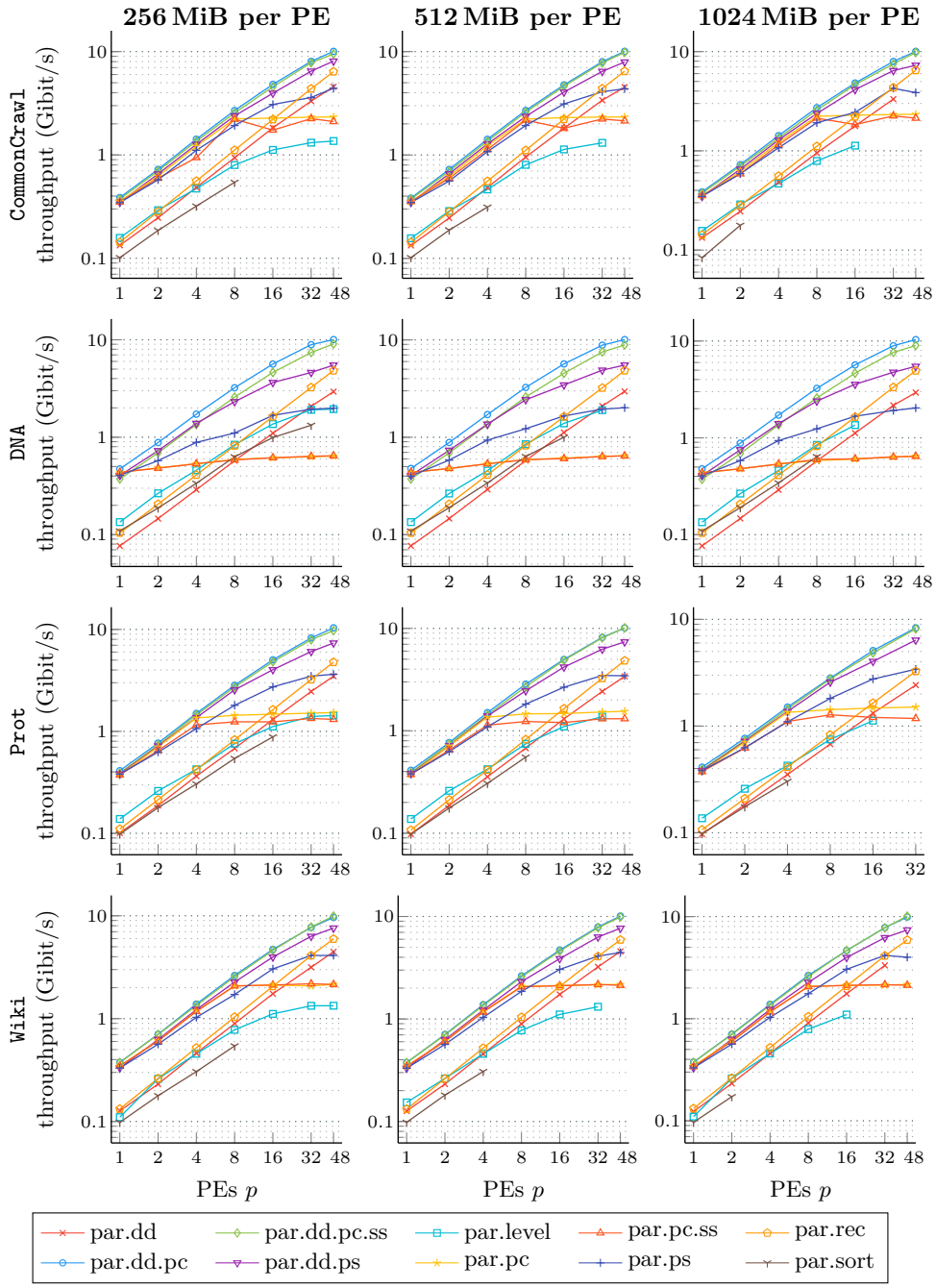


Figure 3.10. Weak scaling parallel wavelet *tree* construction experiments.

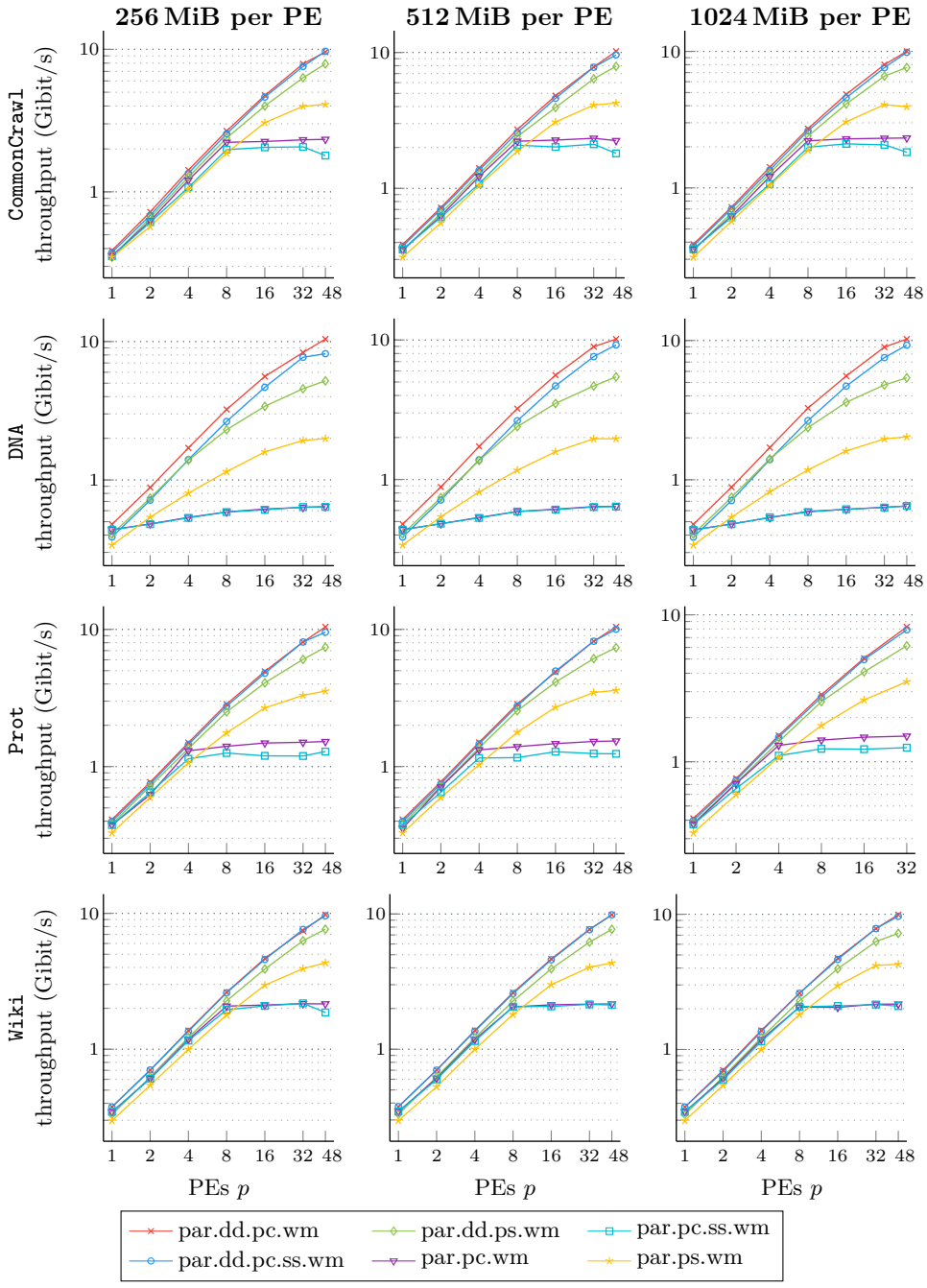


Figure 3.11. Weak scaling parallel wavelet *matrix* construction experiments.

The other three parallel wavelet tree construction algorithms are all based on the domain decomposition and share the same parallel merge routine. Therefore, unsurprisingly, par.dd.pc is the fastest wavelet tree construction algorithm for all sizes, number of PEs, and inputs but `Wiki`. On `Wiki`, the second fastest (on all other inputs) algorithm par.dd.pc.ss is faster. This is no surprise considering that the algorithms used to compute the partial wavelet tree in the domain decomposition are of similar speed with seq.pc being faster than seq.pc.ss (see Section 3.2.4).

Now, let us focus on the throughput using one and 48 processing elements on 1024 MiB input per processing element. We report detailed results in Table 3.1. Our three domain decomposition algorithms are the three fastest algorithms on all inputs. On all inputs but `DNA`, par.rec is the fourth fastest parallel wavelet tree construction algorithm. It was previously the fastest one.

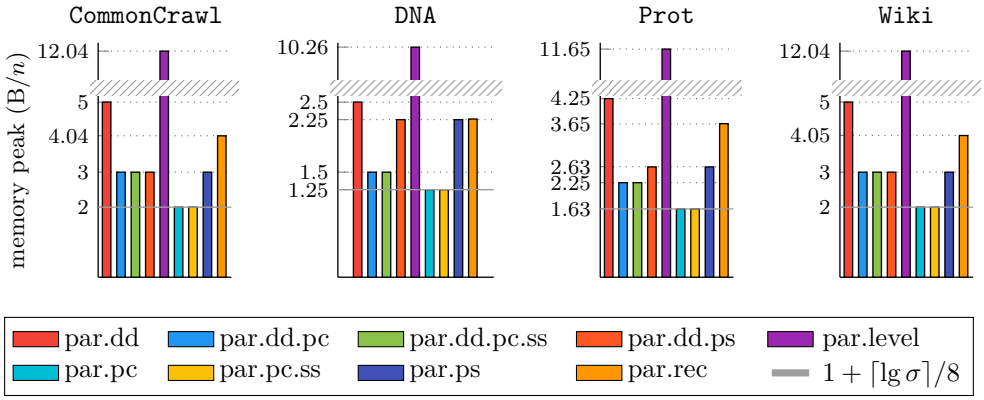
While our algorithms obtain the highest throughput of all parallel wavelet tree construction algorithms, they do not have the highest speedup. To be precise, par.rec achieves a speedup of 45.5, 27.4, 30.5, and 44.6 (on `CommonCrawl`, `DNA`, `Prot`, and `Wiki`), whereas our algorithm with the best speedup only achieves a speedup of 26.2, 24.1, 20.8, and 27.1 (on the same inputs) using 48 processing elements. This is consistent with the results previously reported [Fis+18]. However, due to further engineering of our algorithms, i. e., reducing false sharing, par.dd.pc and our other domain decomposition algorithms are now faster up to 48 cores.

On a related note, we want to mention the COST [McS+15] (see Section 1.3 for a description) of *all* our parallel wavelet tree and wavelet matrix construction algorithms is 2, as when using two processing elements, all parallel versions are faster than the fastest sequential one (which is the same as executing our parallel algorithm using only one processing element). The previously fastest algorithm par.rec has COST 4 on `CommonCrawl` and `Wiki`, and a COST of 8 on `DNA`, which shows that it has a higher overhead, which makes it easier to achieve the better speedup.

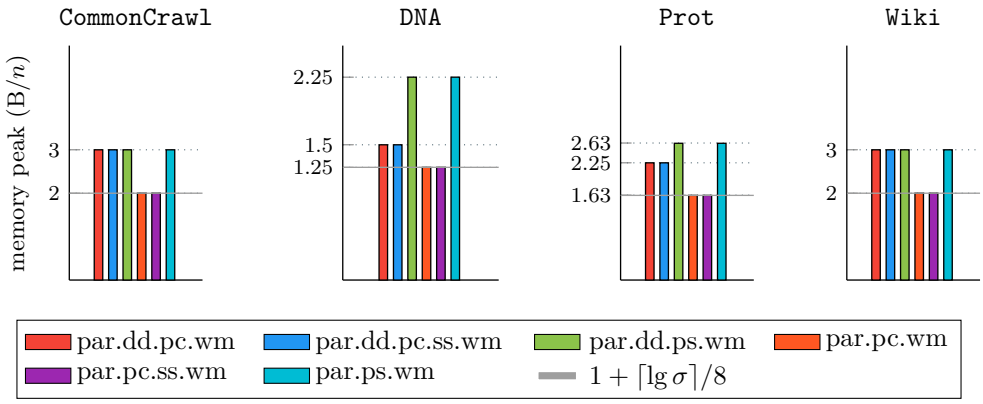
Memory Peak. Now, we take a look at the memory consumption of the parallel wavelet tree and wavelet matrix construction algorithms. We give the results in Figure 3.12. There, we only give the results for algorithms using 48 processing elements and processing 256 MiB input per processing element. This is because (1) the memory requirements increase when increasing the number of processing elements, making this the harder case, and (2) due to their memory consumption, not all algorithms (par.dd, par.sort, and par.level) finish the experiment when using 48 processing elements. Using only 256 MiB input per processing element allows us to include all algorithms but par.sort in the plots.

The results of our wavelet tree and wavelet matrix construction algorithms match our theoretical analysis. Our most memory efficient algorithms are par.pc and par.pc.ss, which require nearly no additional space in addition to the input and output. Actually, they only require the space for one histogram per level of the wavelet tree or wavelet matrix. Next, par.dd.pc and par.dd.pc.ss require only space to merge the wavelet tree (or wavelet matrix). For large alphabets (`CommonCrawl` and `Wiki`), they require as

much space as par.ps and par.dd.ps, which require more space on DNA and Protthan par.dd.pc and par.dd.pc.ss. The previously fastest wavelet tree construction algorithm par.rec requires up to twice as much memory as our most memory efficient algorithms on all instances but DNA, where it requires only 1.8 times as much. Hence, our new parallel wavelet tree and wavelet matrix construction algorithms are the most memory efficient ones. However, the fastest (and best scaling ones) are not the most memory efficient ones, but the fastest one (par.dd.pc) only requires up 1.5 times more memory.



(a) Memory peaks for parallel wavelet tree construction.



(b) Memory peaks for parallel wavelet matrix construction.

Figure 3.12. Memory peaks of the parallel wavelet *tree* and wavelet *matrix* construction algorithms using 48 PEs and 256 MiB input per PE. We use small inputs to have results for par.dd, still we could not include par.sort. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

Table 3.1. Throughput (Gibits/s) of the wavelet tree and wavelet matrix construction algorithms in our weak scaling experiment when using one (t_1) or 48 (t_{48}) PEs and 1024 MiB input per PE. We give the results for 32 PEs (t_{32}) on `Prot`, as this is limited by the text size. Algorithms not included in the listing are not able to compute the wavelet tree or wavelet matrix in this setting. We mark the highest throughput and speedup for each input in bold.

	`CommonCrawl`			`DNA`			`Prot`			`Wiki`		
	t_1	t_{48}	t_{48}/t_1	t_1	t_{48}	t_{48}/t_1	t_1	t_{32}	t_{32}/t_1	t_1	t_{48}	t_{48}/t_1
par.dd				0.08	2.94	38.29	0.10	2.42	24.91			
par.dd.pc	**0.39**	**10.05**	26.04	**0.48**	**10.34**	21.57	**0.41**	**8.33**	20.22	**0.38**	9.89	26.34
par.dd.pc.ss	0.37	9.78	26.18	0.37	8.98	24.08	0.39	8.14	20.76	0.37	**10.12**	27.06
par.dd.ps	0.35	7.28	21.01	0.41	5.48	13.47	0.38	6.39	16.73	0.33	7.40	22.41
par.pc	0.35	2.34	6.67	0.44	0.65	1.48	0.37	1.51	4.02	0.35	2.16	6.27
par.pc.ss	0.35	2.13	6.01	0.44	0.65	1.48	0.37	1.18	3.14	0.34	2.13	6.26
par.ps	0.35	3.86	11.11	0.41	2.04	5.03	0.38	3.41	8.92	0.33	4.00	12.10
par.rec	0.14	6.47	**45.49**	0.10	4.93	**47.44**	0.11	3.26	**30.55**	0.13	5.90	**44.64**
par.dd.pc.wm	**0.39**	**10.02**	25.92	**0.48**	**10.23**	21.39	**0.41**	**8.27**	**20.09**	0.38	**9.92**	**26.46**
par.dd.pc.ss.wm	0.38	9.84	**26.20**	0.39	9.26	**23.92**	0.40	7.90	19.81	**0.38**	9.66	25.71
par.dd.ps.wm	0.35	7.62	21.95	0.41	5.39	13.28	0.38	6.15	16.08	0.33	7.22	21.86
par.pc.wm	0.36	2.32	6.54	0.44	0.65	1.48	0.38	1.49	3.98	0.35	2.17	6.25
par.pc.ss.wm	0.35	1.83	5.15	0.44	0.65	1.49	0.38	1.25	3.32	0.35	2.09	6.05
par.ps.wm	0.31	3.94	12.69	0.34	2.03	5.98	0.33	3.50	10.67	0.30	4.27	14.35

3.4 EXTERNAL MEMORY CONSTRUCTION

The inputs that all wavelet tree and wavelet matrix construction algorithms we have seen so far can process is bounded by the size of the main memory. In this section, we overcome these limitations by introducing different semi-external and fully external memory wavelet tree and wavelet matrix construction algorithms. All these algorithms make use of the bottom-up histogram computation (see Section 3.1), which allows us to reduce the number of scans of the text. This property is especially useful in external memory, since each disk access (even if it is just a scan) is expensive.

3.4.1 Sequential Construction in Semi-External Memory

In this section, we start with a brief discussion of how to adapt the sequential wavelet tree and wavelet matrix construction algorithms from Section 3.2 to the semi-external memory model, which we defined in Section 1.3.4. Remember that in the semi-external memory model; we allow random access on either the input or the output—but not both.

Random Access on the Input. First, we consider a modified and semi-external version of the prefix sorting wavelet tree construction algorithm (seq.ps), see Section 3.2.2. Here, each level of the wavelet tree is written in sequential order, which lets us efficiently stream the bit vectors of the wavelet tree to external memory. Again, we precompute all borders of the intervals.

Then, for each level ℓ, we use counting sort with the length-ℓ bit prefixes as keys to sort the text, such that we can fill the bit vector from left to right. Counting sort requires $\mathcal{O}(n)$ time, given the borders array, hence the running time does not differ from se.pc. Since we require a stable sort, we cannot sort the text in-place [Sed98, p. 300] and thus need additional $n\lceil\lg\sigma\rceil$ bits of space in main memory. We write the output to disk exactly once, and each level is written sequentially. Therefore, the number of I/Os is $\mathrm{scan}(n\lceil\lg\sigma\rceil)$. We call this algorithm *se.ps*.

To overcome the space requirements by sorting, we now describe a new in-place algorithm that re-arranges the text as required by the wavelet tree in $\mathcal{O}(n)$ time. We decompose the text into $\Theta(\sqrt{n})$ blocks of size $\Theta(\sqrt{n})$ and use two buffers of the same size. Then, we separate the text using one buffer for symbols corresponding to a one bit and the other for the other bits. Whenever a buffer is full, we can write it to an already processed part of the text, which has already been written to the buffers. In the end, we have to rearrange the blocks. We denote this variant by *se.ps.ip*. This variant requires less space than se.ps, but due to the in-place re-arranging, it is one of the slowest algorithms (see our evaluation in Section 3.4.4 for details).

Lemma 3.7. *The semi-external algorithms se.ps and se.ps.ip compute the WT of a text of length n over an alphabet of size σ in $\mathcal{O}(n\lg\sigma)$ time using $\mathcal{O}(\mathit{scan}(n\lceil\lg\sigma\rceil))$ I/Os, and $n\lceil\lg\sigma\rceil + \sigma\lceil\lg n\rceil$ (se.ps.ip) and $2n\lceil\lg\sigma\rceil + \sigma\lceil\lg n\rceil$ (se.ps) bits of main memory including input and output, respectively.*

Random Access on the Output. Our second semi-external wavelet tree construction algorithm is the semi-external variant of the single scan prefix counting wavelet tree construction algorithm (seq.pc.ss), see Section 3.2.1. Here, we first compute the histogram for all characters in the text and compute all histograms and interval borders without another scan of the text in $\mathcal{O}(n)$ time, $\text{scan}(n\lceil\lg\sigma\rceil)$ I/Os, and $\sigma\lceil\lg n\rceil$ bits space in main memory, as described in Section 3.1.

Next, we scan the text once again and fill all the bit vectors accordingly using the precomputed borders, i. e., for each symbol, we look at the border for each of the symbol's bit prefixes and set the corresponding bit in each bit vector accordingly (one bit per level) and then we update the borders. This requires $\mathcal{O}(n\lg\sigma)$ time in total for all levels. Setting the bits in the bit vectors still requires random access, which is the reason why this algorithm is only a semi-external memory wavelet tree construction algorithm. Hence, we only read the text from the secondary memory. The number of I/Os is $2\,\text{scan}(n\lceil\lg\sigma\rceil)$. In terms of main memory, we need $n\lceil\lg\sigma\rceil$ bits for the bit vectors of the wavelet tree and $\sigma\lceil\log n\rceil$ bits for histograms that are later used for the starting positions of the intervals. We call this semi-external algorithm *se.pc*.

This algorithm can also be parallelized by parallelizing the computation of the initial histogram and writing the bit vectors for each level in parallel, which scales up to $\lceil\lg\sigma\rceil$ processing elements. We denote this algorithm by *se.par.pc*.

Lemma 3.8. *The semi-external algorithm se.pc computes the wavelet tree of a text of length n over an alphabet of size σ in $\mathcal{O}(n\lg\sigma)$ time using $\mathcal{O}(\text{scan}(n\lceil\lg\sigma\rceil))$ I/Os, and $n\lceil\lg\sigma\rceil + \sigma\lceil\lg n\rceil$ bits of main memory including input and output, respectively.*

Adaptation to the Wavelet Matrix. Our semi-external memory wavelet tree construction algorithms can easily be extended to compute the wavelet matrix instead. To this end, we only have to compute the borders in bit reversal permutation order and thus change the order of the intervals within the bit vectors of each level, due to the similarity of wavelet trees and matrices (see Section 2.3). Also, this change does not affect the running time or the memory requirements; it only affects the content of the border array and subsequently the resulting bit vectors.

3.4.2 Sequential Construction in External Memory

While the semi-external memory algorithms described above reduce the required memory, the inputs we can process are still limited by the size of the main memory. In this section, we describe *fully* external memory (see Section 1.3.4 for a definition of the model) construction algorithms that dispose of these limitations.

If we replace the sorting in se.ps with any external memory sorting algorithm we obtain an external memory version of se.ps. However, sorting in external memory is expensive (in practice), and therefore not the best solution for external memory wavelet tree construction. Now, we present dedicated external memory wavelet tree and wavelet matrix construction algorithms. Unlike before, for the sequential algorithm we first explain how to build the wavelet matrix, and then show how to adapt the algorithm to produce the wavelet tree.

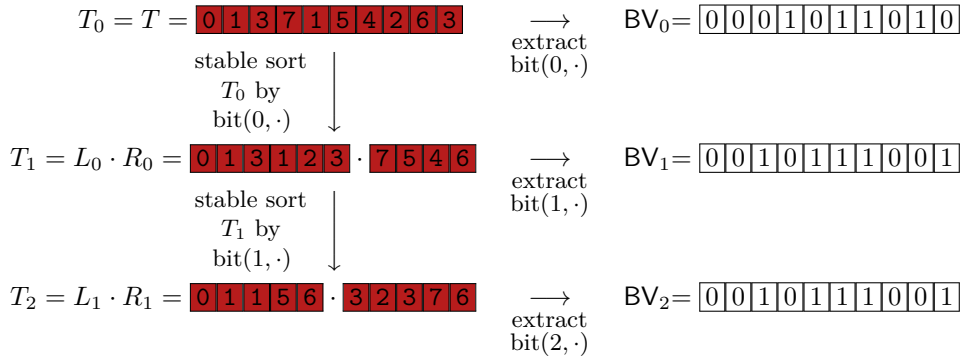


Figure 3.13. Construction of the wavelet matrix for our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$ by partitioning the text, which is highlighted in dark red (●) on the left-hand side, and the extraction of bits on the right-hand side.

Each level ℓ of the wavelet matrix can be interpreted as a reordered version T_ℓ of the original input text T, where the first level represents $T_0 = T$, and each text T_ℓ with $\ell > 0$ can be obtained by stably sorting the text $T_{\ell-1}$ of the previous level by the $(\ell-1)$-th bit. This property of the wavelet matrix has been originally described as *all zeros of the level go left, and all the ones go right* [Cla+15]. If we know T_ℓ, then we can easily build BV_ℓ by taking the ℓ-th bit of each symbol of T_ℓ in left-to-right order. Thus, we can construct the entire wavelet matrix by simply repeatedly sorting the text and extracting the bit vector of one level after each sort. Conveniently, the sorting key in each iteration is only a single bit. Therefore, we only have to create a binary partition of the text, where L_ℓ contains all the zeros of T_ℓ, and R_ℓ contains all the ones (retaining their order). Clearly, we have $T_{\ell+1} = L_\ell \cdot R_\ell$. In the external memory setting we can realize the partitioning by performing a single scan over T_ℓ and appending all characters α with $\mathrm{bit}(\ell, \alpha) = 0$ to L_ℓ and all other characters to R_ℓ. Also, we can simultaneously write the bit vector BV_ℓ by appending $\mathrm{bit}(\ell, \alpha)$ to BV_ℓ. Note that after the scan no additional copying is needed to get $T_{\ell+1}$ from L_ℓ and R_ℓ, as we can simply scan directly over L_ℓ and R_ℓ in the next iteration, see Figure 3.13. The number $\mathsf{Z}[\ell]$ of zeros in each level is $|L_\ell|$.

Analysis. The input text and resulting wavelet structure are of size $n\lceil \lg \sigma \rceil$ bits each, which can be stored using $\mathrm{scan}(n\lceil \lg \sigma \rceil)$ blocks in external memory. The input text is read exactly once (during the initial scan) and the resulting wavelet matrix is written exactly once (one level per scan), causing $2{\cdot}\mathrm{scan}(n\lceil \lg \sigma \rceil)$ I/Os per level. The combined size of L_ℓ and R_ℓ is $n\lceil \lg \sigma \rceil$ bits. Since we have to split these bits into two separate strings—a *string pair*—we might need one additional block in external memory, such that at most $\mathrm{scan}(n\lceil \lg \sigma \rceil)+1$ blocks of external memory are needed. The total number

of string pairs is $\lceil \lg \sigma \rceil - 1$ (one per scan except for the last scan), each of which is written and read exactly once, resulting in $(\lceil \lg \sigma \rceil - 1)(2 \cdot \text{scan}(n\lceil \lg \sigma \rceil) + 2)$ I/Os for all pairs. Therefore, the total number of I/Os used by our algorithm is bounded by $2 \lg \sigma \cdot \text{scan}(n\lceil \lg \sigma \rceil)$.

As we will see later, in terms of I/O complexity there is no difference between the wavelet tree and wavelet matrix construction algorithm. Each of our data structures (input, output, and all string pairs) are accessed exclusively in sequential order. Also, we only need to store two string pairs: one for the previous scan and one for the current scan. If we keep these four strings as well as input and output on separate disks, we have no concurrent and thus no random I/Os.

Now, we determine the time complexity and main memory bounds of our algorithm. Clearly, each of the $\lceil \lg \sigma \rceil$ scans takes $\mathcal{O}(n)$ time. Thus the overall time for the wavelet matrix construction is $\mathcal{O}(n \lg \sigma)$. In terms of space, the wavelet matrix construction is fully external and only needs $\mathcal{O}(1)$ bits of main memory, since all data structures are kept in external memory.

Lemma 3.9. *The fully external algorithm ext.ps computes the wavelet matrix of a text of length n over an alphabet of size σ in $\mathcal{O}(n \lg \sigma)$ time using a total of $2\lceil \lg \sigma \rceil \cdot \text{scan}(n\lceil \lg \sigma \rceil)$ I/Os and $\mathcal{O}(1)$ bits of main memory including input and output.*

Adaptation to the Wavelet Tree. Our external wavelet matrix construction algorithm can easily be adapted to construct the wavelet tree instead. As described in Section 2.3, the bit vector belonging to any interval of the wavelet tree always occurs in the wavelet matrix, too. Only the order of these intervals is different. Our L_ℓ and R_ℓ buffers therefore already contain all the correct intervals, but in wrong order. It is easy to see that L_ℓ contains exactly all of the left children, whereas R_ℓ contains the right children. Clearly, instead of defining $T_{\ell+1} = \mathsf{L}_\ell \cdot \mathsf{R}_\ell$ at the end of each scan, we can define $T_{\ell+1}$ by interleaving L_ℓ and R_ℓ such that left children and right children alternate. This way we will continue with the correct WT order in the next scan. To this end, we only need to know the size of each interval, allowing us to always read the appropriate number of characters from L_ℓ or R_ℓ. Hence, we simply determine the last level's histogram during the initial scan. After the scan we can compute all histograms in the bottom-up fashion, see Section 3.1. We simply keep the histograms of all levels in main memory.

For the wavelet tree we need $\lceil \sigma \lg n \rceil$ additional bits to store the histograms, as we explained in Section 3.1. The wavelet tree construction needs additional $\mathcal{O}(\sigma)$ time to compute the histograms of all levels, resulting in the following lemma.

Lemma 3.10. *The external algorithm ext.ps computes the wavelet matrix of a text of length n over an alphabet of size σ in $\mathcal{O}(n \lg \sigma + \sigma)$ time using a total of $2\lceil \lg \sigma \rceil \cdot \text{scan}(n\lceil \lg \sigma \rceil)$ I/Os and $2\lceil \sigma \lg n \rceil$ bits of main memory including input and output.*

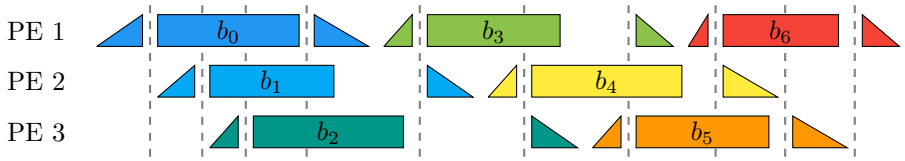


Figure 3.14. Domain decomposition for a text $T = b_0|b_1|b_2|b_3|b_4|b_5|b_6$ split into seven segments. Here, we use ◿ to denote that segment b_i is loaded from external memory, ▭ represents the computation of the partial wavelet tree for segment b_i, and ◺ means writing the partial wavelet tree of segment b_i to external memory. Only one of the three PEs is allowed to read/write at a time, as indicated by the dashed synchronization barriers.

3.4.3 Parallel Construction in External Memory

For a more generic approach, we present a meta-algorithm based on the internal memory domain decomposition, see, e. g., Section 3.3.3 and [FS+17; Lab+17]. Let p be the number of available processing elements, then in the internal memory setting we split the text into $m \cdot p$ segments, and compute the wavelet tree of each segment on a different processing element, using a sequential construction algorithm of our choice. We use the factor m to determine the length of the segments that is relevant for the required main memory. The details are described below. After that, the so called *partial trees* can be merged into one *global tree*, see Figure 3.9 for an example.

In the external memory setting the length of the segments depends on the amount M of main memory. Assume that the sequential construction algorithm needs $s(n, \sigma)$ bits of memory for a text of length n over the alphabet $[0, \sigma)$. Then, the length k of each segment must satisfy $s(k, \sigma) \leq M/p$. This way all processing elements can work simultaneously.

Each processing element runs a simple loop: load the next text segment from external memory into internal memory, compute the wavelet tree of the segment, and write it back to external memory. Only one processing element is allowed to read/write at a time (see Figure 3.14). In terms of external memory layout, we store the partial trees in text order, i. e., the partial tree of the second segment is stored right before the partial tree of the first segment, and so on. Here, each partial tree as the concatenation of its levels (see LT in Figure 3.15).

When merging the partial trees into the global tree, we simply perform a single scan over the partial trees and concatenate the corresponding intervals. Since the length of each interval must be known in order to copy the right amount of bits, we need the histograms of all text parts during the merge phase. However, many of the fastest sequential wavelet tree construction algorithms either build the histograms or can easily be modified to do so, e. g., all wavelet tree construction algorithms that we presented in Section 3.2. We are not using parallelism during the merge phase, since we are only copying bit vectors. In practice, in this step, we are limited by the speed of the external memory, even when using only a single processing element.

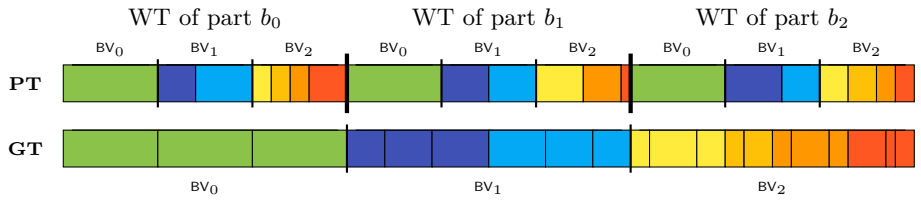


Figure 3.15. External memory layout of partial (PT) and global (GT) WTs for $T = b_0 b_1 b_2$. Best viewed in color, as colors indicate parts of partial trees that are zipped together.

Analysis. First, let us analyze the I/O complexity of our meta-algorithm. The input text, the concatenation of all partial trees as well as the global tree are of size $n\lceil \lg \sigma \rceil$ bits each, which can be stored using $\text{scan}(n\lceil \lg \sigma \rceil)$ blocks in external memory. We read the input text and write the partial trees once, taking $2\,\text{scan}(n\lceil \lg \sigma \rceil)$ sequential I/Os. Reading all partial trees sequentially during the merge phase causes another $\text{scan}(n\lceil \lg \sigma \rceil)$ I/Os. When writing the global tree we jump to a different external memory address for each interval of a partial tree. Hence, we need up to $\sigma\lceil n/k \rceil$ random I/Os in addition to the $\text{scan}(n\lceil \lg \sigma \rceil)$ I/Os that are generally needed to write the global tree. Thus, the total number of I/Os is bound by $4\,\text{scan}(n\lceil \lg \sigma \rceil) + \sigma\lceil n/k \rceil$. In practice we use the entire internal memory as a write buffer while merging the partial trees. This way we maximize the length of sequential writes and keep random I/Os at a minimum.

Now we determine the time complexity of our algorithm as well as the internal memory bounds. Let $t(n, \sigma)$ and $s(n, \sigma)$ be the time and the bits of memory used by the sequential construction algorithm that we deploy as a subroutine. We know that at any given point in time there is either exactly one processor performing I/Os, or all processing elements are computing partial trees. The total I/O time (including the merge phase) is bound by $\mathcal{O}(n + \sigma\lceil n/k \rceil)$. The time during which all processing elements are computing partial trees is bound by $\lceil n/pk \rceil \cdot t(k, \sigma)$. In terms of main memory we use $p \cdot s(k, \sigma)$ bits for up to p simultaneous executions of our internal memory construction algorithm over text segments of size k. Additionally, $\mathcal{O}(\lceil n/k \rceil \sigma \lg n)$ bits are needed to store all histograms. Alternatively, we could write the histograms to disk and only load the one that we need, reducing the required space but increase the number of I/Os.

Lemma 3.11. *Let $t(n, \sigma)$ and $s(n, \sigma)$ be the time and space used by an internal memory wavelet tree construction algorithm, p be the number of processing elements available, M the size of the main memory, and $m \in \mathbb{N}^+$ such that $n/(mp) < M/p$. Then, the external memory algorithm ext.dd computes the WT of a text of length n over an alphabet of size σ using $4\,\text{scan}(n\lceil \lg \sigma \rceil) + \sigma\lceil n/k \rceil$ I/Os. It takes $\mathcal{O}(n + \sigma\lceil n/k \rceil) + \lceil n/pk \rceil \cdot t(k, \sigma)$ time and $\mathcal{O}(\lceil n/k \rceil \sigma \lg n) + p \cdot s(k, \sigma)$ bits of internal memory including input and output.*

Adaptation to the Wavelet Matrix. Adapting this ext.dd to compute the wavelet matrix is simple. The only change necessary is to use a wavelet matrix construction algorithm as a subroutine. If we do so, the time, memory, and I/O bounds described in Lemma 3.11 hold for the corresponding wavelet matrix construction algorithm given that we use an algorithm with the same time and memory bounds.

3.4.4 Experimental Evaluation

As with all algorithms before, we implemented all our semi-external and external wavelet tree and wavelet matrix construction algorithms. Our implementations are available at `www.kurpicz.org/wavelet`. Unlike before, we did not conduct the experiments on nodes of the cluster, but on our external memory system that we described in Section 1.4.1. Here, we can choose between two configurations that either consist of hard disk drives (Ext.hdd) or solid state drives (Ext.ssd).

Our external memory algorithms use the STXXL [Dem+08b] development snapshot (26-09-2017). We compiled all source code using `GCC` 7.4 with flags `-O3` and `-march=native`, and express parallelism using OpenMP 4.5.

Evaluation of Semi-External Memory Algorithms

We compare the following semi-external memory wavelet tree construction algorithms: (i) se.pc, (ii) se.pc, (iii) se.par.pc, (iv) se.ps, and (v) se.ps.ip and their wavelet-matrix-constructing counterparts described in Section 3.4.1, (vi) seq.sdsl, the semi-external memory algorithm contained in the SDSL, (vii) seq.pc, the fastest sequential *main memory* wavelet tree construction algorithm (see Section 3.2.4), and (viii) par.dd.pc, the fastest *shared memory* wavelet tree construction algorithm. The last two algorithms are used to get a baseline and expected to be faster. We show the results of our experiments in Figures 3.16 and 3.17. However, we focus on the wavelet trees construction, as the conclusions also hold for the wavelet matrix construction. Also, the running times of Ext.hdd and Ext.ssd are nearly the same we discuss them combined.

Running Time. The first thing to notice is that the throughput of some algorithms drops when the input size exceeds 8 GiB. This is due to the memory requirements of the algorithms, because the required memory is less than 16 GiB, i. e., the size of the RAM of our system and the operating system has to *swap* some data to disk because it also requires RAM (less then 512 MiB).

As expected, our shared memory algorithm par.dd.pc is the fastest (before it has to swap). Then, se.par.pc is faster on `CommonCrawl` and `Wiki`, i. e., inputs with larger alphabets. Otherwise, par.dd.pc remains the fastest (being more than 3 times faster, as long as no swapping occurs). Another expected result is that the throughput of our parallel semi-external algorithm se.par.pc on inputs with small alphabet is not high. This shows on `DNA` and to some extend on `Prot`. We described the reasons for this in Section 3.3.1.

Regarding our sequential algorithms, our main memory algorithm seq.pc is the fastest on all instances. However, se.ps achieves up to 79 % of seq.pc's throughput

(on Wiki). This is because—even though wavelet tree construction is simple—the algorithms are all compute bound. The in-place version se.ps.ip is our slowest algorithm on all inputs, which is due to the complex in-place sorting. On inputs with small alphabets (DNA and Prot), se.pc is of similar speed as se.ps.ip; on CommonCrawl and Wiki it is of similar speed as se.ps. The semi-external memory algorithm provided by the SDSL is the slowest one we tested (on DNA and of similar speed as se.pc on all other inputs).

Memory Peak. The memory peaks of all tested algorithms but seq.sdsl are (nearly) constant when normalized by the input size. The slightly decreasing memory peak is due to constant sized buffers whose relative size (relative to the input size) becomes smaller with increasing input sizes.

In general, se.pc has the smallest memory peak. Its parallel variant se.par.pc requires only slightly more memory, as each processing element has the constant size buffers, i. e., they exist six times in our experiment. The memory peak of se.ps.ip only differs from the former two with respect to the buffer sizes—when the input's alphabet size is large, i. e., on CommonCrawl and Wiki. However, when the resulting wavelet tree is smaller, then it requires up to twice as much memory (DNA) as se.pc. Next, se.ps is our fastest semi-external wavelet tree construction algorithm. Unfortunately, it also requires the most memory of all our new algorithms; even more than seq.pc on all inputs. Only seq.sdsl requires more memory and is even slower.

Evaluation of External Memory Algorithms

Our external (and parallel) wavelet tree and wavelet matrix algorithms are the only external memory construction algorithms for wavelet trees and wavelet matrices. Hence, we cannot compare ext.ps and ext.dd or the corresponding wavelet matrix construction algorithms with other algorithms and we only report construction times and I/Os. We present the results of two experiments, first a strong scaling experiment and second a weak scaling experiment. During the former we increase the size of the text while keeping the number of used processing elements the same, i. e., one processing element for our sequential algorithms ext.ps and ext.ps.wm and six processing elements for our parallel algorithms ext.dd and ext.dd.wm. The latter experiment is only conducted for our parallel algorithms, as we increase the number of processing elements together with the input sizes.

We now look at the maximum throughput that we can achieve with our parallel algorithms using Ext.ssd and Ext.hdd and denote those by *ssd-max* and *hdd-max*. This throughput is what we achieve reading the text and the wavelet tree (or the wavelet matrix) once and writing the wavelet tree (or wavelet matrix) twice, which are exactly the external memory operations conducted by ext.dd (or ext.dd.wm)—without any computation. Therefore, the throughputs ssd-max and hdd-max are strict upper bounds for the throughput of our algorithms, because the algorithms also have to compute the wavelet tree (or wavelet matrix) in addition to reading and writing data from and to external memory.

In addition, we have two different versions of ext.dd and ext.dd.wm that we now briefly describe. The difference between those two versions is only practical and has no effects on the theoretical analysis of the algorithms.

1. In the *traditional* variant only one processing element is allowed to read or write at the same time. Thus, processing elements may have to wait for other processing elements to finish their I/Os. This is the default version, and we do not give this version a special name.

2. The concurrent read and write version, however, allows for concurrent read and write access by different processing elements. Hence, no processing element has to wait for its I/Os. In this case the operating system has to schedule the read and wrote access to external memory. We denote this version by *(concr. R/W)*.

In the following, similarly to previous evaluations, we only consider the results of the external memory wavelet tree construction algorithms, as they are nearly identical to the results of the external memory wavelet matrix construction algorithms.

Strong Scaling. We use the strong scaling experiments to show that the throughput of our algorithms does not depend on the input size. In Figures 3.18 and 3.19 we show the throughput and I/Os of our external memory algorithms computing the wavelet tree and wavelet matrix, respectively. Our parallel algorithm ext.dd uses all six processing elements.

The sequential algorithm ext.ps is the slowest one. Here, the difference between Ext.ssd and Ext.hdd is minimal, because the algorithms are compute bound and not limited by the bandwidth with which we can read and write data to external memory. For our parallel algorithms, however, there is a difference with the algorithms being up to 30 % faster on Ext.ssd than on Ext.hdd.

In addition, concurrent read and write increases the throughput on all instances when using SSDs, whereas it actually reduces the throughput on HDDs. However, concurrent reading and writing is only beneficial for larger inputs: at least 32 GiB of `Prot`, 64 GiB of `Wiki`, 128 GiB of `DNA`. On `CommonCrawl`, it is beneficial initially (for 32 GiB and 64 GiB) but slower when processing 128 GiB. Here, the wavelet tree construction and wavelet matrix construction actually differ slightly.

Regarding the I/O operations: our parallel algorithm ext.dd requires less reads and writes on all inputs but `DNA` than ext.ps. On the latter, due to the small alphabet size we can sort very efficiently, such that ext.ps requires 11.13 % less write and 33.39 % less read operations. However, even for `Prot` with $\sigma = 27$, ext.dd requires less reads and writes, even though only 1.79 times as many read and 1.49 times as many write operations. On `CommonCrawl` and `Wiki` it requires 2.25 times as much writes and 2.27 as many read operations.

Therefore, ext.dd is the overall better external memory wavelet tree construction algorithm (compared to ext.ps). Additionally, in our weak scaling experiments (see Figures 3.20 and 3.21) we report that ext.dd has a higher throughput than ext.ps even on one processing element.

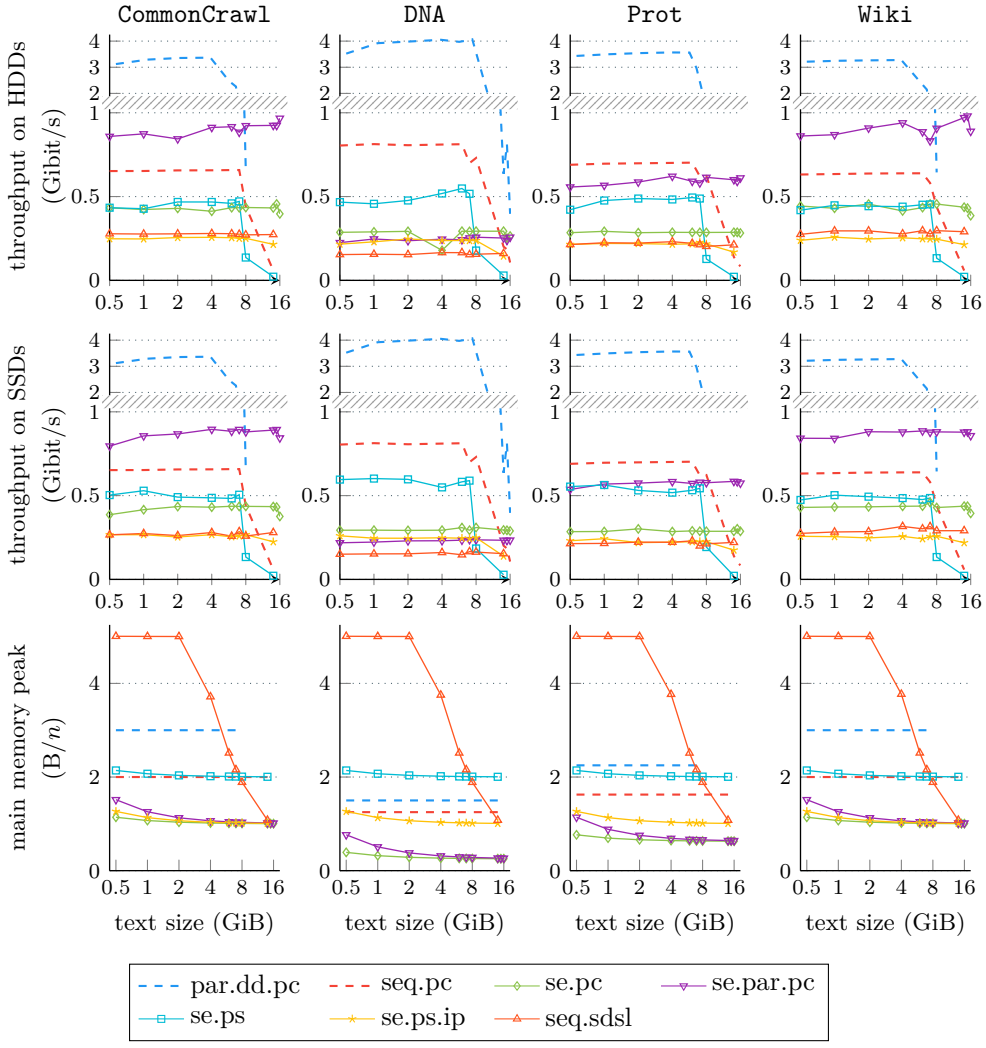


Figure 3.16. Throughput and main memory peaks of the semi-external wavelet *tree* construction algorithms when using the HDDs (first row) and when using the SSDs (second row). The parallel algorithms are using all 6 PEs. In the last row, we give the main memory peak, which is independent of the used drive. Note that we also measured running times for 6, 7, 14, and 15 GiB, to show the algorithm's behavior close to maximum sized inputs that they can process on this hardware.

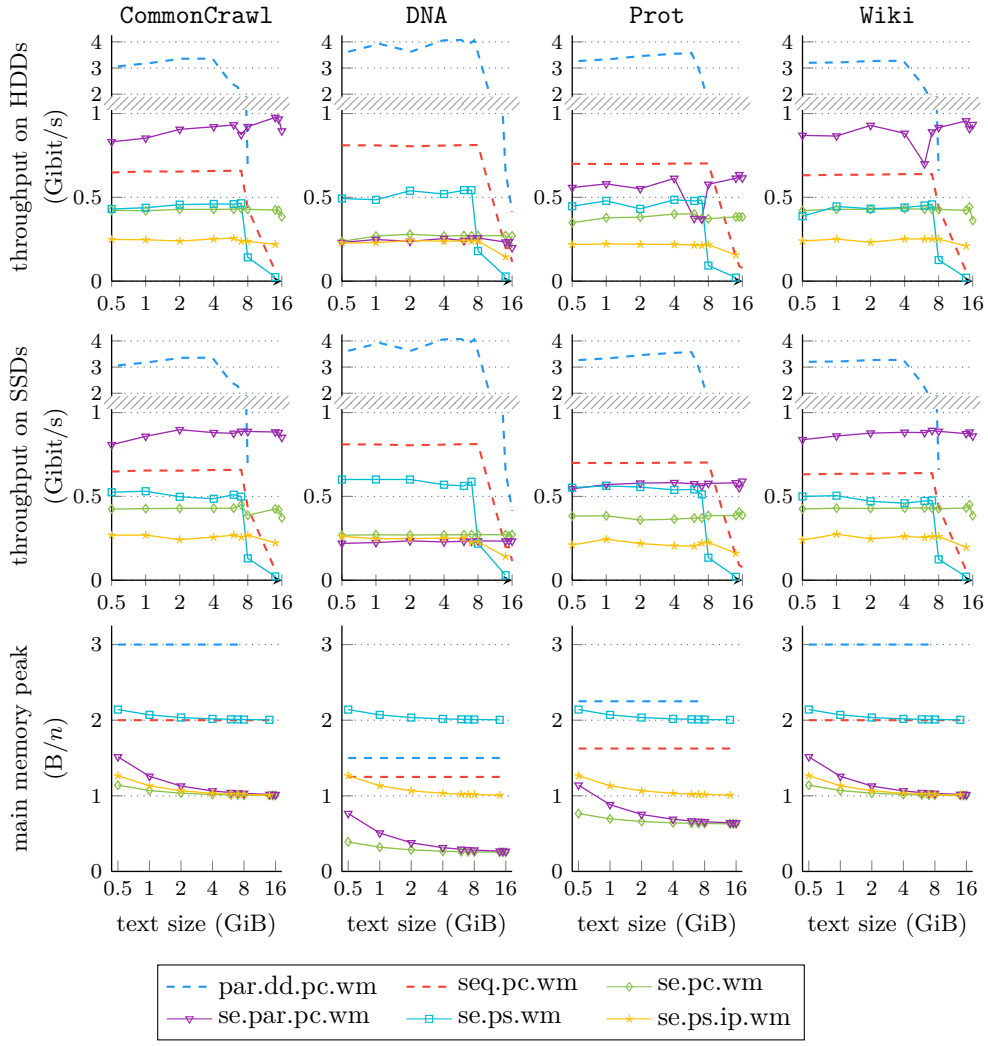


Figure 3.17. Throughput and main memory peaks of the semi-external wavelet *matrix* construction algorithms when using the HDDs (first row) and when using the SSDs (second row). The parallel algorithms are using all 6 PEs. In the last row, we give the main memory peak, which is independent of the used drive. Note that we also measured running times for 6, 7, 14, and 15 GiB, to show the algorithm's behavior close to maximum sized inputs that they can process on this hardware.

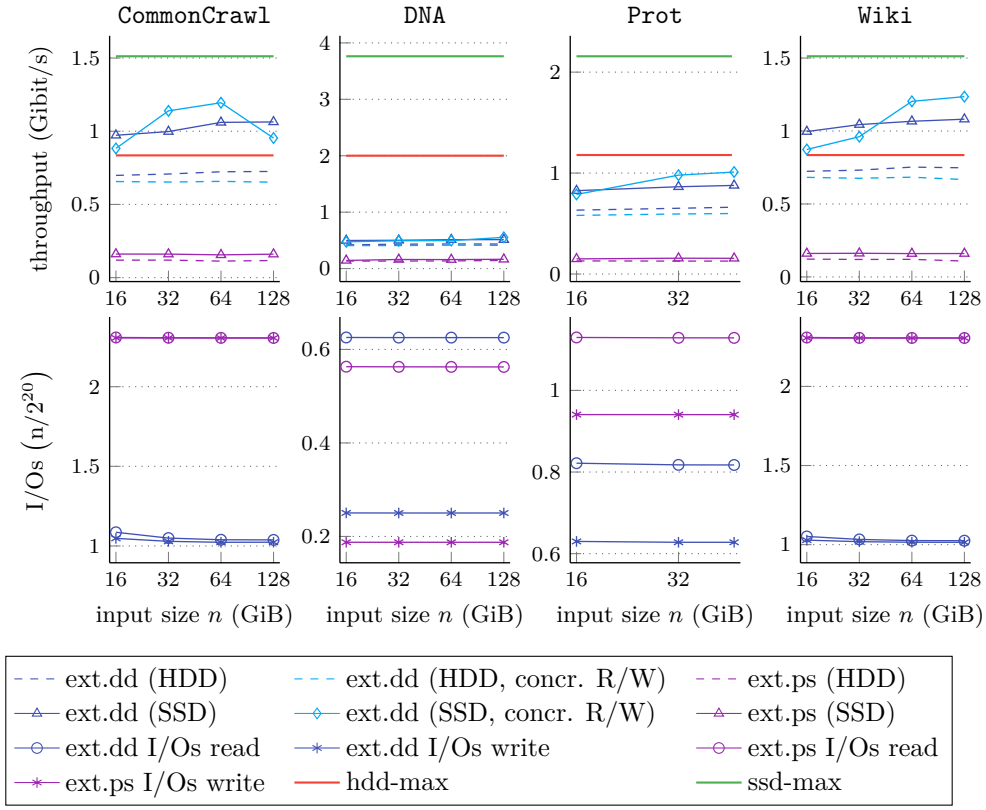


Figure 3.18. Throughput and I/Os of the external memory wavelet *tree* construction algorithms in our strong scaling experiment. Here, our parallel external memory algorithm ext.dd uses six processing elements.

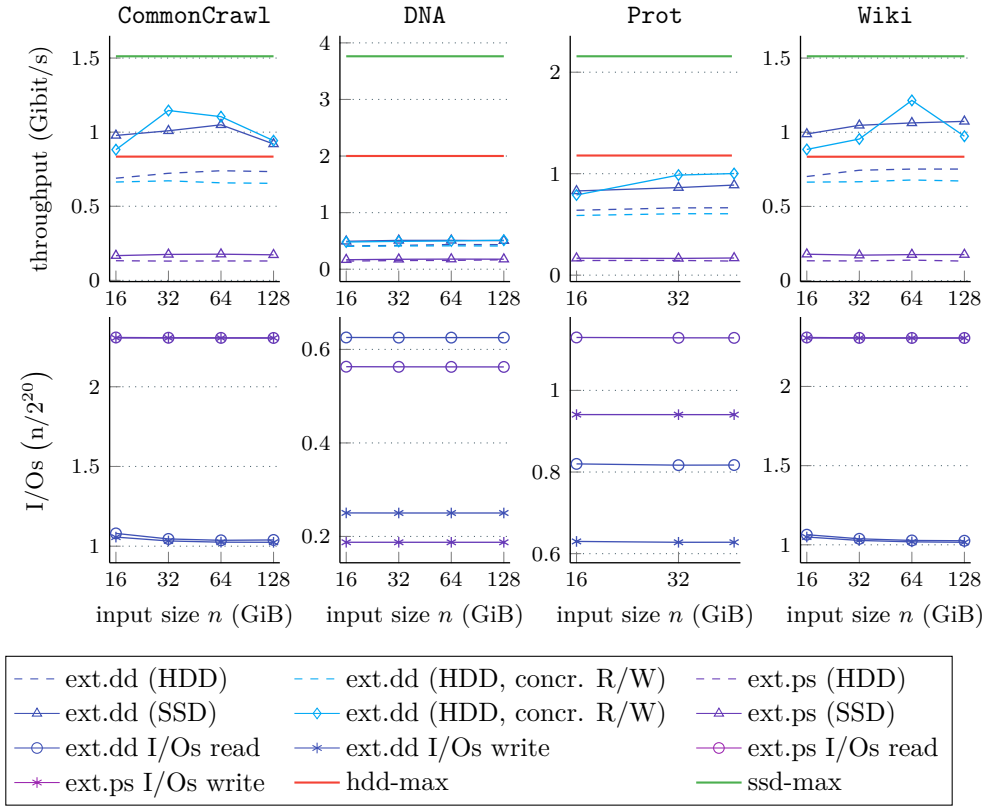


Figure 3.19. Throughput and I/Os of the external memory wavelet *matrix* construction algorithms in our strong scaling experiment. Here, our parallel external memory algorithm ext.dd uses six processing elements.

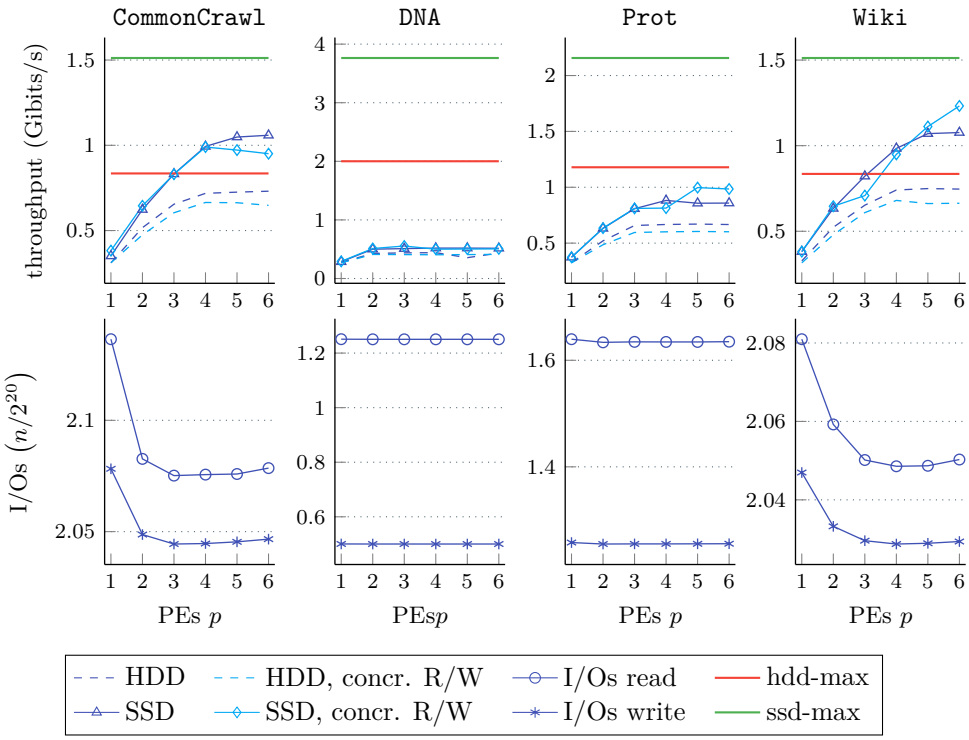


Figure 3.20. Throughput of ext.dd, our parallel wavelet *tree* construction algorithm, for inputs of size 20 GiB per PE in our weak scaling experiment.

Weak Scaling. We give the results of our weak scaling experiments in Figures 3.20 and 3.21, which are used to show that increasing the number of processing elements increases the throughput of our algorithms—even if we also increase the input size. Here, we only give the throughput (and I/Os) for the parallel external memory construction algorithm ext.dd, because as shown in the strong scaling experiments it is clearly superior, and ext.ps does not benefit from the access to additional processing elements. Again, we also show the maximum throughput that ext.dd and ext.dd.wm could achieve in both settings Ext.ssd and Ext.hdd. In this experiment, the results for both wavelet tree and wavelet matrix construction are nearly identical. Hence, we only describe results for wavelet tree construction.

We start with an analysis of the throughput. In both settings, the throughput increases nearly linearly with the number of used processing elements. However, on Ext.hdd, throughput does so only up to two (`DNA`), three (`Prot`), or four (`CommonCrawl` and `Wiki`) processing elements. Using Ext.ssd, we achieve the linear increase up to two (`DNA`), four (`CommonCrawl`), five (`Prot`), or six (`Wiki`) processing elements. Here, it should be noted that the best speedup on Ext.ssd is achieved when we allow concurrent

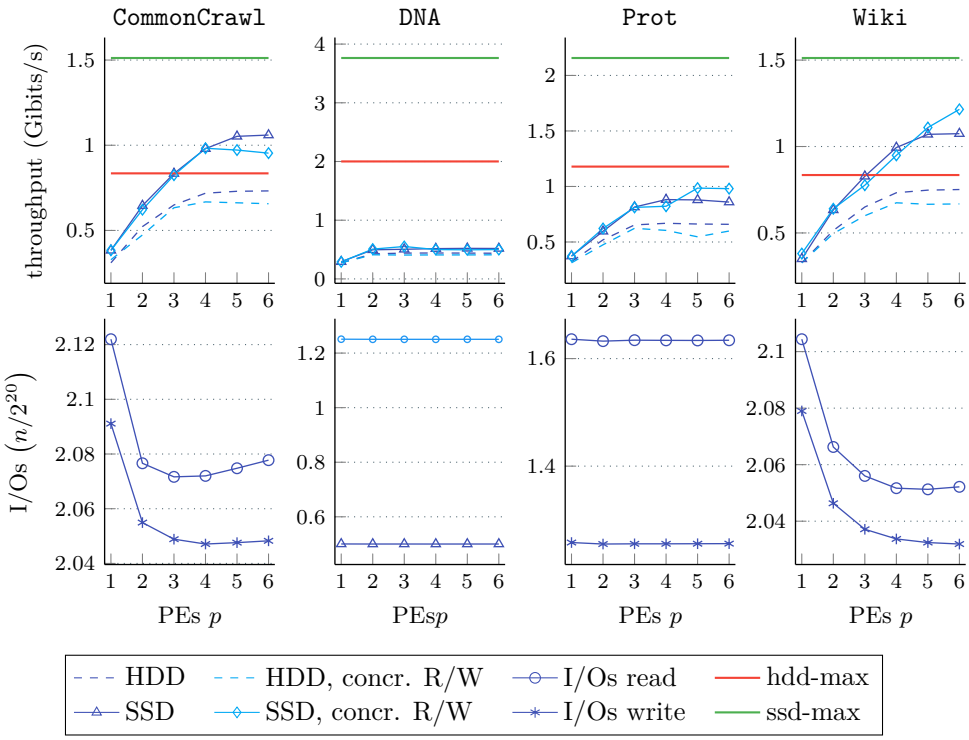


Figure 3.21. Throughput of ext.dd.wm, our parallel wavelet *matrix* construction algorithm, for inputs of size 20 GiB per PE in our weak scaling experiment.

reads and writes. On Ext.hdd this is never beneficial, as it decreases the throughput by around 7 %. Even though the speedup is nearly linear, the best speedup we obtain using six processing elements is on Ext.ssd using concurrent read and write operations on `Wiki`. This results in a speedup of 3.23. Still, these results are very good, because for large alphabets we achieve up to 81 % of the maximal possible throughput (compared to ssd-max) on SSDs and up to 89 % for HDDs (compared to hdd-max).

Now, we look at the number of I/O operations. We always consider I/Os that are normalized by the input size. First, we see that for `DNA` and `Prot` the number of I/Os is nearly constant for any number of processing elements. On `Prot`, we see a slight decrease when we use more than one processing element (by 0.2 %). On `CommonCrawl` and `Wiki` this decrease looks more severe, however it only drops by 1.65 %. This is due to the merging and writing partial wavelet trees to disk.

Overall, we see that ext.dd is the fastest external memory wavelet tree construction algorithm that also runs in parallel and scales well. Our other algorithm ext.ps shows that relying on sorting in external memory is too expensive when constructing a simple structure like the wavelet tree or wavelet matrix.

3.5 HUFFMAN-SHAPED WAVELET TREES

In this section, we take a look at compressed wavelet trees and wavelet matrices. To be more precise, we construct Huffman-shaped wavelet trees [Gro+03] and wavelet matrices [Cla+15], i. e., we construct the wavelet tree or matrix not for the original text but its Huffman encoded counterpart [Huf52]. Compared to the characters that we use as input for a *normal* wavelet tree or wavelet matrix, not all Huffman codes have the same length, hence not all bit vectors in a Huffman-shaped wavelet tree or wavelet matrix have the same length; but there may be more levels as the some Huffman codes may require more bits than the character it encodes. It is possible to balance the tree to obtain $\mathcal{O}(\lg \sigma)$ levels [NP13], which we do not consider in this section. Still, the total length of all bit vectors of a Huffman-shaped wavelet tree or wavelet matrix is at most as long as the total length of the corresponding normal wavelet tree or wavelet matrix. To be more precise, it is shorter by the same ratio the Huffman encoded text is shorter than the input text. The Huffman-shaped variants of the wavelet tree finds many applications in practice, e. g., the paper originally describing the (Huffman-shaped) wavelet tree as part of a compressed full-text indices [Gro+03], but also more recent FM-indices [Gog+19] use Huffman-shaped wavelet trees. In general, Huffman-shaped wavelet trees reduce the required space by increasing construction time.

In Section 3.5.1, we describe how to compute the Huffman codes such that they can be used in wavelet trees and wavelet matrices. Then, in Section 3.5.2, we explain how our algorithms described before—the sequential, shared memory parallel ones for wavelet trees and wavelet matrices—can be adapted to compute Huffman-shaped wavelet trees and matrices. Note that our semi-external and external memory wavelet tree and wavelet matrix construction algorithms can be extended in the same fashion. Last, in Section 3.5.3, we present practical results of the construction. Our implementations are to the author's best knowledge the *first* parallel Huffman-shaped wavelet tree (and wavelet matrix) construction algorithms and also the only sequential construction algorithms apart from the ones in SDSL [Gog+14a] or libcds [Cla+15].

3.5.1 Huffman Codes for Wavelet Trees and Wavelet Matrices

First, we briefly describe the construction of Huffman codes [Huf52] with a focus on codes that can be used for wavelet trees and wavelet matrices. Now, let us take a look how we can obtain Huffman codes for a text.

Computing Huffman Codes

Given a text T over an effective alphabet Σ and its histogram Hist. We describe the construction algorithm for Huffman codes based on trees and forests (a set of disjoint trees). Initially, there is a tree $t_{\{\alpha\}}$ for each $\alpha \in \Sigma$ with weight $w(t_\alpha) \coloneqq \mathsf{Hist}[\alpha]$. Now, we merge two of the trees with the smallest weight, e. g., t_A and t_b, to a tree $t_{A \cup B}$ with weight $w(t_{A \cup B}) = w(t_A) + w(t_B)$ by creating a new root with two children that are t_A and t_B. We repeat this process until there is only one tree left, which is a binary tree by definition. Remember that the leaves of this tree correspond to the

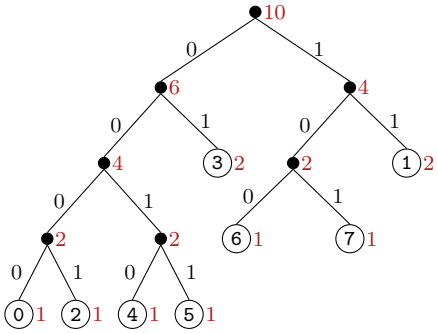


Figure 3.22. A Huffman tree for our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$. Each character is represented by a leaf. The light grey (●) node labels are the total number of occurrences of all characters represented in the subtree. The Huffman code of a character is the concatenation of all edge labels on the path from the root to the character's leaf.

symbols of the alphabet. If we implicitly label all edges going to a left child with a 0 and all other edges with a 1, then the Huffman code for each symbol is given by the concatenation of the labels on its path from the root to its correspond leaf. We give an example of a Huffman tree for our running example in Figure 3.22.

Before we look at the specific construction algorithms for the Huffman codes that are required, let us briefly recall the problems that occur when using Huffman codes for wavelet tree and wavelet matrix construction. As mentioned before, Huffman codes are prefix free variable-length codes. Hence, all codes can have different lengths. Regarding the structure of wavelet trees and wavelet matrices, this can lead to the problem of *disappearing* intervals, where disappearing intervals are intervals that would represent Huffman codes with bit prefix of length ℓ that do *not* occur, whereas Huffman codes with bit prefix $\text{bit\_prefix}(\ell - 1, bp)$ exist. For example, if there is a Huffman code $c = (0\,1)_2$, then there is no character represented by any interval with bit prefix c at level ℓ with $\ell \geq 3$, because Huffman codes are prefix free by definition.

In level-wise wavelet trees and wavelet matrices, disappearing intervals can be problematic when we answer queries, because disappearing intervals change the expected positions of intervals on all levels below and including the one they disappear in. To avoid this problem, all disappearing intervals should occur on the right-hand side of the bit vector they disappear in. Then, no other intervals are affected by their disappearance. This also simplifies queries, as we can easily determine if we have found a code word simply by looking at the length of the bit vector. If we require a bit that is not contained in the bit vector, because its position is greater than the length of the bit vector, then we have identified a code word. To this end, we have to compute the codes slightly different for the wavelet tree and the wavelet matrix, as the order of the intervals at each level differs for both.

Huffman Codes for Wavelet Trees

When computing Huffman codes that are useful for wavelet trees, the disappearing intervals must represent the largest symbols—codes in this case—as the intervals at each level of the wavelet tree are ordered ascending. Therefore, for Huffman-shaped wavelet trees, we use the *canonical* Huffman code, a variant of Huffman code that assigns codes of equal length consecutive and most importantly ascending values. The (canonical) Huffman codes for our running example $T = [\,\mathsf{0}, \mathsf{1}, \mathsf{3}, \mathsf{7}, \mathsf{1}, \mathsf{5}, \mathsf{4}, \mathsf{2}, \mathsf{6}, \mathsf{3}\,]$ are depicted in Figure 3.23.

To obtain the canonical Huffman, we first compute the (normal) Huffman codes for our text. Then, we order these codes by length. Finally, we start with the code word $chc = (\mathsf{0}^\ell)_2$, where ℓ is the length of our shortest code word. Now, chc is the first canonical code word. We increase chc by one and append $\ell' - \ell$ zeros to its right, where ℓ' is the length of the next code word. In practice, we can append zeros by shifting c by that many bits, which makes the transformation very easy to compute. Then, chc is the next canonical code word and we repeat the process until no more code are left to be transformed to canonical code words. Now, we have canonical Huffman codes. However, we need to bitwise negate all code words obtained this way. Otherwise disappearing intervals would occur on the left-hand side of the bit vectors, as short code words correspond to small numbers (if interpreted as integer).

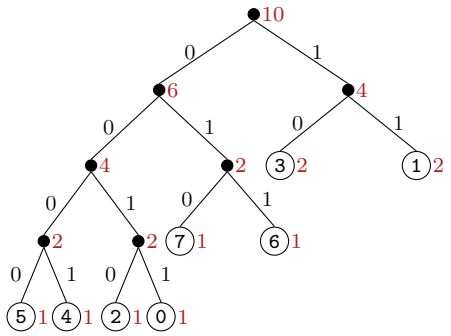
Huffman Codes for Wavelet Matrices

As described in Section 2.3, wavelet matrices do not have the tree structure and the intervals get intermingled with respect to their bit prefixes. Therefore, canonical Huffman codes do not result in disappearing intervals on the right-hand side of the bit vectors. Instead, we use the Huffman-like codes proposed by Claude et al. [Cla+15] that are also optimal prefix free codes. We give an example of Huffman-like codes for our running example in Figure 3.24.

Computing the Huffman-like codes is similar to computing the canonical Huffman codes. First, we compute the lengths of all Huffman codes (by computing the Huffman codes). Then, we start with the set $C = \{(0)_2, (1)_2\}$ and for each code of length one, we use $hlc = \operatorname{argmax}_{c \in C} \operatorname{reverse}(c)$ as code word and remove c from C. Note that there are either only two codes in total, or at most one code with length one. In the next step, we append $(0)_2$ and $(1)_2$ to all elements in C, which doubles the number of elements remaining in C. We then repeat this process for all codes with length two. Afterwards, we again append $(0)_2$ and $(1)_2$ to all elements in C. We repeat this process until we have computed all code words. Now, the text is encoded by codes that have the same length as the Huffman codes. We choose the elements mimicking the bit reversal permutation, resulting in disappearing intervals only on the right-hand side of the bit vectors.

α	$hc(\alpha)$	$chc(\alpha)$
1	$(11)_2$	$(11)_2$
3	$(01)_2$	$(10)_2$
6	$(100)_2$	$(011)_2$
7	$(101)_2$	$(010)_2$
0	$(0000)_2$	$(0011)_2$
2	$(0001)_2$	$(0010)_2$
4	$(0010)_2$	$(0001)_2$
5	$(0011)_2$	$(0000)_2$

(a) Huffman codes (hc) given by the Huffman tree depicted in Figure 3.22 and the resulting bitwise negated canonical Huffman codes (chc).



(b) Modified Huffman tree corresponding to the bitwise negated canonical Huffman codes in (a).

0	1	3	7	1	5	4	2	6	3
0	1	1	0	1	0	0	0	0	1
0	7	5	4	2	6	1	3	1	3
0	1	0	0	0	1	1	0	1	0
0	5	4	2	7	6				
1	0	0	1	0	1				
5	4	0	2						
0	1	1	0						

(c) Huffman-shaped wavelet tree using the bitwise negated canonical Huffman codes given in (a).

Figure 3.23. Huffman codes corresponding to the Huffman tree in Figure 3.22 and the resulting bitwise negated canonical Huffman codes (a), the modified Huffman tree for the bitwise negated canonical Huffman codes (b), and the resulting Huffman-shaped wavelet *tree* (c). All for our running example $T = [0, 1, 3, 7, 1, 5, 4, 2, 6, 3]$. As with our previous examples, the dark red (■) parts show the characters represented at the corresponding position in the array and is not part of the wavelet tree.

3.5.2 Huffman-shaped Wavelet Tree Construction Algorithms

Using the canonical Huffman codes for wavelet trees and the Huffman-like codes for wavelet matrices, we can adapt our algorithms that we described in the previous sections to compute Huffman-shaped wavelet trees and wavelet matrices. Similar to previous sections, we focus on wavelet tree construction, but all algorithms are also extended to also compute the wavelet matrix.

Due to the structure of Huffman-shaped wavelet trees and wavelet matrices, we cannot use the *bottop-up* construction technique, which we described in Section 3.1, because we cannot compute the histogram of level ℓ based on the histogram of level $\ell + 1$, as upper levels in the histogram can contain more bits than the histogram of level ℓ. That is because not all code words have the same length, and some code words are represented by an interval at level ℓ but not by an interval at level $\ell + 1$.

Instead, we compute the wavelet trees and matrices top-down. We compute all histograms during the first scan of the text, which requires moderately additional space but decreases the running time significantly in practice. Also, we *reduce* the text, i. e., whenever we scan through the text at level ℓ, we remove characters whose code word has length ℓ, as those characters are not represented by any interval at level $\ell + 1$. Except for the computation of the histograms we can reuse all algorithms that we have described in this chapter. The computation of the histograms differs because now there are characters that do not occur on all levels. While we can compute the characters that disappear at each level, it is not practical, as this requires a check of each symbol not occurring in the previous level if it appears in the current one. These checks require more time than to simply recompute the histogram. Therefore, we only briefly describe the algorithms that we have implemented.

Sequential Construction Algorithms. For the sequential wavelet tree and wavelet matrix construction we do not have to change anything else (except for the histogram computation). Hence, we reuse the techniques of prefix *counting* (Section 3.2.1) and *sorting* (Section 3.2.1) for their construction. Apart from the top-down constructions, which in practice results in an additional scan of the text per level, the algorithms remain the same. We denote the resulting algorithms by *seq.pc.huff*, *seq.pc.ss.huff*, and *seq.ps.huff*, i. e., we append the suffix *.huff* to denote the algorithms for the Huffman-shaped wavelet tree construction algorithms. Their wavelet-matrix-constructing counterparts are denoted by the *.wm* suffix. We evaluate the sequential construction algorithms in Section 3.5.3.

Parallel Construction Algorithms. Now, we describe how we parallelize the Huffman-shaped wavelet tree and wavelet matrix construction. Based on the running time of our parallel *normal*-shaped wavelet tree construction that we have examined extensively in Section 3.3, we focus on the fastest approach to parallelize wavelet tree and wavelet matrix construction, *domain decomposition*, and apply it to Huffman-shaped wavelet trees and wavelet matrices. We refer to Section 3.3.3 for a detailed description of wavelet tree construction using domain decomposition.

α	$hlc(\alpha)$
1	$(01)_2$
3	$(11)_2$
6	$(001)_2$
7	$(101)_2$
0	$(0000)_2$
2	$(0001)_2$
4	$(1000)_2$
5	$(1001)_2$

(a) Huffman-like codes that have the same length as Huffman codes.

0	1	3	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	0	1
0	1	1	2	6	3	7	5	4	3
0	1	1	0	0	1	0	0	0	1
0	2	6	7	5	4				
0	0	1	1	0	0				
0	2	5	4						
0	1	0	1						

(b) Huffman-shaped wavelet matrix using the Huffman-like codes given in (a).

Figure 3.24. Huffman-like codes [Cla+15] that have the same length as Huffman codes but are constructed differently (a) and the corresponding Huffman-shaped wavelet *matrix* (b) for the text $T = [0, 1, 3, 7, 1, 5, 4, 2, 6, 3]$. As with our previous examples, the dark red (●) parts show the characters represented at the corresponding position in the array and is not part of the wavelet matrix.

Briefly, we compute partial wavelet trees (or wavelet matrices) for a slice of the text in parallel and merge those in parallel. As a result of this, we can use any *sequential* Huffman-shaped wavelet tree (or wavelet matrix) construction algorithms and parallelize it using this approach, however, we have to use *global* Huffman codes, i. e., Huffman codes for the whole text. Otherwise, merging is not possible, because the same character could be encoded differently on different processing elements. We only need a parallel merge for the partial Huffman-shaped wavelet trees (and wavelet matrices). The merge however is similar to the one for normal-shaped wavelet trees and wavelet matrices we describe in Section 3.3.3. The only difference is that we have to consider disappearing intervals.

Using domain decomposition, we can use all our sequential Huffman-shaped wavelet tree (and wavelet matrix) construction algorithms described above and use the in our parallel domain decomposition. This results in the following parallel construction algorithms, where we follow the naming scheme from the previous sections: *par.pc.huff*, *par.pc.ss.huff*, *par.ps.huff*, *par.dd.pc.huff*, *par.dd.pc.ss.huff*, and *par.dd.ps.huff*. Again, the corresponding Huffman-shaped wavelet matrix construction algorithms are denoted by the *.wm* suffix.

It should be noted that we do not compute the Huffman codes in parallel. To be precise, we compute the histograms in parallel, but the computation of the codes based on the number of occurrences is sequential. This is because computing the Huffman codes is fast compared to the computation of the histograms

3.5.3 Experimental Evaluation

We implemented all our Huffman-shaped wavelet tree and wavelet matrix construction algorithms that we described in the previous section. Our implementations are available at `www.kurpicz.org/wavelet`.

For our experiments we used the hardware described in Section 1.4.1 and conducted the experiments on LiDO.big nodes. We used (prefixes of) the texts described in Section 1.4.2 as inputs. Since we want to compare the results of the experiments with our experiments on the sequential (Section 3.2) and parallel (Section 3.3) wavelet tree and wavelet matrix construction we compiled our code with GCC 7.3.0 with flags `-O3` and `-march=native`. We justify the usage of an older compiler version in Section 3.3.

Evaluation of Sequential Algorithms

We compare our Huffman-shaped wavelet tree and wavelet matrix construction algorithm with the only other publicly available implementation that are aware of seq.sdsl.huff, which is part of the SDSL [Gog+14a]. To the author's best knowledge, there are no other implementations of Huffman-shaped wavelet tree and wavelet matrix construction algorithms publicly available.

Since we must compute the Huffman codes for the input, we first load the text into main memory, compute the effective alphabet, and then start the timing, before computing the Huffman codes for the input. Again, we stop the timing as soon as the Huffman-shaped wavelet tree or wavelet matrix has been computed. As in all previous experiments, measured running times are the median of five executions of the construction algorithm. The maximum time for these five executions and an additional computation that we use to compute the memory usage is two hours.

Similar to all previous experimental evaluations of wavelet tree and wavelet matrix construction algorithms, we focus on the results of the results of the Huffman-shaped wavelet tree construction algorithms, as the results for their wavelet-matrix-constructing counterparts are nearly identical. Since Huffman-shaped wavelet trees and wavelet matrices require different Huffman codes, this means that the construction of those requires the same amount of time in practice. Also, by design, the compression that is achieved by using either construction is the same, i. e., the resulting Huffman-shaped wavelet trees and wavelet matrices have the same size.

Construction Time. We first look at the construction times. In Figure 3.25, we give the throughput of our algorithms that has been normalized by the input size. (The throughput for the wavelet matrix construction algorithms are depicted in Figure 3.27.) In general, we obtain half the throughput we get when we construct the *normal*-shaped wavelet tree, compare Section 3.2.4. The reason for this is threefold: (1) we have to compute the corresponding Huffman codes, (2) we have to map each character to its Huffman code whenever we access it, as we cannot practically overwrite the input text with Huffman codes, and (3) the computation is slower as we cannot use our bottom-up approach for reasons we described in Section 3.5.2.

Now, we take a more detailed look at the throughput of the construction algorithms. First, we see that the naive algorithm seq.naive.huff did not finish the experiments for inputs larger than 1 GiB within two hours (which is the time limit for all our experiments in this part of the dissertation). This is because text access becomes more expensive if we have to encode each character whenever we access it, which is necessary as we do not store the encoded text. Storing the encoded text is not beneficial if we use more sophisticated algorithms. Due to its low throughput, we discard seq.naive.huff in the further discussion of Huffman-shaped wavelet tree construction algorithms.

Unlike before, there is not *one* clear fastest or slowest algorithm. The Huffman-shaped wavelet tree construction algorithm seq.sdsl.huff and seq.ps.huff are the two slowest algorithms with seq.ps.huff being slower on `CommonCrawl` and `Wiki`, and seq.sdsl.huff being slower on `DNA`. Both algorithms have nearly the same throughput on `Prot`. The difference in throughput is at most 25 Mibits/s.

The fastest two algorithms are seq.pc.huff and seq.pc.ss.huff. The former is the fastest on `DNA` and `Prot` and the latter on `CommonCrawl`. On `Wiki` both algorithms have a similar throughput with seq.pc.ss.huff becoming faster for larger (at least 4 GiB) inputs. Notably, seq.pc.huff is faster on inputs with smaller alphabet. This is due to the fact that seq.pc.ss.huff only encodes each character once to compute all levels. Encoding a character with its corresponding Huffman code is the bottleneck in this algorithm, which is why we only see this effect for large alphabets. Hence, for Huffman-shaped wavelet trees, the fastest algorithm depends on the input. This shows that the input is of greater importance here, as different Huffman codes result in different sizes of the levels, see Section 1.4.2 for the empirical entropy H_0 of the used inputs.

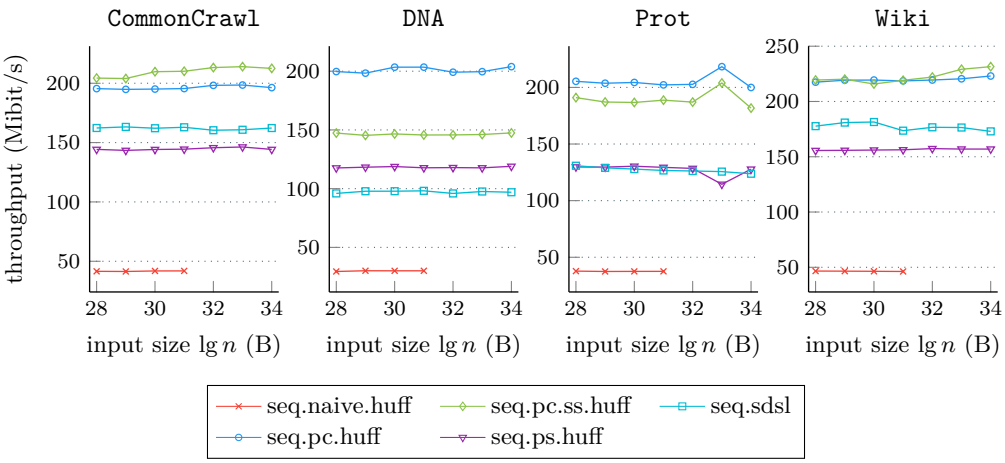


Figure 3.25. Throughput of the sequential Huffman-shaped wavelet *tree* construction algorithms.

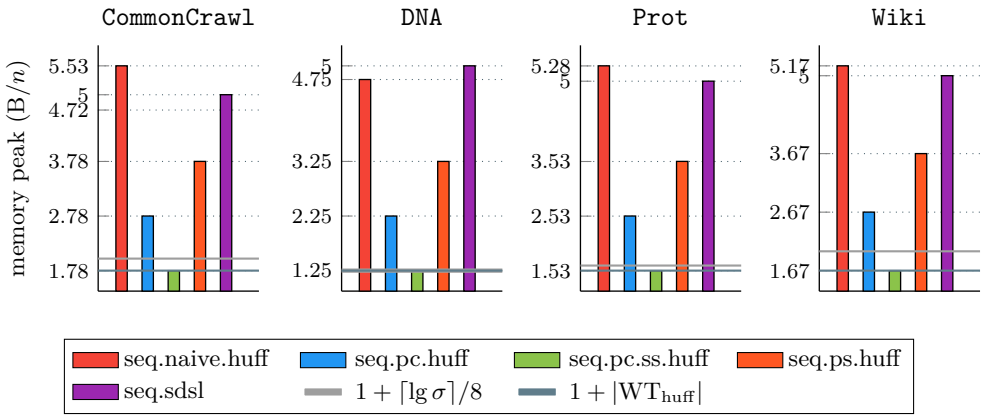


Figure 3.26. Memory peaks of wavelet *tree* construction algorithms for $n = 2^{31}$. Also depicted is the memory required the text and the *normal*-shaped wavelet tree ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character), the text and the Huffman-shaped wavelet tree ($1 + |\mathrm{WT}_{\mathrm{huff}}|$).

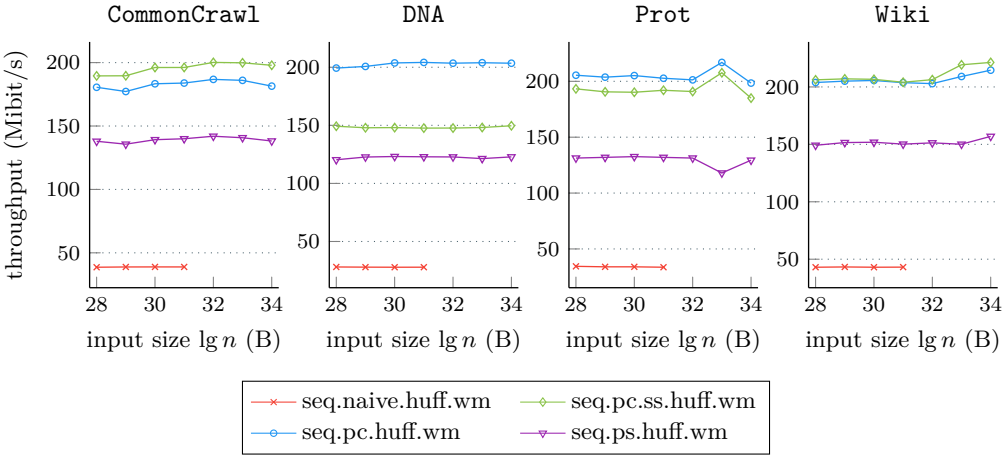


Figure 3.27. Throughput of the sequential Huffman-shaped wavelet *matrix* construction algorithms.

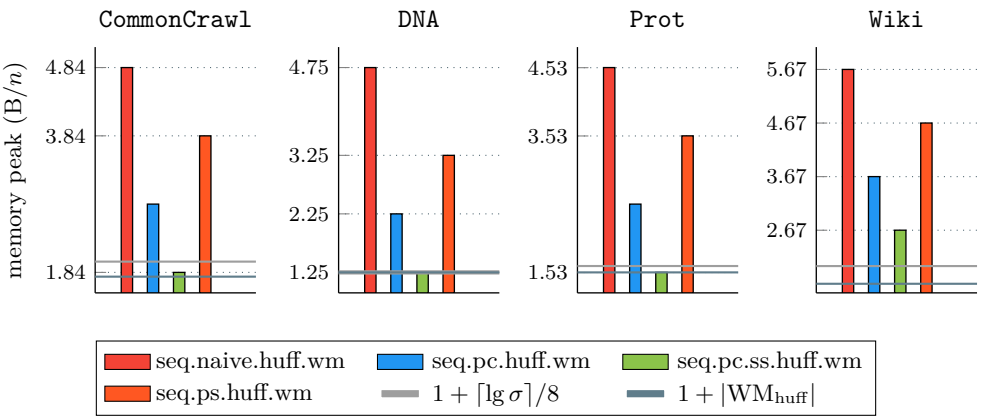


Figure 3.28. Snapshot of the memory peaks of sequential wavelet *matrix* construction algorithms for $n = 2^{31}$. Also depicted is the memory required the text and the *normal*-shaped wavelet matrix ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character), the text and the Huffman-shaped wavelet matrix ($1 + |\mathrm{WT}_{\mathrm{huff}}|$).

Memory Peak. In Figure 3.26, we report the memory peaks of the Huffman-shaped wavelet tree construction algorithms for inputs of size 2 GiB, which is the largest input size that all algorithms could process in the time limit of our experimental setting. (The corresponding results for the Huffman-shaped wavelet matrix construction are depicted in Figure 3.28.) In addition to the size of the Huffman-shaped wavelet tree, we also give the size of the *normal*-shaped wavelet tree.

We see that our naive construction algorithm seq.naive.huff requires the most memory on all inputs but `DNA`, i. e., at least 4.75 bytes per character of the input. Thus, the naive algorithm is not only the slowest but also the most memory inefficient one.

Next, seq.sdsl.huff requires 5 bytes per character of input on all inputs, which is the same as the normal-shaped wavelet tree construction algorithm contained in the SDSL, as we have shown in Figure 3.4. Also, on `DNA` it requires even more memory than seq.naive.huff, making it unpractical compared to our other algorithms.

When looking at the results for our fast algorithms, the biggest difference between our Huffman-shaped and *normal*-shaped wavelet tree construction algorithms is that seq.pc.huff requires more memory than seq.pc.ss.huff. This is because we overwrite the text in seq.pc.huff, which helps ignore characters that are already been fully contained in the Huffman-shaped wavelet tree, i. e., their code length is smaller than the current level. Since all our algorithms are not allowed to change the input text, we need to copy it. If we allow seq.pc.huff to overwrite the input, then it has the same memory peak as seq.pc.ss.huff.

Evaluation of Parallel Algorithms

Since there are no other parallel Huffman-shaped wavelet tree or wavelet matrix construction algorithms, we only present the results for our algorithms. We conducted a weak scaling experiment where we constructed the wavelet tree for 128 MiB, 256 MiB, and 512 MiB per processing element and $1, 2, 4, 8, 16, 32$, or 48 processing elements. Other than that, the setting is exactly the same as for the sequential Huffman-shaped wavelet tree construction algorithms. Again, we only interpret the results for the wavelet tree construction, as the results for wavelet matrices are very similar.

Construction Time. We give the throughput of the Huffman-shaped wavelet tree construction algorithms in our weak scaling experiment in Figure 3.29. (In Figure 3.30, we give the throughput for the wavelet matrix construction.)

Using the naive algorithm par.dd.naive.huff in the domain decomposition results, as expected, is the slowest parallel Huffman-shaped wavelet tree construction algorithms. While being slow, it scales reasonably well, as expected when using domain decomposition to parallelize the construction. Next, par.dd.ps.huff.wm is the second slowest algorithm on all inputs but `Wiki`. Because of its sequential performance this is without surprise.

Finally, par.dd.pc.ss.huff and par.dd.pc.ss.huff are the two fastest parallel Huffman-shaped wavelet tree construction algorithm. However, we need to look at all texts individually, as the algorithms behave differently (as in the sequential case) depending on the input. On `CommonCrawl`, both par.dd.pc.ss.huff is slightly faster (0.01 Gibits/s) on one processing element and is 1.07 Gibits/s faster on 48 processing elements.

When looking at our inputs with small alphabets, the situation is different. On `DNA`, par.dd.pc.huff is faster than par.dd.pc.ss.huff when using less than 16 processing elements. Using 16 processing elements, the algorithms are of similar speed. When we use more than 16 processing elements, par.dd.pc.ss.huff is faster than par.dd.pc.ss.huff. This is also the same on `Prot`, however, the difference in throughput is smaller. Finally, on `Wiki`, par.dd.pc.ss.huff is the fastest algorithm.

We give the throughput of all algorithms on one and on 48 processing elements in Table 3.2. There, we also report the speedup of all algorithms. It is interesting that the speedup of our Huffman-shaped wavelet tree construction algorithms on 48 processing elements is higher than the one of our normal wavelet tree construction algorithms. This is because the computation is not compute bound but limited by the bandwidth of the main memory. This limitation is not as strong when we have to encode the characters. Summarizing, par.dd.pc.ss.huff has the highest speedups on all inputs expect for `Wiki` where par.dd.pc.ss.huff has the highest speedup.

The COST (Section 1.3.3) of the parallelization is two, as our parallel algorithms are as fast as the fastest sequential Huffman-shaped wavelet tree construction algorithm when using only one processing element but faster if two or more processing elements are used. Therefore, our parallel algorithms do not only scale well because they are slow when executed as sequential algorithm.

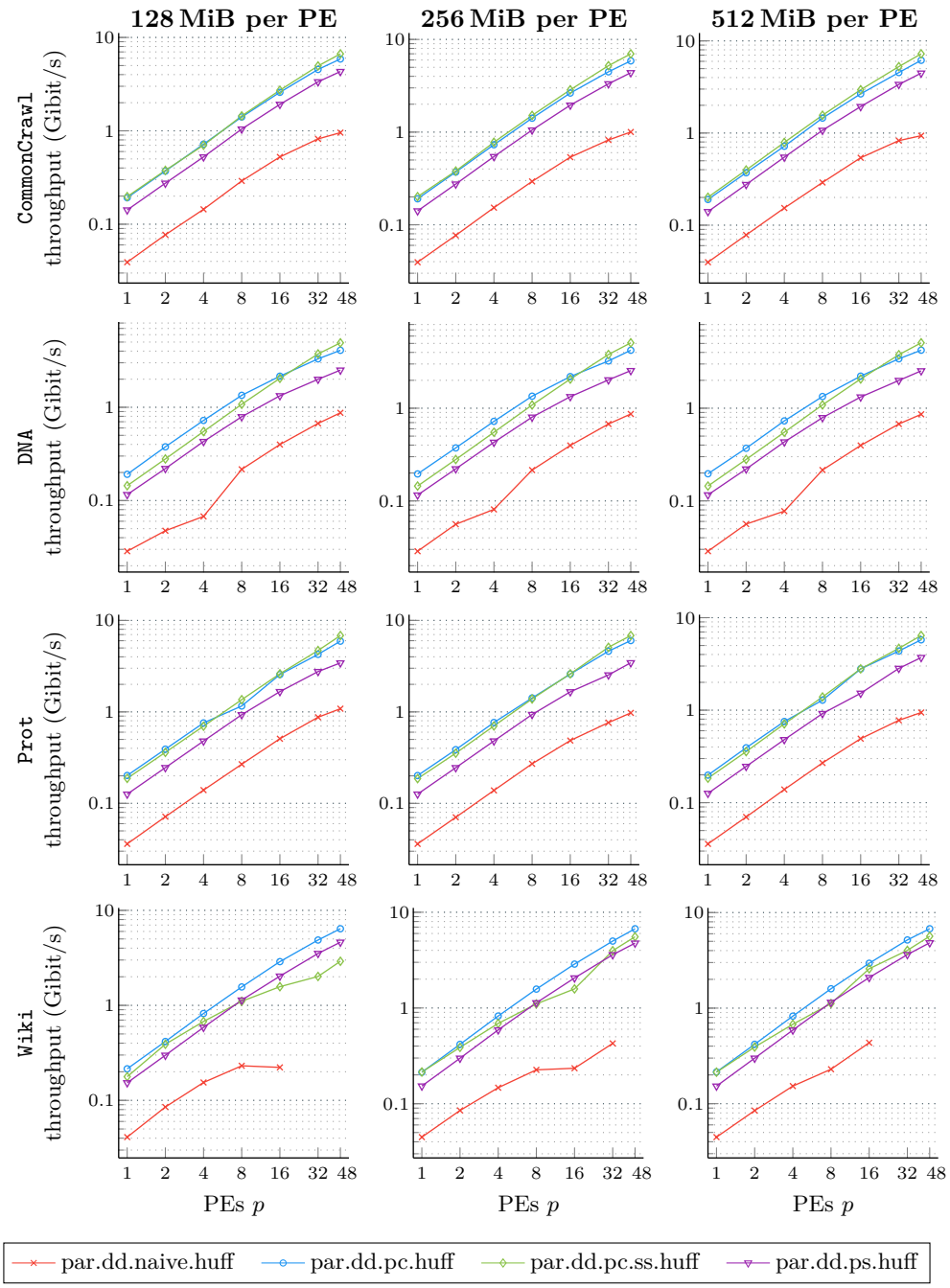


Figure 3.29. Weak scaling Huffman-shaped wavelet *tree* construction experiments.

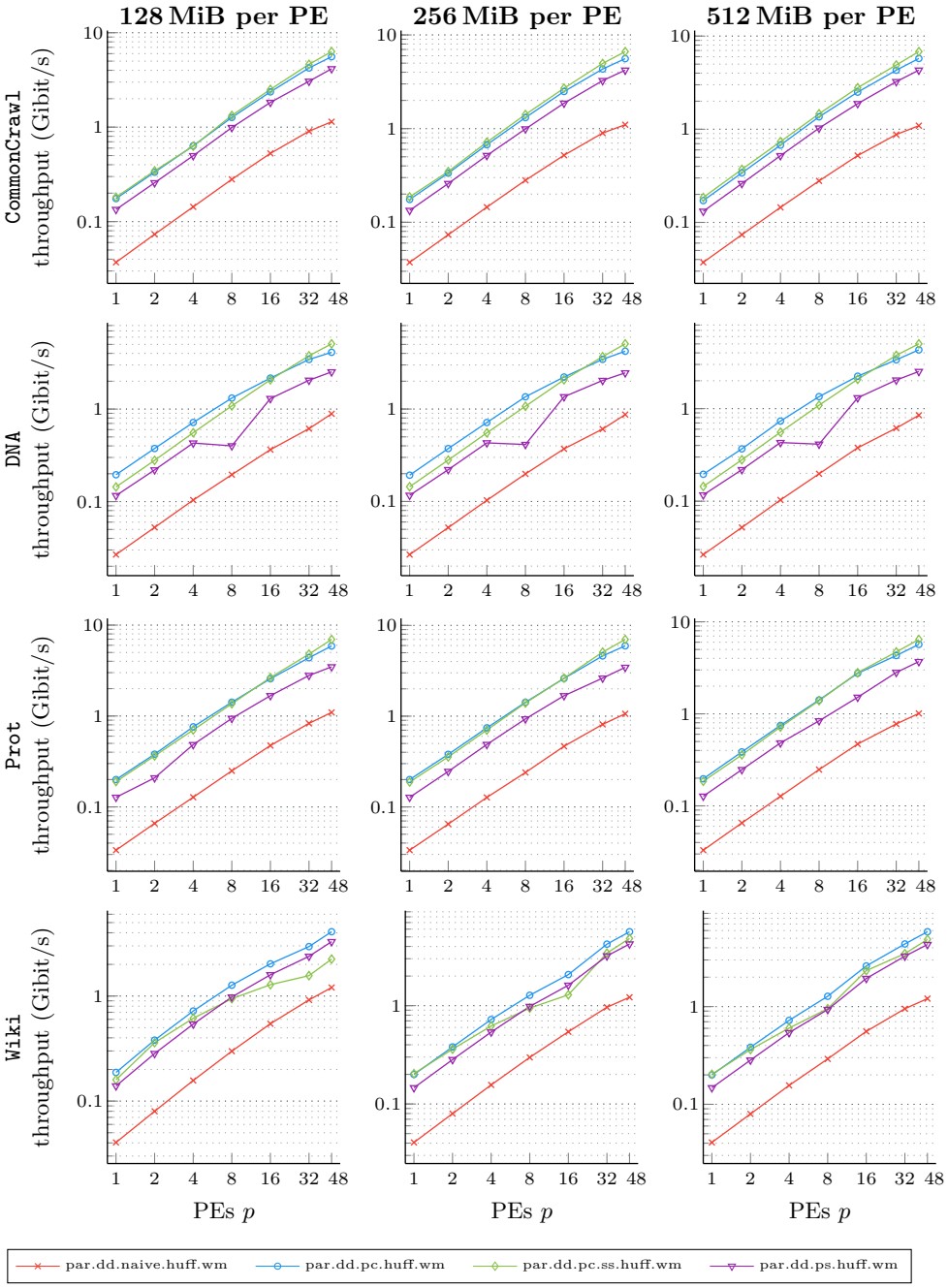


Figure 3.30. Weak scaling Huffman-shaped wavelet *matrix* construction experiments.

Table 3.2. Throughput (Gibits/s) of the Huffman-shaped wavelet tree and wavelet matrix construction algorithms in our weak scaling experiment when using one (t_1) or 48 (t_{48}) PEs and 512 MiB input per PE. Missing values mean that the algorithm could not compute the Huffman-shaped wavelet tree for the input. We mark the highest throughput and speedup for each input in bold.

	`CommonCrawl`			`DNA`			`Prot`			`Wiki`		
	t_1	t_{48}	t_{48}/t_1	t_1	t_{48}	t_{48}/t_1	t_1	t_{48}	t_{48}/t_1	t_1	t_{48}	t_{48}/t_1
par.dd.naive.huff	0.04	0.94	23.59	0.03	0.86	30.02	0.04	0.94	26.31			
par.dd.pc.huff	0.19	6.16	32.40	**0.20**	4.23	21.54	**0.20**	5.76	29.01	**0.21**	**6.91**	**32.18**
par.dd.pc.ss.huff	**0.20**	**7.23**	**36.01**	0.14	**5.07**	**35.10**	0.18	**6.41**	**34.85**	0.21	5.67	26.52
par.dd.ps.huff	0.14	4.47	31.84	0.12	2.53	21.80	0.13	3.71	29.39	0.15	4.83	31.62
par.dd.naive.huff.wm	0.04	1.09	29.26	0.03	0.85	31.97	0.03	1.01	30.60	0.04	1.21	**29.63**
par.dd.pc.huff.wm	0.17	5.75	33.53	**0.20**	4.31	22.04	**0.20**	5.68	28.84	0.20	**5.83**	29.20
par.dd.pc.ss.huff.wm	**0.19**	**6.81**	**36.75**	0.14	**5.01**	**34.70**	0.19	**6.43**	**34.71**	**0.20**	4.87	23.96
par.dd.ps.huff.wm	0.13	4.31	32.73	0.12	2.52	21.56	0.13	3.69	29.01	0.15	4.32	29.23

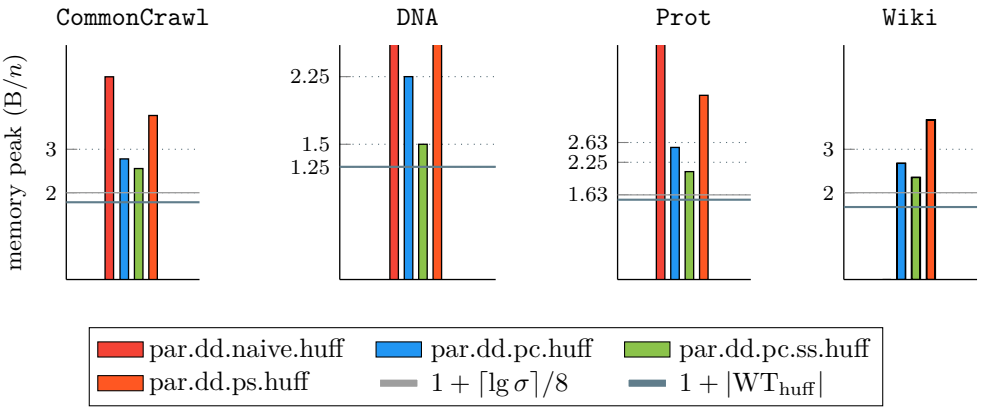


Figure 3.31. Memory peaks of the parallel wavelet *tree* construction algorithms using 48 PEs and 256 MiB input per PE. We show this for the small inputs to have the same setting as for the parallel wavelet tree construction algorithms in Figure 3.12a. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

Memory Peak. We give the memory peak of our parallel Huffman-shaped wavelet tree construction algorithms in Figure 3.31. (The results for the corresponding wavelet matrix construction are shown in Figure 3.32.) On all inputs par.dd.pc.ss.huff is the most memory efficient algorithm. Next is par.dd.pc.huff. The reason is the same as in the sequential case, which we discussed in detail before. The other algorithms, par.dd.naive.huff and par.dd.ps.huff also have the expected memory requirements.

All in all, this makes par.dd.pc.ss.huff the fastest (depending on the input) and most memory efficient parallel Huffman-shaped wavelet tree construction algorithm and par.dd.pc.huff, too, fastest (depending on the input) but not that memory efficient construction algorithm.

3.6 CONCLUSION AND FUTURE WORK

We presented a novel approach wavelet tree and wavelet matrix construction—the *bottom-up* construction. Here, we make great use of the structure of wavelet trees and the dependencies on the histogram of the text. Due to their similarity, we can also extend all wavelet tree construction algorithms to work for the wavelet matrix, too.

Our sequential wavelet tree and wavelet matrix construction algorithms that are based on this approach are the fastest and most memory efficient construction algorithms to date. More precisely, one of our algorithms seq.pc (and its variant seq.pc.ss) is the fastest and most memory efficient one.

Next, we parallelized our algorithms. To this end, we not only created parallel versions of our algorithms but also used domain decomposition, a meta-approach that can work with any sequential wavelet tree and wavelet matrix algorithm. Using our fast

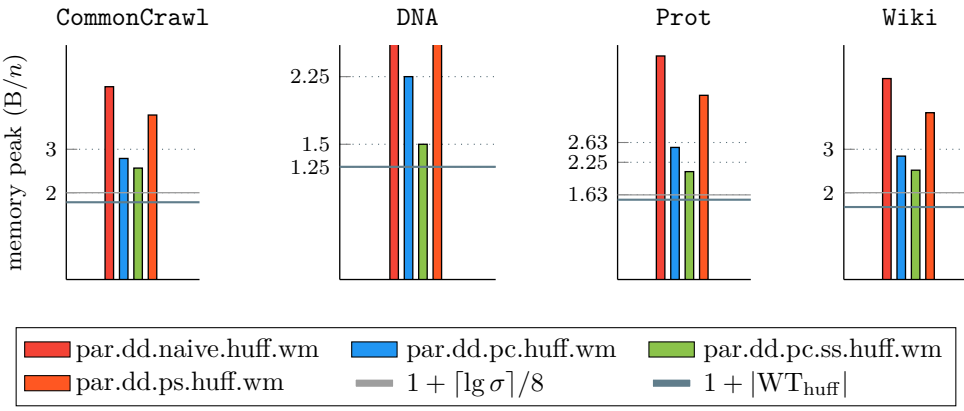


Figure 3.32. Memory peaks of the parallel wavelet *matrix* construction algorithms using 48 PEs and 256 MiB input per PE. We show this for the small inputs to have the same setting as for the parallel wavelet tree construction algorithms in Figure 3.12b. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

sequential construction algorithms in combination with a highly engineered parallel merge, we obtained the fastest and again most memory efficient parallel wavelet tree and wavelet matrix construction algorithm. Again, one of our presented algorithms is both: the fastest and the most memory efficient one. It should also be noted that those wavelet matrix construction algorithms are also the first parallel ones that have been implemented.

Then, we presented semi-external and external memory wavelet tree and wavelet matrix construction algorithms that allow us to overcome the limitations regarding the input size, which is given by the size of the main memory (for our sequential and parallel construction algorithms). The parallelization of the external memory construction algorithms scales well and requires nearly no RAM for the computation.

Additionally, our theoretical result shows the similarity of wavelet trees and wavelet matrices by providing a compact data structure that allows each wavelet tree construction algorithm to compute the wavelet matrix instead in the same asymptotic running time (in Section 2.4). The other direction has also been shown (with slightly worse space requirements) [Din19].

Future Work. Recently, the sequential wavelet tree and wavelet matrix construction algorithm with the best asymptotic running time [Bab+15; Mun+16] has been implemented by Kaneta [Kan18]. Unfortunately, this implementation is not publicly available, thus implementing the algorithms presented by Kaneta would be of great importance to verify the reported results. Dinklage et al. [Din+20] presented distributed memory parallel wavelet tree and wavelet matrix construction algorithm that

are not limited by the hardware limitations of a single machine, which our algorithms are. A hybrid of these two concepts could provide the best of these two worlds, as our shared memory parallel algorithms are faster than the distributed memory parallel construction algorithms. There, the fastest distributed memory wavelet tree construction algorithm requires 3 (`CommonCrawl` and `Wiki`) and 4 (`DNA`) LiDO.small nodes (60 processing elements in total) to achieve the same throughput as our fastest shared memory construction algorithm using one LiDO.small node with 20 processing elements [Din+20, Figure 12]. Since all this work only considers the construction of wavelet trees and wavelet matrices, it remains open how to efficiently answer queries in practice in parallel (shared and distributed memory).

Part II

Distributed Memory Text Index Construction

CHAPTER 4

AN EXCURSION TO SUFFIX SORTING IN MAIN MEMORY

Independently presented by Manber and Myers [MM93] and Gonnet et al. [Gon+92], the *suffix array* (SA) of a text T of length n is a permutation of [0,n) such that

$$T[\mathsf{SA}[i]..n) < T[\mathsf{SA}[j]..n), \text{ for } i < j \in [0, n).$$

In other words, the suffix array of the text T contains the starting positions of the lexicographically sorted suffixes of T. The *inverse* suffix array, denoted by SA^{-1}, is the inverse permutation of the suffix array, i. e., $\mathsf{SA}^{-1}[\mathsf{SA}[i]] = i$ for all $i \in [0, n)$.

From now on, we assume that the last character of the text is a *sentinel* (denoted by \$) that is unique in the text and lexicographically smaller than all other characters. This gives us some convenient properties, e. g., $\mathsf{SA}[0] = n - 1$, but most importantly that no propper suffix is the prefix of another suffix. In addition, is easy to realize in practice.

	0	1	2	3	4	5
T	c	d	c	d	e	\$
SA	5	0	2	1	3	4
LCP	0	0	2	0	1	0

\$
c d c d e \$
c d e \$
d c d e \$
d e \$
e \$

Figure 4.1. Suffix and LCP array for $T = [\,\mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{\$}\,]$. We highlight the LCPs in green (🟢).

The suffix array is often accompanied by the *longest common prefix* (LCP) array, see Figure 4.1. The LCP array contains the length of the longest common prefixes of two lexicographically consecutive suffixes. To be precise $\mathsf{LCP}[0] = 0$ and for all other $i \in [1, n)$ we have

$$\mathsf{LCP}[i] = \max\{s \geq 0 \colon T[\mathsf{SA}[i]..\mathsf{SA}[i] + s) = T[\mathsf{SA}[i-1]..\mathsf{SA}[i-1] + s)\}.$$

We denote the LCP value of any two suffixes starting at text positions $i, j \in [1, n)$ by $\mathsf{lcp}(i, j) = \max\{s \geq 0 \colon T[i..i + s) = T[j..j + s)\}$. Using the LCP array and the inverse suffix array, we can compute LCP values as $\mathsf{lcp}(i, j) = \min\{\mathsf{LCP}[k] \colon \mathsf{SA}^{-1}[i] < k \leq \mathsf{SA}^{-1}[j]\}$. *Range minimum queries* (RMQs) retrieve the minimum of an interval in a static array in constant time, requiring $\mathcal{O}(n)$ preprocessing time, e. g., [FH11].

Part II of this dissertation is based on three publications. This chapter is based on our work on the *DivSufSort* [FK17], which is the fastest main memory suffix array construction algorithm in practice on most inputs, see also Section 4.1. Here, we also motivate work on suffix array construction. Chapter 5 is based on two publications. First, we presented a multitude of distributed suffix array construction algorithms implemented in the distributed big data batch computation framework *Thrill* [Bin+18]. Then, we extended *DivSufSort* to also work in distributed memory [FK19], resulting in a lightweight and still competitively fast distributed algorithm.

While the focus of this part lies on *distributed* memory suffix array construction, we first introduce algorithms working in *main* memory. The reasons are threefold: (1.) The majority of existing work on suffix sorting has been conducted focusing on the main memory, (2.) the motivation for suffix sorting is independent of the model of computation, and (3.) some are the basis for our distributed algorithms, most notably *DivSufSort* [FK19]. So we have a brief look at different construction strategies in Section 4.1, describe the fastest algorithm—*DivSufSort*—in detail in Section 4.2, extend this algorithm to also compute the LCP array in Section 4.3, and finally comparing it with other suffix and LCP array construction algorithms in Section 4.4.

4.1 SUFFIX ARRAY CONSTRUCTION ALGORITHMS

Suffix arrays belong to the most well researched text data structures. Since their introduction, (to the author's best knowledge) 24 different main memory suffix array construction algorithms have been presented. All of these suffix sorting algorithms belong to one type or are a hybrid of two of the following four types of algorithms.

1. *Prefix doubling* algorithms start with the length-1 prefix of each suffix and determine their ranks, i. e., the number of smaller length-1 prefixes. Next, those ranks are used to determine the ranks of the length-2 prefixes. The length of the prefixes is doubled during each iteration until all ranks are unique.

2. *Recursive* algorithms reduce the size of the text until the suffixes can easily be sorted, and then recursively solve the problem for the larger texts.

3. *Induced copying* algorithms sort a small subset of suffixes and induce the lexicographical order of all other suffixes using the sorted suffixes. Baier [Bai16] described another approach they call *grouping* that is another form of induced copying. Here, suffixes are assigned to groups that are then refined and thus implicitly sorted (similar to induced copying), but the groups have more complex properties that are used to obtain a linear time algorithm.

We describe these types in more detail whenever we explain corresponding distributed suffix array construction algorithms. A chronologically sorted overview of all main memory suffix array construction algorithms including their types is depicted in Figure 4.2. For a detailed description of these algorithms, we refer to the survey article by Puglisi et al. [Pug+07] and the recent and complete survey by Bingmann [Bin18, p. 168–176].

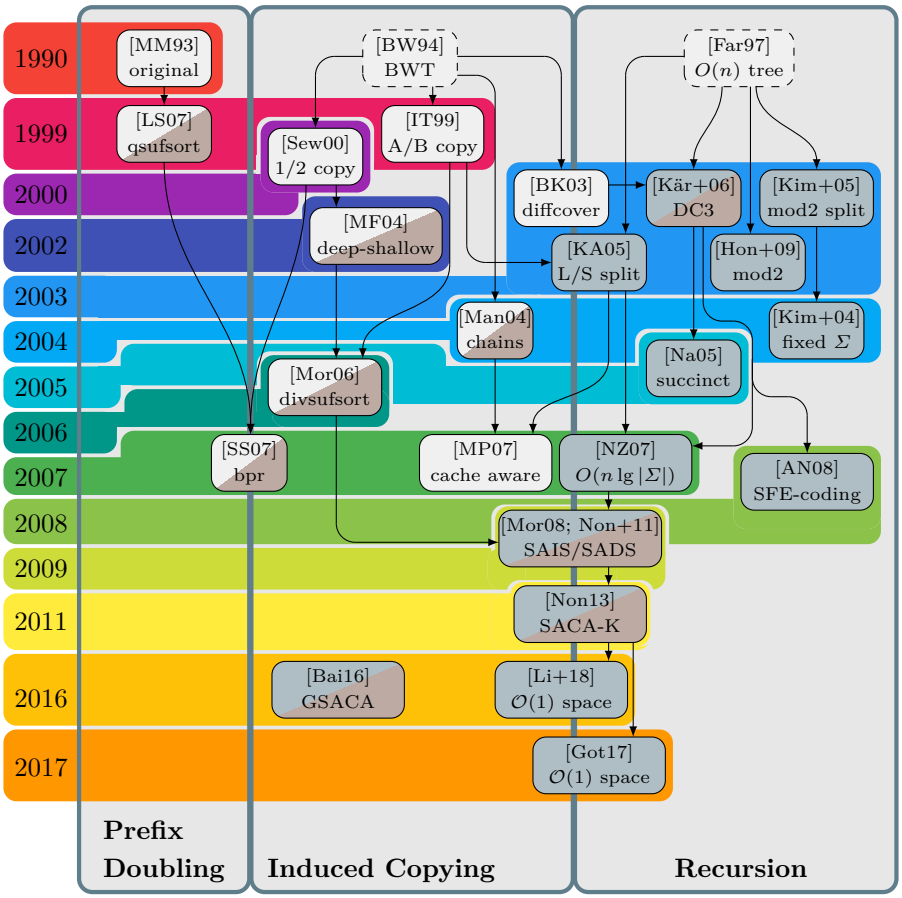


Figure 4.2. Historical development of suffix sorting algorithms in main memory (enhanced and updated, based on [Bin18; Pug+07]). For each algorithm, we cite its most recent publication, and the years on the left hand side show the year of its first publication. In some cases, the years do not match, e. g., due to a later journal publication. Suffix sorting algorithms are marked with a grey background (▢), if they have linear running time, and a brown background (▢), if an implementation is publicly available. We separate the algorithms in four types and connect algorithms that share algorithmic ideas by arrows. The dashed algorithms can be used to compute the suffix array, however, were developed to compute other data structures.

Table 4.1. Characteristics of the texts in the *Pizza & Chili* corpus.

name	size (MiB)	σ
sources	201.10	230
proteins	1129.19	27
dna	385.21	16
english	2107.99	239
dblp.xml	281.10	97

name	size (MiB)	σ
Escherichia_Coli	107.47	15
cere	439.92	5
coreutils	124.83	192
einstein.de.txt	70.76	84
einstein.en.txt	351.03	105
influenza	147.63	15
kernel	155.03	125
para	409.37	5
world_leaders	18.23	63

Evaluation of Practical Main Memory Suffix Sorting

While there exists an extensive amount of work describing the main memory suffix array construction algorithms, there is no prior work comparing them *all*. Out of the 24 main memory suffix array construction algorithms there are (to the best of the author's knowledge) 11 publicly available implementations of sequential main memory suffix sorting algorithms, which we also highlighted in Figure 4.2. It is generally accepted that *DivSufSort* [Mor06] is the fastest suffix array construction algorithm in main memory—despite having a superlinear running time. In this section, we present a practical evaluation of main memory suffix sorting algorithms.

Experimental Setup. We obtained the source code of all implementations and compiled all sources using GCC 8.3.0 and compiler flags `-O3` and `-march=native`. The source code of the suffix sorting algorithms and the code of the testing framework *SACABench* is available at `www.kurpicz.org/sacabench`. We developed SACABench to give all main memory suffix sorting algorithms a test bed, as they have never been compared *all* together. It is easy to extend with new suffix sorting algorithms and provided the plots that we present in this evaluation (which we slightly modified to better fit in this dissertation). We conducted the experiments on LiDO.small nodes and used the texts described in Section 1.4.2 as inputs. As all tested algorithms only run main memory, we also use the text in the *Pizza & Chili* corpus [FN05], which is the de facto standard corpus for text algorithms running in main memory, which we describe in slightly more detail on the next page. For the experiments, we executed each algorithms five times and report the median running time. We start the timer as soon as the input is available in main memory and stop the timer when the suffix array is computed. Some algorithms require minor modifications of the input, e. g., adding a constant number of sentinels. Due to the small effect on the running time, we do not count these manipulations to the running time of the algorithm.

The Pizza & Chili Corpus. We give some basic characteristics of the texts in the Pizza & Chili corpus in Table 4.1. On the right-hand side we list real world texts: *sources* is source code from the Linux kernel and GCC, *proteins* contains protein data from the Swiss-Prot (see Section 1.4.2), *dna* is DNA data from the Gutenberg Project, *english* contains English text from the Gutenberg Project, and *dblp.xml* is XML data containing computer science bibliography. We refer to [FN05] for more detailed descriptions of the texts. Additionally, there are highly repetitive texts in the Pizza & Chili corpus: *Escherichia_Coli*, *cere*, *coreutils*, *einstein.de.txt*, *einstein.en.txt*, *influenza*, *kernel*, and *world_leaders*. For those texts, however, no additional information is available. All texts are available at `http://http://pizzachili.dcc.uchile.cl/`.

Throughput and Memory Requirements. In Figure 4.3, we give the average of all experiments, i. e., the average throughput and memory peaks of all algorithms on all 18 texts. We report the detailed results for each text in Figures 4.4–4.6. In all figures, we give the throughput (upper half) and the memory peak (lower half).

Since we are interested in a suffix sorting algorithm that is fast on all types of input, see Figure 4.3. We now look at the average throughput on all inputs to make some general statements about suffix sorting in main memory. First and foremost, we can report that the folklore that *DivSufSort is the fastest suffix array construction algorithm* is true in our experiment.

DivSufSort achieves an average throughput 7.7 MiB/s, which makes it the fastest one. The second fastest one in SAIS-LITE with an average throughput of 4.7 MiB/s. Interestingly, both implementations have been coded by the same person—Yuta Mori. While the difference in throughput seems quite large (*DivSufSort* is 1.63 times faster than SAIS-LITE), on repetitive inputs SAIS-LITE's throughput is closer to *DivSufSort*'s throughput. The third fastest algorithm is BPR (bucket-pointer refinement), which has an average throughput of 4.54 MiB/s, which is close to the throughput of *SAIS-LITE*. All other tested suffix sorting algorithms do not even achieve halve of *DivSufSort*'s throughput.

When we look at the memory peak during construction, we see that there are three behaviors among the tested algorithms.

1. The first type requires only 5 Bytes of memory per character of the input. This also includes algorithms that require slightly more memory, e. g., a constant amount of memory for each character in the alphabet. This is constant for these implementation that limit the alphabet size to 256. Deep-Shallow, *DivSufSort*, SACA-K, SAIS-LITE, and qsufsort belong to this type.

2. Next, there are algorithms that require 5 Bytes of memory per character of the input in addition to a non-constant number that is less than 1 Byte per character. SADS and SAIS belong to this type.

3. Finally, there are algorithm that require a huge amount of memory compared to the first two types. BPR, DC3, GSACA, and MSufSort belong to this type.

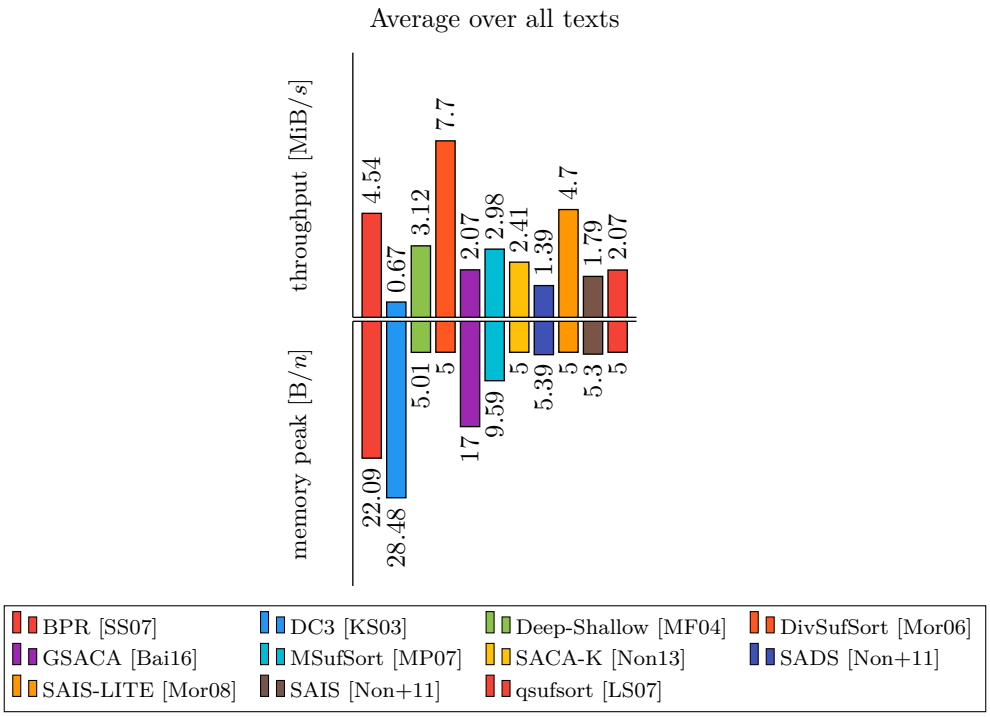


Figure 4.3. Average throughput in MiB per second and memory peak in Bytes per character of the input over all texts given in Table 4.1, which are shown in detail in Figures 4.4–4.6.

The memory peak of BPR, DC3, Deep-Shallow, MSufSort, SADS, and SAIS does depend on the input. All other algorithms have no measurable

While these observations summarize the experiments quite well, we now have a look at the detailed results, which we give in Figures 4.4–4.6.

Most importantly, SAIS-LITE is faster than *DivSufSort* on einstein.en.txt where it is 9.7 % faster, which is a repetitive input. Additionally, on the inputs coreutils and influenza *DivSufSort* is only slightly faster than SAIS-LITE (3.8 % and 8.7 %, respectively).

However, there are also inputs where SAIS-LITE is not even the second fastest suffix sorting algorithm out of the ones we tested. On proteins, SAIS-LITE is slower than BPR, Deep-Shallow, MSufSort, and qsufsort. Here, it is 2.52 times slower than *DivSufSort*. The same is true for dna, `Prot`, and `Wiki`. Note that all algorithms that are faster than SAIS-LITE also require more memory than SAIS-LITE.

In conclusion, *DivSufSort* is not only the fastest but also one of the most memory efficient suffix sorting algorithms that has a publicly available implementation. For that reason, we take a detailed look at it in the next section.

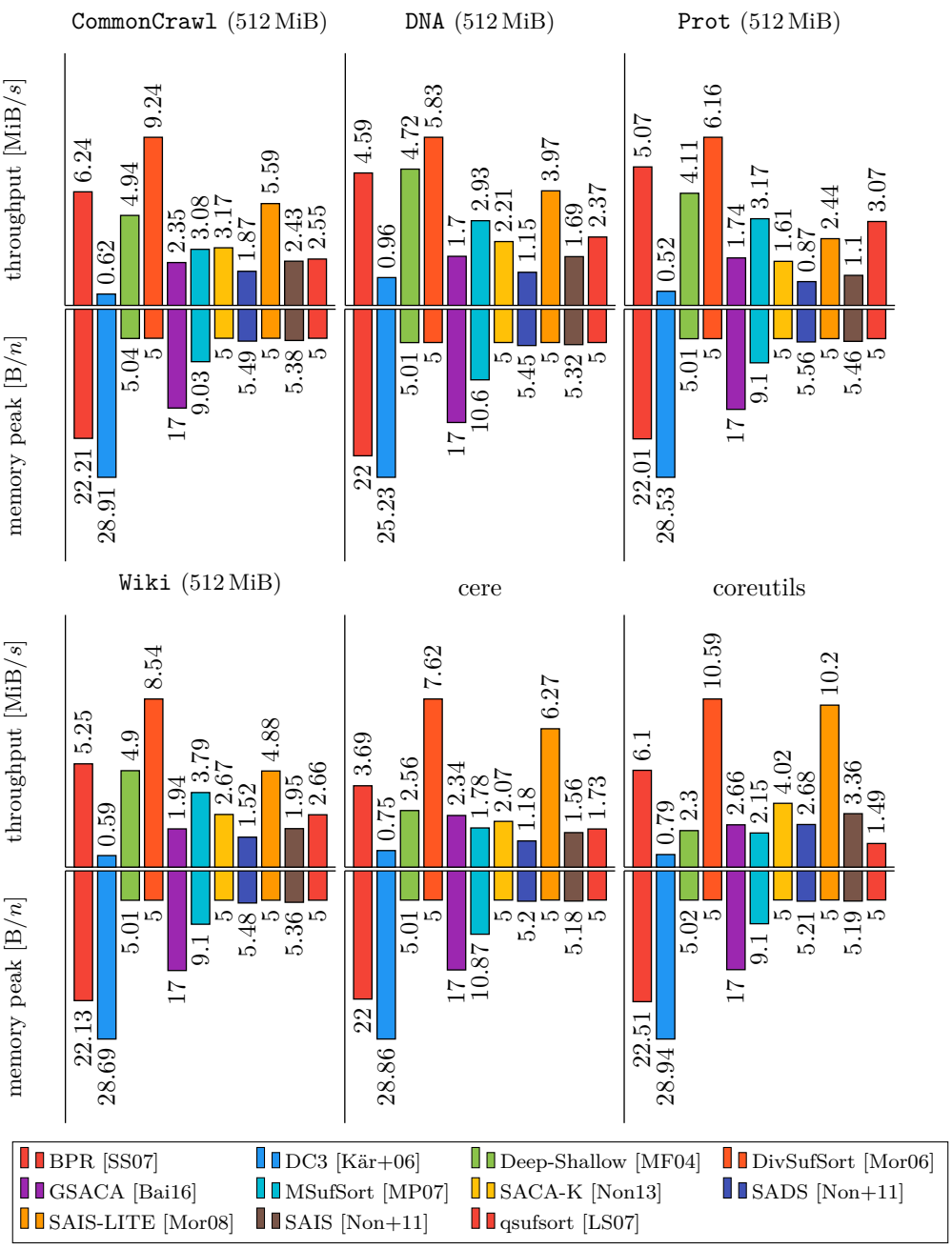


Figure 4.4. Throughput in MiB per second and memory peak in Bytes per character of the input of the main memory suffix sorting algorithms.

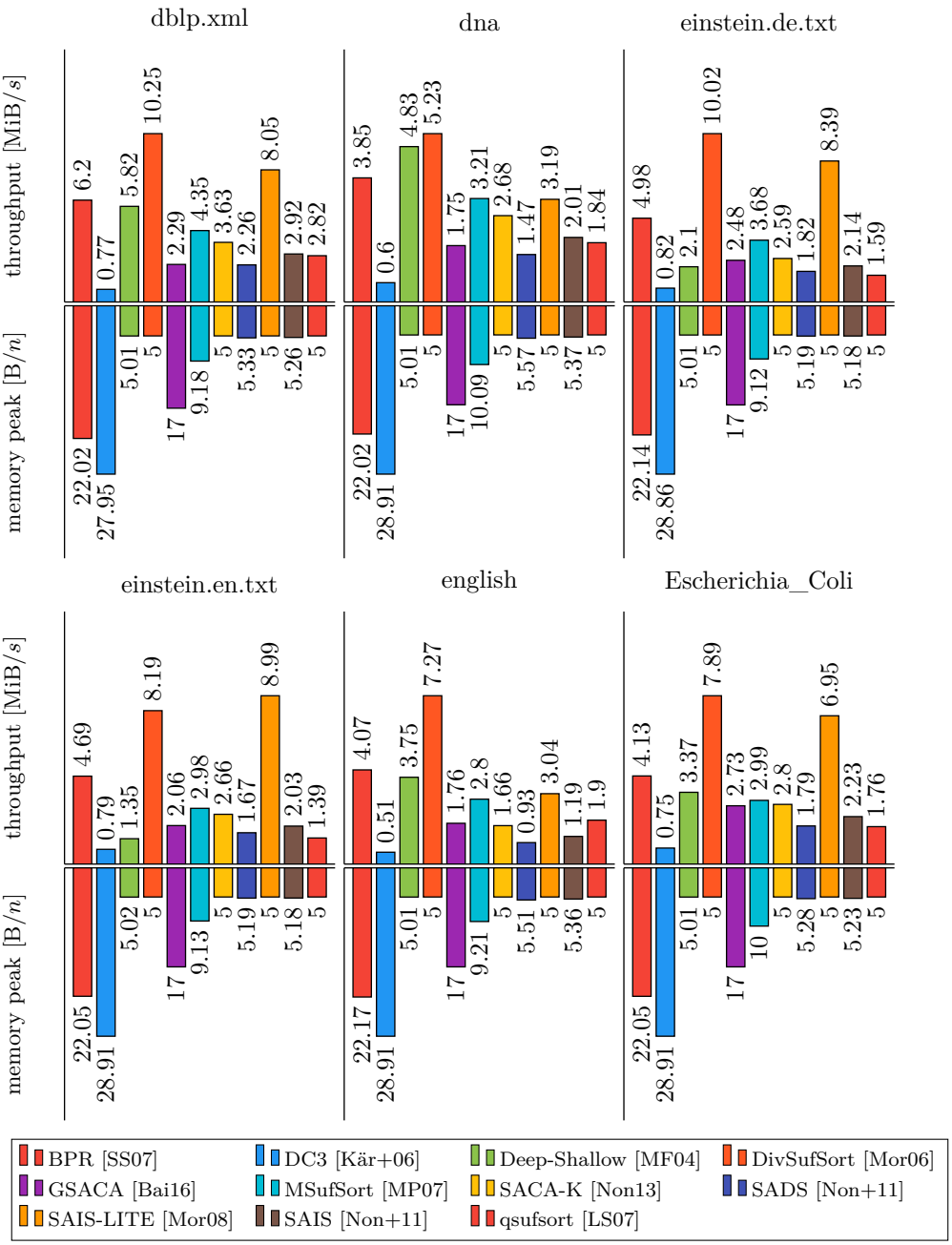


Figure 4.5. Throughput in MiB per second and memory peak in Bytes per character of the input of the main memory suffix sorting algorithms (continuation 1 of 2).

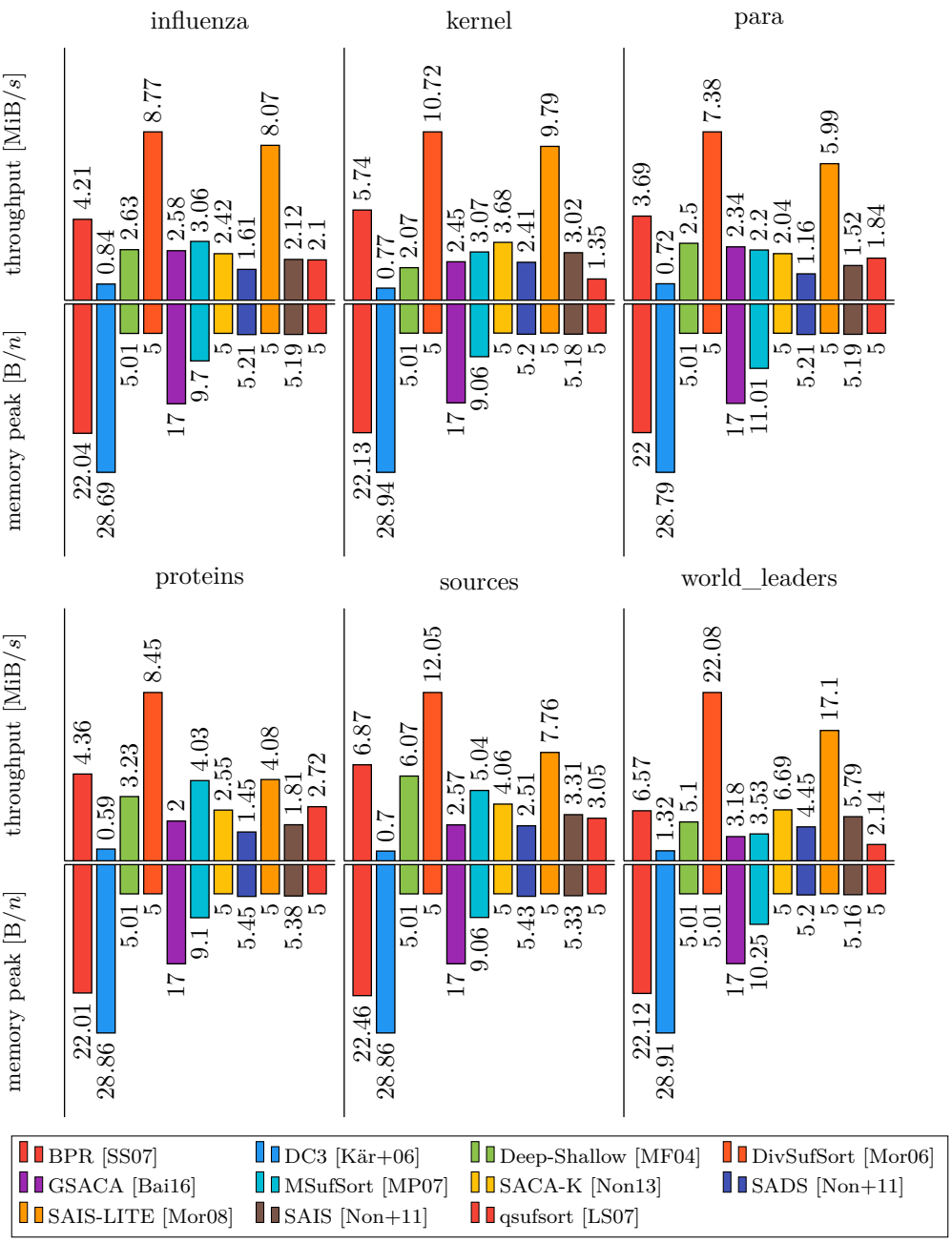


Figure 4.6. Throughput in MiB per second and memory peak in Bytes per character of the input of the main memory suffix sorting algorithms (continuation 2 of 2).

4.2 DISMANTLING DIVSUFSORT

We have seen that the fastest and one of the most space-conscious algorithms to construct the suffix array is *DivSufSort* (implemented by Yuta Mori), which however has only appeared as (almost undocumented) source code, and has never been described in an academic context (except for the paper on which this section is based on [FK17]). The speed and its space-consciousness make *DivSufSort* still the method of choice in many software systems, e. g., in bioinformatics libraries [PS15] and the succinct data structures library (SDSL) [Gog+14a]. Also, DivSufSort has been extended to work in different models of computation, e. g., shared-memory [Lab+17] and distributed memory, see Chapter 5. Ultimately, *DivSufSort* is one of the most used suffix array construction algorithms in practice.

For this reason, we take a detailed look at it in this section. Since it roughly consists of three phases, the section has a similar structure:

1. We start by classifying the suffixes in Section 4.2.1. The classification is easy to compute, as we can assign each suffix to a class based on its length-2 prefix, which can be done in a single scan of the text.

2. Next, we sort all suffixes of one of the classes that contains at most half of all suffixes, in Section 4.2.2. This is the only time we have to sort suffixes by comparing them character-wise.

3. In the last step, which we describe in Section 4.2.3, we can induce the lexicographical order of all other suffixes based on the ones that we have sorted. This requires two scans of the (at the beginning partially filled) suffix array.

The description of the phases is supplemented with line numbers that correspond to lines in the original source code of the DivSufSort implementation by Mori [Mor06], of which we also show the relevant extracts (with matching line numbers) in Appendix A.

4.2.1 Classification of Suffixes

We use the classification introduced by by Itoh and Tanaka [IT99] to distinguish between two *classes* of suffixes (originally called type A and type B suffixes) in combination with a notation established by Kärkkäinen et al. [Kär+17] for a similar classification that is used in SAIS [Non+11] and other suffix array construction algorithms, e. g., [Bin+16a; Got17; Li+18]. Here, each suffix belongs to one class and up to one sub-class. The classification is based on a length-2 prefix of the suffix. We associate the starting position of the suffix with its class:

Definition 4.1. *Let T be a text of length n and $i \in [0, n)$, then the suffix $T[i..n)$ belongs to exactly one of the following two classes.*

(C1) $i \in C^- \iff T[i] > T[i+1]$ *or* $(T[i] = T[i+1]$ *and* $i+1 \in C^-)$ *or* $i = n-1$,

(C2) $i \in C^+ \iff T[i] < T[i+1]$ *or* $(T[i] = T[i+1]$ *and* $i+1 \in C^+)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
Class	$C^{+\triangleright}$	C^-	$C^{+\triangleright}$	C^-	$C^{+\triangleright}$	C^-	$C^{+\triangleright}$	C^-	$C^{+\odot}$	$C^{+\triangleright}$	C^-	C^-	C^-

Figure 4.7. Classification of $T = [\,\mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{c}, \mathtt{d}, \mathtt{d}, \$\,]$, which is our running example for the rest of this section.

The last suffix $T[n-1..n)$ is in C^- by definition, as the empty string is lexicographically smaller than any suffix. Whenever the class of two consecutive suffixes differs, we are interested in the suffix *before* the change. We call $C^{+\triangleright}$ a *sub-class* of C^+, and suffixes in $C^{+\triangleright}$ are often historically called $B^\star$-suffixes [Mor06].

Definition 4.2. *Let T be a text of length n and $i \in [0, n)$, then*

(C3) $i \in C^{+\triangleright} \iff i \in C^+$ *and* $i+1 \in C^-$.

We say "a suffix $T[i..n)$ is in C" if $i \in C$, where C can denote any (sub-)class. For any (sub-)class C, we let C-suffixes denote all suffixes $T[i..n)$ with $i \in C$. Note that the number of suffixes in $C^{+\triangleright}$, i. e., $|C^{+\triangleright}|$, is at most $n/2$, as at most any other suffix can be in C^+ and left of a suffix in C^-. We define $m \coloneqq |C^{+\triangleright}|$. For an example of the classification see Figure 4.7.

Sub-classes are later used to identify fine-grained intervals in the suffix array, which we make heavy use of during our inducing step, see Section 4.2.3. We further need to filter suffixes by their first (two) characters. For any (sub-)class of suffixes C described above, let $\alpha, \beta \in \Sigma$. Then:

- $C_\alpha \coloneqq \{i \in C\colon T[i] = \alpha\}$ and
- $C_{\alpha\beta} \coloneqq \{i \in C_\alpha\colon T[i+1] = \beta\}$.

Using these filters, we can sort suffixes based on their (sub-)class. To this end, we also need to identify all suffixes in C^+ that are not in $C^{+\triangleright}$ i. e., that are followed by a suffix in C^+ instead of a suffix in C^-. For this reason, we introduce the additional class $C^{+\odot} \coloneqq C^+ \setminus C^{+\triangleright}$.

Lemma 4.1. *Let T be a text of size n over an alphabet Σ and $i, j \in [0, n)$, then*

1. $T[i..n) < T[j..n)$, *if* $i \in C^-_{\alpha\beta}$ *and* $j \in C^+_{\alpha\beta}$ *for any* $\alpha, \beta \in \Sigma$, *and*
2. $T[i..n) < T[j..n)$, *if* $i \in C^{+\triangleright}_{\alpha\beta}$ *and* $j \in C^{+\odot}_{\alpha\beta}$ *for any* $\alpha, \beta \in \Sigma$.

Proof. We now prove the first case. By definitions $(C1)$ and $(C2)$ this case occurs if and only if $\alpha = \beta$, as otherwise either $C^-_{\alpha\beta}$ or $C^+_{\alpha\beta}$ is empty. Hence, the LCP value of the suffixes is at least two. Now, let $\gamma = T[i + \mathsf{lcp}(i,j)]$ and $\gamma' = T[j + \mathsf{lcp}(i,j)]$ be the first characters where the suffixes differ. Therefore, we know that $\gamma \leq \alpha$ and

$\gamma' \geq \alpha$—again, due to definitions $(C1)$ and $(C2)$ of the classes. Since the characters differ, at least one of the inequalities is strict, which concludes the proof of the first case.

The argument for the second case works analogously. Since the last suffix of a text is in C^- by definition $(C1)$, we know that $i \in [0, n-1)$ and $j \in [0, n-2)$. We also know that $i+1 \in C^-$, and $j+1 \in C^+$, due to definitions $(C2)$ and $(C3)$. Therefore, $T[i+2] < \beta \leq T[j+2]$, which proves this case. Note that it is possible that $T[i+2] = \epsilon$, i. e., an empty string, if $T[i+1] = \$$. Still, $\epsilon < \beta$ by definition. □

Let $\overrightarrow{C}$ denote the starting positions of the suffixes in C in lexicographical order. In conjunction with Lemma 4.1 this allows us to define the suffix array solely based on the (sub-)classes:

Observation 4.1. *We can express the* SA *of a text over an alphabet* $\Sigma = [0, \sigma)$ *as follows:*

$$\mathsf{SA} = \overrightarrow{C^-_{00}}\,\overrightarrow{C^{+\triangleright}_{00}}\,\overrightarrow{C^{+\odot}_{00}}\,\overrightarrow{C^-_{01}}\,\overrightarrow{C^{+\triangleright}_{01}}\,\overrightarrow{C^{+\odot}_{01}} \ldots \overrightarrow{C^-_{\sigma-1\sigma-1}}\,\overrightarrow{C^{+\triangleright}_{\sigma-1\sigma-1}}\,\overrightarrow{C^{+\odot}_{\sigma-1\sigma-1}}.$$

We call the positions in the suffix array that are covered by $\overrightarrow{C_{\alpha\beta}}$-suffixes *bucket*. Throughout the computation we utilize two additional arrays to store information about the buckets: (1.) BUCKET_A for starting or ending positions of suffixes in C^- of size σ, and (2.) BUCKET_B for starting or ending positions of suffixes in C^+ and suffixes in $C^{+\triangleright}$of size σ^2. For an easier access to suffixes in C^+ and $C^{+\triangleright}$, we say that BUCKET_B$[\alpha, \beta]$ is short for BUCKET_B$[\alpha \cdot \sigma + \beta]$ and BUCKET_BSTAR$[\alpha, \beta]$ is short for BUCKET_B$[\beta \cdot \sigma + \alpha]$, when we identify the characters of the text as integers in $[0, \sigma)$. Information about the (sub-)classes C^+ and $C^{+\triangleright}$ of suffixes can be stored in the same array, as $C^{+\triangleright}_{\alpha\alpha} = \emptyset$ for all $\alpha \in \Sigma$ and $C^+_{\alpha\beta} = \emptyset$ for all $\alpha, \beta \in \Sigma$ with $\alpha > \beta$, due to definitions $(C2)$ and $(C3)$. We sketch this memory layout in Figure 4.8.

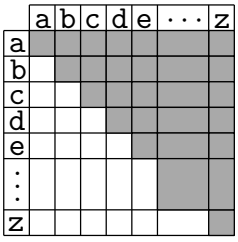


Figure 4.8. BUCKET_B (gray) and BUCKET_-BSTAR represented as a 2-dimensional array.

Initializing DivSufSort

In addition to the classification, which we do not store explicitly, information on the sizes of all (sub-)classes are computed and stored. This initialization of DivSufSort is listed in divsufsort.c. First, we scan T from right to left (line 60), determine the type of each suffix, and store the sizes of the corresponding buckets in BUCKET_A, BUCKET_B and BUCKET_BSTAR (lines 62, 69, and 65). In addition, we store $C^{+\triangleright}$ at the end of SA such that SA$[n-m..n)$ contains $C^{+\triangleright}$ in increasing order, i. e., text order (line 66). We call this part of the suffix array PAb where PAb$[i]$ = SA$[n-m+i]$ for all $0 \leq i < m$ (line 94), see Figures 4.9a and 4.9b, where we highlight PAb in light blue (🔵) on the next page.

Next (lines 81 to 90), we compute the prefix sum over the sizes of C^-_α, $C^+_{\alpha\beta}$, and $C^{+\triangleright}_{\alpha\beta}$ for all $\alpha, \beta \in \Sigma$, such that

$$\mathsf{BUCKET\_A}[\alpha] = \sum_{\beta \in [0,\alpha)} \left(|C^-_\beta| + \sum_{\beta' \in [\beta,\sigma)} |C^+_{\beta\beta'}| \right),$$

i. e., the leftmost position of the bucket, and

$$\mathsf{BUCKET\_BSTAR}[\alpha,\beta] = \sum_{\alpha' \in [0,\alpha],\, \beta' \in [\alpha',\beta]} |C^{+\triangleright}_{\alpha'\beta'}|,$$

i. e., the rightmost position of the bucket with respect only to other suffixes in $C^{+\triangleright}$. Hence, the positions in BUCKET_BSTAR are in $[0, m)$, see Figures 4.9c and 4.9d, where Figure 4.9c remains unchanged. During the sorting step, we do not sort the text positions directly. Instead we sort *references* to these positions. These references are stored in SA[0..m) (line 97). During this step, BUCKET_BSTAR$[\alpha,\beta]$ is updated (line 97), such that at the end of the computation, it contains the leftmost reference corresponding to a suffixes in $C^{+\triangleright}_{\alpha\beta}$. Again, those positions are within the interval [0,m). Then, the reference to the last rightmost (in text order) suffix in $C^{+\triangleright}$ is put at the beginning of its corresponding bucket (line 100). This reference is a special case as it has no successor in PAb that is required for the comparison of two $C^{+\triangleright}$-substrings (see next section), as depicted in Figures 4.9e and 4.9f.

4.2.2 Sorting of Sampled Suffixes

In this section, we describe how the suffixes in $C^{+\triangleright}$ are sorted in three steps. To this end, we need $C^{+\triangleright}$*-substrings*. Let $\text{next}(i) \coloneqq \min\{j > i \colon j \in C^{+\triangleright} \cup \{n-1\}\}$, i. e., the starting position of the next $C^{+\triangleright}$-suffix in text order (or the end of the text if i is the last position). Then, we can define the $C^{+\triangleright}$-substrings as follows.

Definition 4.3 ($C^{+\triangleright}$-substrings)**.** *For any $C^{+\triangleright}$-suffix $i \in C^{+\triangleright}$ the $C^{+\triangleright}_{\alpha\beta}$-substrings is the substring $S \coloneqq T[i..\min\{\text{next}(i)+2, n\})$.*

Now, we consider all $C^{+\triangleright}_{\alpha\beta}$-suffixes in text order, as we obtain their starting positions during the classification. Then, we sort all $C^{+\triangleright}_{\alpha\beta}$-substrings independently for all $\alpha, \beta \in \Sigma$ with $\alpha < \beta$ (lines 134 to 142) using functions defined in sssort.c. We can do so, because $C^{+\triangleright}_{\alpha\beta}$-substrings are implicitly sorted by the first two characters. Then (second step starting at line 146), a partial ISA (named ISAb) is computed, containing the ranks of the partially sorted $C^{+\triangleright}$-suffixes (sorted by their initial $C^{+\triangleright}$-substrings). Using these ranks we compute the lexicographical order of all $C^{+\triangleright}$-suffixes adopting an approach similar to prefix doubling (see Section 5.2). In the last step, we use functions defined in trsort.c (line 159). We augment the approach with *repetition detection* as introduced by Maniscalco and Puglisi [MP07].

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
SA$[i]$	0	0	0	0	0	0	0	0	0	2	4	6	9

(a)

	$	c	d	(c,c)	(c,d)
BUCKET_A	1	0	6	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(b)

	$	c	d										
i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
SA$[i]$	0	0	0	0	0	0	0	0	0	2	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	**0**	**1**	**7**	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(d)

	$	c	d										
i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
SA$[i]$	4	0	1	2	3	0	0	0	0	2	4	6	9

(e)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	**0**

(f)

Figure 4.9. The suffix array and the buckets after the first scan of T are shown in (a) and (b). PAb has a light blue background (🟦) in (a), (c), and (e) and contains the text positions of all $C^{+\triangleright}$-suffixes in *text order*. The buckets in (b) contain the number of suffixes beginning with the corresponding characters. In (d), they are updated such that the first position of each C_α-bucket is stored in BUCKET_A$[\alpha]$ (bold entires). The suffix array does not change during this update, see (c). In (e) we store references to the text positions in SA$[0..m)$, which has a green background (🟩) and update the corresponding BUCKET_BSTAR with the first position in SA[0..m) (bold entry in (f)).

Sorting the $C^{+\triangleright}$-Substrings

All $C^{+\triangleright}_{\alpha\beta}$-substrings in a BUCKET_BSTAR are sorted in-place. The interval of the suffix array that has not been used yet (SA$[m..n-m)$) serves as a buffer, denoted by buf with buf$[i]$:= SA$[m+i]$ for all $i \in [0, n-2m)$, during the sorting (line 133). Sorting can be conducted in parallel. All $C^{+\triangleright}$-substrings are presorted by beeing in the same BUCKET_BSTAR. Therefore, we can sort multiple BUCKET_BSTAR in parallel (see divsufsort.c, lines 105 to 131), as $C^{+\triangleright}$-substrings in other buckets are either all smaller or larger. Here, each processing element gets a buffer of size $\lfloor |\text{buf}|/p \rfloor$, where p is the number of processing elements. All following line numbers refer to sssort.c.

	$	c						d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
SA$[i]$	3	0	$\tilde{1}$	$\tilde{2}$	4	0	0	0	0	2	4	6	9

(a)

Ref.	Text Pos.	$C^{+\triangleright}$-substring
3	6	`cdcc`
0	0	`cdcd`
1	2	`cdcd`
2	4	`cdcd`
4	9	`cdd$`

(b)

Figure 4.10. The lexicographically sorted references in green (🟢) of the $C^{+\triangleright}$-substrings in SA[0..m) in blue (🔵) in (a). For readability we write $\tilde{i}$ if i is bitwise negated. The content of the buckets is not changed in this step. The references, their corresponding (lexicographically sorted) text positions and the $C^{+\triangleright}$-substrings are shown in (b).

In the default configuration we only sort 1024 elements at once (see SS_BLOCKSIZE, e. g., line 763). If the size of buf is smaller than 1024 or the size of the current bucket, the bucket is divided in smaller subbuckets which are then sorted and merged (see line 767, splitting due to the buffer size and the loop at line 770 splitting with respect to the number of elements). Lines 789 to 802 are used to merge the last considered subbuckets. If the currently sorted bucket contains the last $C^{+\triangleright}$-substring, then it is moved to the corresponding position (lines 811 and 813).

The heavy lifting is done by the function ss_mintrosort, which is an implementation of *Introspective Sort* (ISS) [Mus97]. It sorts all $C^{+\triangleright}$-substring within the interval [`first`, `last`] (line 310). ISS uses *Multikey Quicksort* (MKQS) [BS97] and *Heapsort* (HS). MKQS is used $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times to sort an interval before HS is used (if there are still elements in the interval that have been equal to the pivot each time, see line 333). MKQS divides each interval into three subintervals with respect to a pivot element. The first subinterval contains all substrings whose k-th character is smaller than the pivot, the second subinterval contains all substrings whose k-th character is equal to the pivot, and the last subinterval contains all substrings whose k-th character is greater than the pivot. We call k the depth of the current iteration (line 332). ISS is not implemented recursively; instead, a stack is used to keep track of the unsorted subintervals and the smaller subintervals are always processed first. This guarantees a maximum stack size of $\lceil \lg \ell \rceil$, where ℓ is the initial interval size [Meh84, p. 67]. The subintervals containing the substrings whose k-th character is not equal to the pivot are sorted using MKQS $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times before using HS, where now `last` and `first` refer to the first and last positions of these intervals (lines 414 and 428). Unsorted (sub)bucket smaller than a threshold (8 in the default configuration) are sorted using *Insertionsort* and then marked sorted (line 326). Here, we use the function ss_compare to compares two $C^{+\triangleright}$-substrings starting at the current depth.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	-1	0	1	2	-1	3	3	3	0	4	4	6	9

Figure 4.11. ISAb contains the inverse suffix array of the sorted $C^{+\triangleright}$-substrings. ISAb[i] = SA[$m + i$] for all $0 \leq i < m$ in amber (🟡). While computing ISAb, we mark completely sorted intervals in SA[$0..m-1$] in green (🟢). The leftmost position of a sorted interval of length k is changed to $-k$, e. g., SA[0] and SA[4].

There are $C^{+\triangleright}$-suffixes that are not sorted completely by their initial $C^{+\triangleright}$-substrings e.g., in our example $T = [\,\mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{d}, \mathtt{c}, \mathtt{c}, \mathtt{d}, \mathtt{d}, \$\,]$ the $C^{+\triangleright}$-substring cdcd occurs three times, see Figure 4.10. Hence, we cannot sort the corresponding $C^{+\triangleright}$-suffixes just using the $C^{+\triangleright}$-substring. $C^{+\triangleright}$-substrings that are not unique, are marked by storing their bitwise negated reference (line 178). Only the first reference in an interval of unsorted substrings is stored normally to identify the beginning of said interval (line 178). The idea of sorting the suffixes in an (α, β)-bucket up to a certain depth is similar to the approach of Manzini and Ferragina [MF04], who sort the suffixes up to a certain LCP value before switching techniques.

Computing the Partial Inverse Suffix Array.

After the $C^{+\triangleright}$-substrings are sorted, we compute the ISA for the partially sorted $C^{+\triangleright}$-substrings (lines 146 to 156). The inverse suffix array for the $C^{+\triangleright}$-suffixes is stored in SA[$m..2m$) and referred to as ISAb with ISAb[i] = SA[$m + i$]. If $m > n/3$, ISAb overlaps with PAb. This does not matter, as we do not require the text positions at this point any more. ISAb[i] contains the *rank* of the i-th $C^{+\triangleright}$-suffix, i. e., the number of lexicographically smaller $C^{+\triangleright}$-suffixes. All references to line numbers in this subsection refer to divsufsort.c. We scan SA[$0..m$) from right to left (line 146) and distinguish between bitwise negated references (values < 0, starting at line 154) and non-negated references (values ≥ 0, starting at line 147). In the first case, we have reached an interval where we have references of suffixes which could not be sorted comparing only the $C^{+\triangleright}$-substring. To each of those suffixes we assign the greatest feasible rank, i. e., $m - i$, where i is the number of lexicographically greater suffixes (similar to Larsson and Sadakane [LS07]). In addition we also store the bitwise negation of the references, i. e., the original reference. In the other case (a value ≥ 0) we simply assign the correct rank to the $C^{+\triangleright}$-suffix. Whenever we scan an interval of completely sorted $C^{+\triangleright}$-suffixes, we mark the first position of the interval in SA[$0..m$) with $-k$, where k is the size of the interval (line 150). Now we can identify all sorted intervals as they start with a negative value whose absolute value is the length of the interval. All remaining positions can still contain values from previous computations. In our example (see Figure 4.11) we have two fully sorted intervals of length 1 at SA[0] and SA[4], and an only partially sorted interval in SA[1..3].

Sorting the $C^{+\triangleright}$-Suffixes.

In the last part of the $C^{+\triangleright}$-suffix sorting in DivSufSort we compute the correct ranks of all $C^{+\triangleright}$-suffixes and store them in ISAb. During this step, we only require information about the ranks of the suffixes and have no random access to the text, i. e., PAb is not required any more. All line numbers in this section refer to trsort.c. Using ISAb, we compute the ranks of all $C^{+\triangleright}$-suffixes using an approach similar to prefix doubling [LS07]. Instead of doubling the length of the suffixes we double the number of considered $C^{+\triangleright}$-substrings that can have an arbitrary length (line 563). Here, ISAd$[i]$ refers to the rank of the $(i + 2^k)$-th $C^{+\triangleright}$-suffix, where k is the current iteration of the doubling algorithm. Obviously, we need to update the ranks when we double the number of considered substrings, i. e., compute the new ranks for the $C^{+\triangleright}$-suffixes. Since the ranks in the ISA are given in text order, we can access the rank of the next (in text order) $C^{+\triangleright}$-substring for any given substring.

Repetition Detection.

We use Quicksort (QS) to sort the $C^{+\triangleright}$-substrings by their ranks. This allows us to use the *repetition detection* introduced by Maniscalco and Puglisi [MP07] (see line 452 for the identification and the function tr_copy for the computation of the correct ranks). A *repetition* in T is a substring $T[i..i + rp)$ with $r \geq 2, p \geq 0$ and $i, i + rp \in [0, n)$ such that $T[i..i + p) = T[i + p..i + 2p) = \cdots = T[i + (r - 1)p..i + rp)$. Here, we call r is the number of repetitions and p is the *period*. Those repetitions are a problem if $T[i..n)$ is a $C^{+\triangleright}$-suffix, since then $T[kp..n)$ is a $C^{+\triangleright}$-suffix for all $k \leq r$. We can simply sort all those suffixes by looking at the first character not belonging to the repetition ($T[i + rp + l] \neq T[i + l]$). If $T[i + rp + l] < T[i + l]$ then $T[i + (r - 1)p + 1, i + rp] < T[(i - 1) + (r - 1)p + 1, (i - 1) + rp]$ for all $1 < i \leq r$. The analogous case is true for $T[i+rp+l] > T[i+l]$, i. e., $T[i+(r-1)p+1, i+rp] > T[(i-1)+(r-1)p+1, (i-1)+rp]$ for all $1 < i \leq r$. This is done in lines 276 (and 282), where we increase (and decrease) the ranks of all suffixes in the repetition. The identification of a repetition is supported by QS, which divides each interval into three subintervals (like MKQS). We choose the median rank of the $C^{+\triangleright}$-suffixes that are considered during this doubling step as the pivot element for QS (line 455). If the (current) rank of the first $C^{+\triangleright}$-suffix in the subinterval (considered in this doubling step) is equal to the pivot element, i. e., ISAb$[i]$ = ISAd$[i]$ where i is the first $C^{+\triangleright}$-suffix in the interval, then we have found a repetition (line 452, where tr_ilg denotes the logarithm of the interval size, which is also the number of iterations until Heapsort is used instead of QS).

From the Inverse Suffix Array to sorted $C^{+\triangleright}$-Suffixes

Now we have computed the ISA of all $C^{+\triangleright}$-suffixes (stored in ISAb), i. e., we have all $C^{+\triangleright}$-suffixes in lexicographic order. From this point on, all line numbers refer to divsufsort.c, again. Next (see loop starting at line 162), we scan T from right to left, and when we read the i-th $C^{+\triangleright}$-suffix at position j, we store j at position SA[ISAb$[i]$]. Since we use all the $C^{+\triangleright}$-suffixes to induce the C^{+}-suffixes (and we do not want to induce C^{-}-suffixes during the first inducing phase) we store the bitwise negation of

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	-1	-4	1	0	-1	3	2	1	0	4	4	6	9

(a)

	0	1	2	3	4	5	6	7	8	9	10	11	12
	c	d	c	d	c	d	c	d	c	c	d	d	$
	6̃	4̃	2̃	0	9	3	2	1	0	4	4	6	9

(b)

	$	c						d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	6̃	4̃	6̃	4̃	2̃	0	9	1	0	4	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	**1**	**6**
BUCKET_BSTAR	-	-	-	-	**1**

(d)

Figure 4.12. ISAb in amber (🟡) in (a) and (b) contains the ranks of all $C^{+\triangleright}$-suffixes. The lexicographically sorted text positions of the $C^{+\triangleright}$-suffixes are shown green (🟢) in (b). Each text position i is bitwise negated if $T[i-1..n)$ has type C^-. In (c) all text positions of the $C^{+\triangleright}$-suffixes are at their correct position in SA$[0..n-1]$ (🟢). The buckets d contain the leftmost position of the corresponding suffixes.

j if $T[j-1..n)$ has type C^- (line 167). FiguresFigures 4.12a and 4.12b show the transition in SA$[0..m)$ for our example. Now, SA$[0..m)$ contains the text positions of all $C^{+\triangleright}$-suffixes in lexicographic order. Next (see loop beginning at line 173), we need to put these text positions at their correct position in SA$[0..n)$ (line 182). While doing so, we update BUCKET_B and BUCKET_BSTAR such that they contain the rightmost position of the corresponding buckets (lines 177 and 185). Figures 4.12c and 4.12d show this step for our running example.

4.2.3 Inducing of Suffixes

Due to the types of the suffixes, we know that in any (α, β)-bucket the C^--suffixes are lexicographically smaller than the C^+-suffixes, and that $C^{+\triangleright}$-suffixes are lexicographically smaller than $C^{+\odot}$-suffixes, which allows us to express the suffix array as buckets, see Observation 4.1. We also know that in lexicographic order, all consecutive intervals of C^+-suffixes are left of at least one $C^{+\triangleright}$-suffix and all C^--suffixes are right of at least one C^+-suffix that starts with the same character, see Figure 4.7 for an example. Now we scan SA twice: once from right to left, where all C^+-suffixes are induced (we can skip all parts of SA containing only C^--suffixes), and then from left to right to induce all C^--suffixes. All following line numbers refer to difsufsort.c. A step-by-step example is given in Figure 4.13.

During the inducing of the C^+-suffixes, i. e., the first scan of SA (see loop starting at line 205), whenever we read an entry i in SA such that $i > 0$ (line 211), we store the entry $i-1$ at the rightmost free position (a position in which a correct text position

has not been stored yet) in the $(T[i-1], T[i])$-bucket (line 220). If $T[i-2] > T[i-1]$, then $T[i-2..n)$ is an C^--suffix, which is not induced during the first scan, but the bitwise negated value of $i-1$ is stored instead (line 217). Every position is overwritten with its bitwise negated value. If the position was already bitwise negated, i. e., it has been induced and the corresponding suffix has type C^-, it is considered during the next scan (line 226) and we negate it bitwise as preparation for the second scan. After the first scan, all suffixes that have been used for inducing are represented by their bitwise negated position whereas all other suffixes are represented by their position, i. e., a positive integer. It should be noted that all induced suffixes are lexicographically smaller than the suffix they are induced from: if we induce from an (α, β)-bucket, we know that $\alpha < \beta$, since we are considering C^+-suffixes. In addition, we only induce in (α, β)-buckets with $\alpha < \beta$; only C^+-suffixes are considered during this scan.

Before SA is scanned a second time, $n-1$ is stored at the beginning of the $T[n-1]$-bucket (line 234). If $T[n-2..n)$ has type C^-, we store $n-1$ (we want to induce $T[n-2..n)$ during the second scan). Otherwise, we store the bitwise negation of $n-1$.

During the second scan of SA (see loop starting at line 236), whenever an entry i of SA is smaller than 0 it is overwritten by its bitwise negated value, i. e., the position of the suffix in the correct position in the suffix array (line 249). Whenever $i > 0$ (line 237) the suffix $T[i-1..n)$ is induced at the leftmost free position in the $T[i-1]$-bucket (line 243). Since all remaining suffixes are induced during this scan, it is sufficient to identify the border using the α-buckets, i. e., the value stored in BUCKET_A$[\alpha]$. If the induced suffix is inducing a C^+-suffix, its bitwise negated value is induced instead (line 240). At the end of the scan, SA contains the indices of all suffixes in lexicographic order. Hence, we have compute the suffix array, which we also show for our running example in the last row of Figure 4.13, which concludes our description of *DivSufSort*.

4.2.4 Running Time and Memory Requirements

Now, we briefly look at the running time and memory requirements. The classification of suffixes (Section 4.2.1) requires $\mathcal{O}(n)$ time, as we just have to scan the text once. Sorting the $C^{+\triangleright}$-suffixes (Section 4.2.2), however, requires $\mathcal{O}(n \lg n)$ time due to the prefix doubling like approach. We have to sort $\mathcal{O}(m) = \mathcal{O}(n)$ $C^{+\triangleright}$-suffixes, which requires $\mathcal{O}(n \lg n)$ time, when done as described, which is very fast in practice, as shown in Section 4.1. Alternatively, one can use a recursive approach to sort the $C^{+\triangleright}$-suffixes, which requires $\mathcal{O}(n)$ time, e. g., [Got17; Li+18; Non+11; Non13]. Finally, inducing of the suffixes (Section 4.2.3) can be done in $\mathcal{O}(n)$ time, because we just have to scan the suffix array twice. Therefore, *DivSufSort* has a running time of $\mathcal{O}(n \lg n)$.

DivSufSort requires $\mathcal{O}(\sigma^2)$ words of space in addition to the input and the space for the suffix array to store the additional information (Section 4.2.2). Admittedly, the way it is implemented uses additional n bits of space to mark the suffixes, i. e., bitwise negating them during inducing (Section 4.2.3). This is realized by using *signed* integers; we can only use all but one bit of each position in SA, implicitly using n bits. This can be avoided by storing more additional information in the beginning as we describe in Section 5.4, when we present a distributed variant of *DivSufSort*.

		Scanned Interval													BUCKET_A[$]	BUCKET_A[c]	BUCKET_A[d]	BUCKET_B[c, c]	BUCKET_BSTAR[c, d]	
	i	0	1	2	3	4	5	6	7	8	9	10	11	12						
	SA[i]	6̃	4̃	6̃	4̃	2̃	0	9	1	0	4	4	6	9	0	1	7	1	1	
	SA[i]	6̃	4̃	6̃	4̃	2̃	0	9	1	0	4	4	6	9	0	1	7	1	1	
	SA[i]	6̃	**8̃**	6̃	4̃	2̃	0	**9̃**	1	0	4	4	6	9	0	1	7	**0**	1	
	SA[i]	6̃	8̃	6̃	4̃	2̃	**0̃**	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	6̃	8̃	6̃	4̃	**2**	0̃	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	6̃	8̃	6̃	**4**	2	0̃	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	6̃	8̃	**6**	4	2	0̃	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	6̃	**8**	6	4	2	0̃	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	**12**	8	6	4	2	0̃	9̃	1	0	4	4	6	9	0	1	7	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	1	0	4	4	6	9	**1**	1	7	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	**11**	0	4	4	6	9	1	1	8	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	11	**7**	4	4	6	9	1	1	9	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	11	7	**5**	4	6	9	1	1	10	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	11	7	5	**3**	6	9	1	1	11	0	1	
	SA[i]	12	8	6	4	2	0̃	9̃	11	7	5	3	**1**	9	1	1	**12**	0	1	
	SA[i]	12	8	6	4	2	**0**	9̃	11	7	5	3	1	9	1	1	12	0	1	
	SA[i]	12	8	6	4	2	0	**9**	11	7	5	3	1	9	1	1	12	0	1	
	SA[i]	12	8	6	4	2	0	9	11	7	5	3	1	**10**	1	1	**13**	0	1	

First Induction Phase

Second Induction Phase

Figure 4.13. During the first phase, we induce C^+-suffixes and only scan intervals where C^+- and $C^{+\triangleright}$-suffixes occur. Each of those intervals ends left of the succeeding α-bucket. Its borders are stored in the corresponding BUCKET_BSTAR (both highlighted in cyan, the right border is not part of the interval). After the first phase we put the last suffix at the beginning of its corresponding bucket. During the second phase we scan the whole array, as we also store the bitwise negation of all entries that have already been used for inducing. The currently considered entry is marked lime (). The entries highlighted orange () are the positions where a value is induced. The bucket that contains the position is highlighted in the same color. Entries that have changed are bold in the following row.

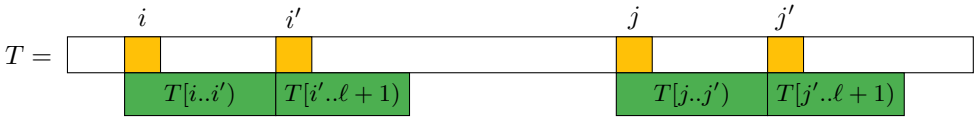


Figure 4.14. Let $T[i..n), T[j..n), T[i'..n)$ and $T[j'..n)$ be $C^{+\triangleright}$-suffixes such that there is no $C^{+\triangleright}$-suffix $T[k..n)$ with $i < k < i'$ or $j < k < j'$, and let the LCP value of $T[i..n)$ and $T[j..n)$ be $\ell = \mathsf{lcp}(i,j) + i$. Then the LCP value of $T[i'..n)$ and $T[j'..n)$ is $\mathsf{lcp}(i',j') = \ell - i' = \mathsf{lcp}(i,j) - (i'-i)$.

4.3 INDUCING THE LCP ARRAY

We now show how to modify *DivSufSort* such that it also computes the LCP array in addition to the suffix array. To do so, we extend DivSufSort at three points of the computation of the suffix array. First, we need to compute the LCP values of all suffixes in $C^{+\triangleright}$. Next, during the inducing step, we also induce the LCP values for C^{-}- and C^{+}-suffixes. For this we utilize a technique also described in [Bin+16a; Fis11] that allows us to answer RMQs on the LCP array using only a stack [GO11]. Last, we compute the LCP values of suffixes at the border of buckets, as those values cannot be induced.

Recall that the LCP value of two arbitrary suffixes $T[i..n)$ and $T[j..n)$ is denoted by $\mathsf{lcp}(i,j)$. We need the following additional definition: Given an array A of length ℓ and $0 \le i \le j \le \ell$, a *range minimum query* $\mathsf{RMQ}_A[i,j]$ asks for the minimum in $A[i,j]$, in symbols: $\mathsf{RMQ}_A[i,j] = \min\{A[k] \colon i \le k \le j\}$.

4.3.1 Computing the LCP Values of the $C^{+\triangleright}$-Suffixes

After the sorting of the $C^{+\triangleright}$-suffixes (right before the $C^{+\triangleright}$-suffixes are placed at their correct position in $\mathsf{SA}[0..n)$), all $C^{+\triangleright}$-suffixes are lexicographically sorted in $\mathsf{SA}[0..m)$. There are two cases regarding m (the number of $C^{+\triangleright}$-suffixes). If $m > n/3$, we have overwritten the text positions of the $C^{+\triangleright}$-suffixes in PAb with ISAb. In this case we must compute the LCP values naively. For all tested instances (see Section 4.4) we have $m \le n/3$. Otherwise (we still know the text positions of all $C^{+\triangleright}$-suffixes), we compute their LCP values using a sparse version of the Φ-algorithm [Kär+09], based on Observation 4.2, which was also used implicitly in [Bin+16a; Fis11].

Observation 4.2. *If $T[i..n), T[i'..n), T[j..n)$ and $T[j'..n)$ are $C^{+\triangleright}$-suffixes such that $i < i', j < j'$ and there is no other $C^{+\triangleright}$-suffix $T[k..n)$ such that $i < k < i'$ or $j < k < j'$, then $\mathsf{lcp}(i',j') \ge \mathsf{lcp}(i,j) - (i'-i)$.*

We can compute this lower bound for the LCP value as we know the distance (in the text) of two $C^{+\triangleright}$-suffixes, i. e., $\mathsf{PAb}[i] - \mathsf{PAb}[j]$ is the distance of the i-th and j-th $C^{+\triangleright}$-suffix with $1 \le i \le j \le m$. See Figure 4.14 for an example. Algorithm 4.1 shows the *sparse* version of the Φ-algorithm. The difference to the original algorithm [Kär+09] is that the next considered suffix is an arbitrary number of character shorter

Algorithm 4.1. Sparse Φ-Algorithm

Input : T, m, SA, ISAb = SA$[m..2m-1]$, PAb = SA$[n-m..n-1]$ and LCP, PHI = LCP$[m..2m-1]$ PHI = DELTA$[n-m..n-1]$.
Output : LCP$[0..m-1]$ contains the LCP values of the $C^{+\triangleright}$-suffixes.

```
1 PHI[SA[0]] = −1
2 for i = 1; i ≤ m − 1; i = i + 1 do
3 |  PHI[SA[i]] = SA[i − 1]
4 |  DELTA[i − 1] = PAb[i] − PAb[i + 1]
5 for i = 0, p = 0; i < m; i = i + 1 do
6 |  while T[PAb[i] + p + 1] = T[PAb[PHI[i]] + p + 1] do
7 |  |  p = p + 1
8 |  PHI[i] = p and p = max {0, p − DELTA[i]}
9 for i = 0; i < m; i = j + 1 do LCP[ISAb[i]] = PHI[i];
```

than the previous one, which is handled by Observation 4.2. The computation of the LCP values does not require any additional memory except for the n words for LCP, where we temporarily store additional data.

First (lines 1 to 4 of Algorithm 4.1), we fill PHI (stored in LCP$[m..2m]$) such that PHI$[i]$ contains the text position of the suffix that is lexicographically consecutive to the i-th suffix (text position). In DELTA$[i]$ (stored in LCP$[n-m..n]$) we store the text distance of the i-th and $(i+1)$-th $C^{+\triangleright}$-suffix (text occurrence), i. e., PAb$[i+1]-$PAb$[i]$. Then (lines 5 to 8), we compute the sparse LCP array using Observation 4.2. As we store the LCP values in PHI in text order, we need to rewrite them to LCP (line 9).

4.3.2 Inducing LCP Values in Addition to the Suffix Array

During the inducing of the C^+-suffixes (see Section 4.2.3), whenever a suffix is induced at position u in SA and there is already a suffix at position $u+1$ in the same (α,β)-bucket, there are two cases:

1. The suffixes T[SA$[u]..n$) and T[SA$[u+1]..n$) have been induced from suffixes T[SA$[v]..n$), T[SA$[w]..n$) in the same (α,β)-bucket; in this case LCP$[u+1]$ = RMQ$_{\text{LCP}}[v+1,w]+1$.
2. Otherwise, the LCP value is either 2 if and only if T[SA$[v]..n$), T[SA$[w]..n$) are in the same α-bucket and 1 if not.

The computation of the LCP values during the inducing of the C^--suffixes works analogously. This leads to the following observation for the general case:

Observation 4.3. *Let* SA$[u]=i$, SA$[u+1]=j$, SA$[v]=i+1$ *and* SA$[w]=j+1$ *such that* $T[i..n)$ *and* $T[j..n)$ *are in the same* α*-bucket, and* $u+1<v$ and $u+1<w$ *or* $w<u$ and $v<u$. *Then* LCP$[u+1]$ = RMQ$_{\text{LCP}}[\min\{v,w\}+1, \max\{v,w\}]+1$.

Unfortunately, not all LCP values can be induced this way. The missing cases are covered in Section 4.3.3. Instead of using a dynamic RMQ data structure, we can

i	0	1	2	3	4	5	6
$\mathsf{A}[i]$	4	2	0	1	4	3	2

(a) An example array for our min-stack.

			$\langle 4,4\rangle$				
		$\langle 5,3\rangle$	$\langle 5,3\rangle$			$\langle 1,2\rangle$	$\langle 1,2\rangle$
	$\langle 6,2\rangle$	$\langle 6,2\rangle$	$\langle 6,2\rangle$	$\langle 3,1\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$
	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$	$\langle \infty,-1\rangle$
i	6	5	4	3	2	1	0

(b) The min stack while scanning array A from right to left.

Figure 4.15. The min-stack for each *current* position i (b) while scanning A (a) from right to left. A tuple (p, v) contains the position p of the value v. For the current position i the stack can be used to answer RMQs of the type $\mathsf{RMQ}_{\mathsf{A}}[i, j]$ with $j \geq i$ by looking at elements from the top until a position k with $k \geq j$ is found.

answer the RMQs using a *min-stack* [Bin+16a; Fis11; GO11]. See also Figure 4.15 for an example. We only need to consider RMQs for suffixes from the same (α, β)-bucket. To this end, we build the min-stack while scanning an interval $[\mathsf{first}, \mathsf{last}]$ (from right to left) of the LCP array. An entry on the min-stack is a tuple $\langle k, \mathsf{LCP}[k]\rangle$. Initially, the tuple $\langle -\infty, -1\rangle$ is pushed on the min-stack. To update the min-stack at position $i \in [\mathsf{first}, \mathsf{last}]$, we look at the top of the min-stack and pop the tuple $\langle k, \mathsf{LCP}[k]\rangle$ if $\mathsf{LCP}[k] \geq \mathsf{LCP}[i]$. We repeat this process until no tuple is popped. Then we push $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack.

Now we want to answer $\mathsf{RMQ}_{\mathsf{LCP}}[i, j]$ with $\mathsf{first} \leq i < j \leq \mathsf{last}$. (It should be noted that at this point we have not pushed $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack or have popped any tuple from the min-stack in the process of adding it to the min-stack.) To this end, we scan the min-stack from top to bottom, until we find two consecutive tuples $\langle k, \mathsf{LCP}[k]\rangle$, $\langle k', \mathsf{LCP}[k']\rangle$ such that $k' > j$. Then, $\mathsf{RMQ}_{\mathsf{LCP}}[i, j] = \mathsf{LCP}[k]$. If we scan the text from left to right, the min-stack works analogously. The only difference is that the initial tuple is $\langle -1, -1\rangle$ and we search for the two consecutive tuples until $k' < j$

The min-stack is cleared whenever we arrive at a new (α, β)-bucket, i. e., we only keep the $\langle n, -1\rangle$-tuple. In the implementation, the min-stack is realized using a single array and a reference to its current top. In addition to the min-stack, we require for each α-bucket the position of where the last suffix has been induced from. This is the position we look for when querying the min-stack.

4.3.3 Special Cases during LCP Induction

There are three special cases ((SC1)–(SC3)) where the LCP value cannot be induced using the min-stack (or RMQs in general).

(SC1) The first case occurs if a suffix is induced next to a $C^{+\triangleright}$-suffix in the suffix array. The inducing can happen to the left or right of the already placed $C^{+\triangleright}$-suffix. The former case is easy, as there cannot be a C^-- or C^+-suffix to the left of a $C^{+\triangleright}$-suffix in the same (α, β)-bucket. Therefore, we only need to check whether the suffixes are in the same α-bucket to compute the LCP value for the $C^{+\triangleright}$-suffix, which is either 0 or 1.

(SC2) The second case (a suffix is induced to the right of a $C^{+\triangleright}$-suffix) is more demanding, as the LCP value must be computed. First, we check whether both the $C^{+\triangleright}$-suffix $T[i..n)$ and the C^+-suffix $T[j..n)$ are in the same (α, β)-bucket. If not, the LCP value is 1 if they occur in the same α-bucket, and 0 otherwise. However, if they occur in the same (α, β)-bucket, we know that $T[i..n)$ has a prefix $\alpha\beta\gamma$, $\gamma \in \Sigma$, such that $\alpha < \beta \geq \gamma$, and that $T[j..n)$ has a prefix $\alpha\beta\delta$, $\delta \in \Sigma$, such that $\alpha < \beta \leq \delta$. Hence, the LCP value is $\max\{k \geq 0 \colon T[i+1..i+k+2) = T[j+1..j+k+2)\} + 1$, i. e., the first appearance of a character not equal to β in either suffix.

(SC3) In the last case, an C^--suffix is induced next to a C^+-suffix, the LCP value can be determined in an analogous way to the second case.

Therefore, we can solve the first case (SC1) without comparing suffixes at all. For the second case (SC2) and third case (SC3), we have to scan the suffixes until we see a new character.

4.4 EXPERIMENTAL EVALUATION

We extended *DivSufSort* to compute the LCP array as described above and denote it by *DivSufSort-LCP*. The source code is available at `www.kurpicz.org/lcp`. As before, we use LiDO.small nodes to run the experiments. The code was compiled using GCC 9.2.0 with flags `-O3` and `-march=native`. For inputs we use prefixes of the texts described in Section 1.4.2 in addition to *Pizza & Chili* corpus, which we described in Section 4.1. We compare our implementation with the following LCP construction algorithms.

SAIS-LCP [Fis11] is an LCP array construction algorithm based on SAIS [Mor08; Non+11] and is similar to the algorithm we described in this section, with the main difference being the underlying induced copying suffix sorting algorithm. As SAIS is a linear running time algorithm, this LCP array construction algorithm has $\mathcal{O}(n)$ running time.

KLAAP [Kas+01] is a linear time LCP array construction algorithm that uses the fact that if an LCP value k exists in the LCP array, then the LCP value $k-1$

also has to exist to save text comparisons. To be more precise, if $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i-1], \mathsf{SA}[i]) = k$ and $k > 0$, then $\mathsf{lcp}(\mathsf{SA}[i-1]+1, \mathsf{SA}[i]+1) = k-1$. However, we do not know if $\mathsf{SA}[i-1]+1$ and $\mathsf{SA}[i]+1$ are consecutive entries in the suffix array. But we know that the corresponding LCP value will be at least $k-1$ (if $\mathsf{SA}[i-1]+1$ and $\mathsf{SA}[i]+1$ are consecutive entries). The inverse suffix array is used to identify the required text positions in constant time. We implemented this algorithm for the following experiments. The KLAAP algorithm is also implemented as part of the SDSL. We denote the implementation contained in the SDSL by *KLAAP (SDSL)*.

Φ-algorithm [Kär+09] is a practical improvement of Kasai et al.'s [Kas+01] KLAAP algorithm that computes the *permuted* LCP (PLCP) array instead, i. e., the LCP array in text order. Therefore, the LCP array has to be computed from the PCLP array. Nevertheless, the different order the LCP array is initially constructed in allows to reduce the number of cache misses (by up to one per entry of the LCP array) and speeds up the algorithm in practice, as we see in the experimental evaluation. Like the KLAAP algorithm, we implemented this algorithm for the following experiments. The *Φ*-algorithm, too, is implemented in the SDSL, we denote this algorithm by *Φ-algorithm (SDSL)*.

SE-Φ-algorithm is a semi-external memory variant of the *Φ*-algorithms, which is part of the SDSL. It should be noted that the SDSL does not necessarily write to external memory. The data is often cached and for smaller inputs (fitting into main memory) kept in main memory.

GO and *GO2* [GO11] are two variants of an algorithms with $\mathcal{O}(n^2)$ running time that are, however, fast in practice. This is due to the authors' observation that (in their experiments) one cache miss takes as long as the comparison of 20 characters. Both algorithms require the *Burrows-Wheeler transform* (BWT) [BW94], which can be computed in linear time given the suffix array and is a permutation of the input.

BWT and *BWT2* are two LCP array construction algorithms that are also based on the BWT. They are part of the SDSL and, as far as the author known, not described in any publication.

Running Time and Memory Requirements

In Tables 4.2 and 4.3 we report the throughput of the LCP array construction algorithms. On all but four inputs (`DNA` (1024 MiB), `Prot` (1024 MiB), english, and proteins) DivSufSort-LCP is the fastest LCP array construction algorithm. However, it should be noted that here it is only slightly faster than the *Φ*-algorithm—the fastest LCP array construction algorithm on all other inputs.

Yet, DivSufSort-LCP is most often the fastest LCP array construction algorithm based on induced copying. It is faster than SAIS-LCP on all inputs but `CommonCrawl`, influenza, and kernel.

The memory requirements of all LCP array construction algorithms are listed in Tables 4.4 and 4.5. There, we see that the algorithms implemented in the SDSL are the most memory efficient ones, requiring only 7 bytes per character of the text. However, this is due to the design of the SDSL, all algorithms behave like semi-external memory (Section 1.3.4) algorithms, but the data is still kept in main memory if the RAM is large enough, which is the case for all tested inputs. If this is the case, the algorithms in the SDSL are the most memory efficient ones. Hence, algorithms based on induced copying require 7 bytes per character of the text, making DivSufSort-LCP the ideal combination of speed and memory requirements, because the Φ-algorithm need 13 bytes per character of the text.

4.5 CONCLUSION AND FUTURE WORK

In this chapter, we tested if *DivSufSort* is, as generally assumed, the fastest main memory suffix sorting algorithm. However, very much to our surprise, there exist inputs on which this is not true and SAIS is faster. We identified that those inputs are (highly) repetitive, i. e., inputs where a some substrings occur a lot. This is due to the different techniques used for lexicographically sorting the *special* suffixes that are needed to induce the lexicographical order of all other suffixes. Here, the recursive approach employed by SAIS resolves the repetitive substrings better than the doubling approach used by *DivSufSort*. Still, it remains true—and is now experimentally proven—that *DivSufSort* is in general the fastest main memory suffix sorting algorithm.

We then looked at LCP array construction algorithms, as the LCP array often accompanies the suffix array. Here, the Φ-algorithm is the fastest on most inputs, but it also requires the most memory. LCP array construction based on induced copying, however, has a good trade-off between running time and memory requirements. As DivSufSort-LCP is only slightly slower than the Φ-algorithm but requires only 69 % of its memory.

Future Work. While in-place suffix sorting in main memory has been solved by Goto [Got17] and Li et al. [Li+18], there are still open questions regarding main memory suffix sorting in practice: (i) Is there a linear time suffix sorting algorithm faster than *DivSufSort* on *all* inputs? (ii) Are the theoretical optimal suffix sorting algorithms by Goto [Got17] and Li et al. [Li+18] practical? (iii) Baier [Bai16] has provided a new technique for main memory suffix sorting, are there further ways to compute the suffix array in linear time, or can we just improve the existing ones? In conclusion, the question is *can we be faster*? The same holds for the LCP array construction.

Table 4.2. Throughput in MiB per seconds of LCP array construction algorithms. All algorithms requiring the suffix array as input compute it using DivSufSort. The suffix array's construction time is included. If the BWT is required for the construction, its construction time is included, too, and the BWT is computed using the suffix array.

	text	DivSufSort-LCP	SAIS-LCP	KLAAP	Φ-algorithm	KLAAP (SDSL)	Φ-algorithm (SDSL)	SE-Φ-algorithm	GO	GO2	BWT	BWT2
256 MiB	CommonCrawl	1.54	4.00	4.95	**5.29**	3.46			3.31		1.68	1.61
	DNA	4.29	3.22	4.24	**4.60**	2.99	3.21	2.89	3.00	3.20	2.88	2.66
	Prot	3.91	3.05	4.03	**4.27**	2.75	3.34	3.33	2.78	2.95	1.51	1.64
	Wiki	4.91	3.48	4.63	**5.00**	3.10	3.58	3.34	3.29	3.34		1.99
512 MiB	CommonCrawl	1.38	3.15	4.32	**5.01**	3.20			3.07	2.95	1.58	1.52
	DNA	4.06	2.54	3.68	**4.36**	2.64	2.93	2.60	2.88	3.04	2.82	2.60
	Prot	3.40	2.64	3.55	**3.85**	2.55	2.70	2.76	2.67	2.58	1.58	1.57
	Wiki	4.00	3.27	4.29	**4.70**	2.88	3.40	2.93	3.08	2.91	2.06	2.00
1024 MiB	CommonCrawl	1.21	3.43	4.29	**4.40**	2.71			2.94	2.73	1.30	1.37
	DNA	**3.86**	2.39	3.46	3.83	2.34	2.96	2.34	2.98	2.99	2.60	2.53
	Prot	**3.61**	2.39	2.88	3.60	2.28	2.50	2.51	2.37	2.41	1.61	1.51
	Wiki	4.36	2.50	3.88	**4.42**	2.43	2.98	2.88	3.01	2.97	1.98	1.87

Table 4.3. Throughput in MiB per seconds of LCP array construction algorithms (continuation). All algorithms requiring the suffix array as input compute it using DivSufSort. The suffix array's construction time is included. If the BWT is required for the construction, its construction time is included, too, and the BWT is computed using the suffix array.

text	DivSufSort-LCP	SAIS-LCP	KLAAP	Φ-algorithm	KLAAP (SDSL)	Φ-algorithm (SDSL)	SE-Φ-algorithm	GO	GO2	BWT	BWT2
dblp.xml	6.13	3.99	5.98	**6.31**	3.98	4.56	3.82	4.01	3.79	2.55	2.41
dna	3.71	3.14	3.94	**4.25**	2.64	3.21	2.65	3.12	3.09	2.72	2.46
english	**4.43**	3.12	4.20	4.12	2.94	3.37	3.02	2.63	2.73	1.58	1.68
proteins	**4.39**	3.08	4.07	4.33	2.83	3.41	3.00	2.49	2.57	1.57	1.64
sources	6.47	4.94	6.15	**6.53**	4.18	4.56	4.08	3.87	4.00	1.93	1.97
Escherichia_Coli	5.0	4.4	5.1	5.5	3.5	4.1	3.7	3.6	2.5	**1.8**	2.0
cere	4.8	4.3	5.1	5.6	3.6	4.0	3.6	3.6	2.2	**1.5**	1.8
coreutils	6.2	5.8	6.0	6.6	4.1	4.6	4.6	4.0	2.6	**0.8**	1.1
einstein.de.txt	5.9	5.2	6.3	6.7	4.3	4.7	5.1	3.9	2.3	**0.6**	1.2
einstein.en.txt	5.0	4.6	5.4	5.8	3.9	4.0	4.1	3.6	1.9	**0.5**	1.1
influenza	4.6	5.2	5.7	6.2	3.9	4.3	3.6	3.9	2.6	**2.1**	2.1
kernel	5.3	5.5	5.9	6.5	3.9	4.2	4.7	4.0	2.3	**0.3**	1.1
para	4.7	4.2	5.0	5.4	3.2	3.8	3.4	3.4	2.1	**1.3**	1.8
world_leaders	10.8	8.4	9.7	11.4	5.8	7.3	7.4	6.1	3.7	**2.0**	2.4

Table 4.4. Memory peak in byte per character of the input of the LCP array construction algorithms. All algorithms requiring the suffix array as input compute it using DivSufSort, which requires 4 bytes per character of the input.

	text	DivSufSort-LCP	SAIS-LCP	KLAAP	Φ-algorithm	KLAAP (SDSL)	Φ-algorithm (SDSL)	SE-Φ-algorithm	GO	GO2	BWT	BWT2
256 MiB	CommonCrawl	9.01	9.01	13.01	13.01	10.01			7.02		7.04	**7.01**
	DNA	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.01	7.05	7.09	7.09
	Prot	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.03	7.05	7.05	7.01
	Wiki	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.03	7.02		7.01
512 MiB	CommonCrawl	9.00	9.00	13.00	13.00	10.01			7.01	7.04	7.06	**7.01**
	DNA	9.00	9.00	13.00	13.00	10.01	**7.00**	7.00	7.01	7.04	7.01	7.01
	Prot	9.00	9.00	13.00	13.00	10.01	**7.00**	7.00	7.01	7.04	7.01	7.01
	Wiki	9.00	9.00	13.00	13.00	10.01	**7.00**	7.00	7.01	7.01	7.05	7.01
1024 MiB	CommonCrawl	9.00	9.00	13.00	13.00	10.00			7.01	7.00	7.00	**7.00**
	DNA	9.00	9.00	13.00	13.00	10.00	**7.00**	7.00	7.00	7.00	7.00	7.00
	Prot	9.00	9.00	13.00	13.00	10.00	**7.00**	7.00	7.01	7.00	7.00	7.00
	Wiki	9.00	9.00	13.00	13.00	10.00	**7.00**	7.00	7.05	7.01	7.00	7.00

Table 4.5. Memory peak in MiB per character of the input of the LCP array construction algorithms (continuation). All algorithms requiring the suffix array as input compute it using DivSufSort, which requires 4 bytes per character of the input.

text	DivSufSort-LCP	SAIS-LCP	KLAAP	Φ-algorithm	KLAAP (SDSL)	Φ-algorithm (SDSL)	SE-Φ-algorithm	GO	GO2	BWT	BWT2
dblp.xml	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.01	7.05	7.05	7.01
dna	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.02	7.03	7.12	7.09
english	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.02	7.05	7.06	7.01
proteins	9.01	9.01	13.01	13.01	10.01	**7.01**	7.01	7.02	7.05	7.05	7.01
sources	9.01	9.01	13.01	13.01	10.01	**7.01**	7.18	7.18	7.19	7.19	7.16
Escherichia_Coli	9.02	9.02	13.02	13.02	10.03	**7.03**	7.22	7.03	7.21	7.39	7.39
cere	9.01	9.01	13.01	13.01	10.01	7.01	**7.01**	7.01	7.80	7.15	7.10
coreutils	9.01	9.01	13.01	13.01	10.02	**7.02**	7.18	7.02	7.86	7.21	7.15
einstein.de.txt	9.02	9.02	13.02	13.02	10.03	**7.03**	7.24	7.03	8.90	7.03	7.33
einstein.en.txt	9.01	9.01	13.01	13.01	10.01	7.01	**7.01**	7.01	9.13	7.01	7.01
influenza	9.01	9.01	13.01	13.01	10.02	**7.02**	7.20	7.02	7.20	7.03	7.23
kernel	9.01	9.01	13.01	13.01	10.01	**7.01**	7.17	7.01	8.79	7.21	7.15
para	9.01	9.01	13.01	13.01	10.01	7.01	**7.01**	7.01	7.70	7.05	7.10
world_leaders	9.04	9.04	13.04	13.04	10.07	7.07	**7.05**	7.07	7.65	7.14	7.09

CHAPTER 5

DISTRIBUTED SUFFIX ARRAY CONSTRUCTION

In this chapter, we focus on *distributed memory* suffix array construction, which is a field where less work has been conducted than for main memory. Previously, in Section 4.1, we described different approaches to compute the suffix array, namely: *prefix doubling*, *induced copying*, and *recursion*. Now, we adapt these approaches to distributed memory. First, in Sections 5.2 and 5.3, we present algorithms based on prefix doubling and recursion. Then, in Section 5.4, we also present the first distributed induced copying suffix array construction algorithm, which is a distributed variant of *DivSufSort* (which we thoroughly described in Section 4.2). Finally, in Section 5.6, we present the extensive experimental results of our distributed suffix array construction.

Now, before we delve into suffix sorting in distributed memory, we briefly mention suffix array construction algorithms in other models of computation, which we described in Section 1.3. In *shared memory*, there exists a parallel version of the recursive DC3 algorithm [KS03], a parallel implementation of DivSufSort [Lab+17], SACA-K [Lao+18a], and SAIS [Lao+18b]. Similar to shared memory, the *GPU* (graphics processing unit) can be used to execute algorithms in parallel. However, implementing a scalable GPU algorithm has more restrictions than shared memory algorithms, which are not relevant for this dissertation as we only list the algorithms for completeness. The following suffix array construction algorithms are designed to be run on the GPU: [Bah+19b], [DK13], [Osi12], [SM09], and [Wan+16]. However, all of these GPU-algorithms except the one by Bahne et al. [Bah+19b] have either no publicly available implementation or provide an implementation that does not compute a correct suffix array for all inputs. In *external memory*, induced copying algorithms by Bingmann et al. [Bin+16a], Kärkkäinen et al. [Kär+17], and Nong et al. [Non+15], which are based on the linear time recursive and induced copying hybrid SAIS [Mor08; Non+11] and minor parts of DivSufSort [Mor06] (see also Section 4.2) are the fastest algorithms. There also are external memory suffix array construction algorithms based on prefix doubling [Dem+08a], recursion [KS03], and an external memory algorithm that computes partial suffix arrays and merges them [KK17] that also has a parallel (still external memory) version by Kärkkäinen et al. [Kär+15].

Overview of Distributed Suffix Array Construction Algorithms. Unlike for main memory, the list of distributed suffix array construction algorithms is relatively short. Kärkkäinen and Sanders [KS03] show that their linear time suffix array construction algorithm can be distributed, which only seems natural, as it is mostly based on sorting and merging. This results in BSP costs of $\mathcal{O}((1 + G/\lg(n/p))n \lg n/p + \lg^2 pL)$, where p is the number of processing elements. There also exists a practical evaluation of this algorithm [KS07] and an independently developed publicly available implementation [Bin12], which only exists as code but has not been published anywhere. We use the latter in our experiments in Section 5.6.

Next, Navarro et al. [Nav+97] present a distributed algorithm that is very similar to the *domain decomposition* approach we use for wavelet tree construction in Section 3.3.3. Initially, each processing element computes the suffix array for its local slice of the text, before those slices are merged. Obviously, the merging of suffix arrays is more complex than merging the partial wavelet trees.

Futamura et al.'s [Fut+01] distributed suffix array construction algorithm works similar to Navarro et al.'s [Nav+97]. They use a more complex scheme to distribute the suffixes which makes it more complex to efficiently compute the local suffix arrays. However, the benefit of this approach is that the merging becomes easier.

The algorithms by Futamura et al. [Fut+01] and Kulla and Sanders [KS07] were compared in a small survey by Metwally et al. [Met+16]. There the authors reimplemented the algorithms. Yet, their implementations do not work well for larger inputs as they have huge memory requirements and are very slow (even on small inputs). Futamura et al.'s [Fut+01] algorithm was also reimplemented by Abdelhadi et al. [Abd+14]. This implementation also suffers from huge memory requirements and a very slow running time, which we comment on in our experiments in Section 5.6.

Finally, there is a distributed suffix array construction algorithm based on prefix doubling by Flick and Aluru [FA15]. We highlight this algorithm, as it has a publicly available implementation that works well even for larger inputs and is our main competitor in our evaluation in Section 5.6. We explain the concept of prefix doubling in general and the specifics of distributed prefix doubling in detail in Section 5.2. There, we also describe Flick and Aluru's [FA15] distributed prefix doubling suffix array construction algorithm in more detail.

5.1 PRELIMINARIES

The processing elements that are used to run distributed algorithms have to communicate. To this end, we make use of two different frameworks: the message passing interface (*MPI*) and a distributed batch processing framework (*Thrill*), which we describe in more detail in Sections 5.1.1 and 5.1.2. The main difference between these two approaches is that MPI allows us to directly send and recieve any data from any processing element to another processing element, whereas Thrill stores all data distributed over all processing elements without the user knowing of the specific location and allows access only via predefined functions.

5.1.1 MPI: The Message Passing Interface

The message passing interface (MPI, see `www.mpi-forum.org`) describes a standard for message passing in parallel algorithms. MPI can be used to implement parallel shared memory algorithms but is mostly used to implement parallel distributed memory algorithms. There exist multiple implementations of the standard, e. g., OpenMPI and MPICH. In the following, we briefly introduce MPI, which helps us understand the following algorithms but is by no means a complete description of MPI.

Point-to-Point Communication. The most basic operation provided by MPI is *sending* and *receiving* data. Most of the functionality provided by MPI is sending and receiving data in more or less complex ways. In this context, point-to-point communication means that one processing element i sends data and another processing element j receives the data send by processing element i.

Collective Communication. Collective communication provides more advanced ways to communicate data. For example, we can *broadcast* data from one processing element to all others. Obviously, this can be realized by sending and receiving data point-to-point, however, using collective communication can be faster due to more sophisticated algorithms provided by MPI. More complex examples are *scan* (prefix sum), *scatter* (one processing element sends data to all other processing elements, which can be different for each receiving processing element), *gather* (one processing element receives data from all other processing elements), or *reduce* (like gather, but an operation is applied to all data, e. g., summing up all data or choosing the maximum).

One-Sided Communication. One-sided communication is also known as *remote direct memory access* (RDMA). In MPI, this allows us to directly access RAM on other processing elements without having them send it. Still, synchronizations are necessary and lots of random RDMA is slow in practice for that reason.

In general, MPI provides an interface that we use to write our distributed algorithms. Most importantly, we can always access all data and send arbitrary data from one processing element to another. This is the main difference between MPI and Thrill, which is the framework that we describe below.

5.1.2 Thrill: A Distributed Big Data Batch Processing Framework

Thrill [Bin+16b] works with *distributed immutable arrays* (DIAs) storing data. Data in DIAs cannot be accessed directly; instead there is a rich set of DIA operations which can be used to transform DIAs. In the following, we give a list and describe a subset of these operations, which are all the operations necessary to describe our distributed suffix array construction algorithms that we implemented using Thrill. For a complete overview of Thrill's features, we refer to the dissertation of Bingmann [Bin18, p. 233–268].

Filter(f) takes a DIA$\langle A\rangle$ X and a function $f\colon A \to \texttt{bool}$, and returns the DIA$\langle A\rangle$ containing $[\, x \in X \mid f(x)\,]$ within which the order of items is maintained.

Map(f) applies the function $f\colon A \to B$ to each item in the input DIA$\langle A\rangle$ X, and returns a DIA$\langle B\rangle$ Y with $Y[i] = f(X[i])$ for all $i = 0, \ldots, |X| - 1$.

Window$_k$(w) and FlatWindow$_k$(w') take an input DIA$\langle A\rangle$ X and a window function $w\colon \mathbb{N}_0 \times A^k \to B$. The operation scans over X with a window of size k and applies w once to each set of k consecutive items from X and their index in X. The final $k-1$ indexes with less than k consecutive items are delivered to w as partial windows padded with sentinel values. The result of all invocations of w is returned as a DIA$\langle B\rangle$ containing $|X|$ items in corresponding order. FlatWindow is a variant of Window which takes a input DIA$\langle A\rangle$ X and a window function $w'\colon \mathbb{N}_0 \times A^k \to \text{list}(B)$, that can *emit* zero or more items that are concatenated in the resulting DIA$\langle B\rangle$.

PrefixSum(s) takes an input DIA$\langle A\rangle$ X and an associative operation $s\colon A \times A \to A$ (by default $s = +$), and returns a DIA$\langle A\rangle$ Y such that $Y[0] = X[0]$ and $Y[i] = s(Y[i-1], X[i])$ for all $i = 1, \ldots, |X| - 1$,

Sort(c) sorts an input DIA$\langle A\rangle$ X with respect to a less-comparison function $c\colon A \times A \to \texttt{bool}$,

Merge($X_1, \ldots, X_n, c$) merges a set of sorted DIA$\langle A\rangle$s $X_1, \ldots, X_n$ and a less-comparison function $c\colon A \times A \to \texttt{bool}$. The results is DIA$\langle A\rangle$ Y that contains all tuples of $X_1, \ldots, X_n$ and is sorted with respect to c,

Zip($X_1, \ldots, X_n, f$) takes a set of DIAs $X_1, \ldots, X_n$ of type $A_1, \ldots, A_n$ of equal size ($|X_1| = \cdots = |X_n|$) and a function $f\colon A_1 \times \cdots \times A_n \to B$, and returns DIA$\langle B\rangle$ Y with $Y[i] = f(X_1[i], \ldots, X_n[i])$ for all $i = 0, \ldots, |X_1| - 1$,

ZipWithIndex(f) takes an input DIA$\langle A\rangle$ X and a function $f : A \times \mathbb{N}_0 \to B$. Then, ZipWithIndex returns DIA$\langle B\rangle$ Y with $Y[i] = f(X[i], i)$ for all $i = 0, \ldots, |X| - 1$,

ZipWindow$_{[k_1,\ldots,k_n]}$($[X_1, \ldots, X_n], z$) combines a Zip operation and a Window operation: Given a list DIA$\langle A_1\rangle X_1$, ..., DIA$\langle A_n\rangle X_n$, and a function $z : \mathbb{N}_0 \times A_1^{k_1} \times \ldots \times A_n^{k_n} \to B$, ZipWindow returns DIA$\langle B\rangle$ Y with $Y[i] = z(i, X_1[i, \ldots, i + k_1], \ldots, X_n[i, \ldots, i + k_n])$ for all $i = 0, \ldots, |X| - 1$,

Max(c) takes an input DIA$\langle A\rangle$ X, Max returns the maximum item $m = \max_c X$ with respect to a less-comparison function $c\colon A \times A \to \texttt{bool}$, and

Size() returns the number of items in X, i. e., $|X|$, give an input DIA$\langle A\rangle$ X.

Method Chaining. Thrill applies chains of functions (*method chaining*) to a DIA, e. g., if we have a DIA$\langle \mathbb{N}_0\rangle$ $N = \{0, 1, 2, \ldots, 9\}$ and want to compute the prefix sum of all odd elements, then we write N.Filter($a \mapsto (a \mod 2) = 1$).PrefixSum(). We make heavy use of this notation in our pseudo code listings.

5.2 DISTRIBUTED PREFIX DOUBLING

In this section, we present *prefix doubling* algorithms for suffix sorting. Let us start by reviewing their general idea. For better exposition of these ideas, we define the *h-order* $\leq_h$ on strings as their lexicographic order limited to depth h: $a|_h$ and $b|_h$ are the strings a or b truncated to h characters. Then $a \leq_h b$ if and only if $a|_h \leq b|_h$. Other comparison operators like $a =_h b$ and $a <_h b$ are defined accordingly. As previously, let T be a text of length n over an alphabet of size σ. For $h < n$, the h-order of suffixes of T may not be unique, e. g., with respect to $\leq_2$, all suffixes starting with the same two characters are considered equal; their order is not fixed.

A set of suffixes equal under $=_h$ is called an *h-group* and they all start with the same h characters. A *rank* with respect to $\leq_h$ of the suffixes in an h-group is any number greater than the total size of all h-groups containing lexicographically smaller suffixes and smaller than any rank of an h-groups containing lexicographically larger suffixes. A rank with respect to $\leq_h$ is also called a *(lexicographic) h-name* or *h-rank*. We denote the *partital* suffix array that is sorted using the h-ranks of the suffixes SA^h. Note that SA^h is only unique if all h-groups consist of a single suffix.

Since h doubles in each round, $h \geq n$ after $\lceil \lg n \rceil$ rounds and thus prefix doubling algorithms have worst case running time $\mathcal{O}(n \lg n)$. To be more precise, the algorithm terminates when SA^h has no more unsorted h-groups and becomes the suffix array. This already happens after $\lceil \lg(\text{maxlcp}(T)) \rceil$ iterations, yielding $\mathcal{O}(n \log(\text{maxlcp}(T)))$ running time, where $\text{maxlcp}(T)$ denotes the maximum value in T's LCP array.

The essential goal of a prefix doubling algorithm is to give each suffix a lexicographic 2^k-name in iteration k using information from iteration $k-1$. Manber and Myers [MM93] observed that one can compute a 2^k-rank for the prefix $T[i..i+2^k)$ of suffix $T[i..n)$ using already computed 2^{k-1}-ranks of the prefixes $T[i..i+2^{k-1})$ and $T[i+2^{k-1}..i+2^k)$. Since this idea is of great importance for this section, we give a detailed explanation of it in the following.

The main idea to bringing this to distributed memory is to store and sort tuples (i, r_i) containing the rank r_i of the i-th suffix. During each iteration k, this rank should be the rank of the truncated suffix $T[i..n)|_{2^k}$, i. e., $T[i..i+2^k)$.

PD1 First, we have to compute the rank of each length-1 prefix, which is the rank of the character and given by the alphabet. To this end, we associate each index with a rank. During the first iteration, this rank is the rank of the character $T[i]$ among all characters occurring in T. Thus we creating rank-tuples $(i, T[i])$ for all $i \in [0, n)$. Also, we start counting the number of iterations; we let be $k = 0$.

Having these rank-tuples, we first check if we already have computed the suffix array. We conduct this check during each iteration.

PD2 Whenever we compute new ranks for the suffixes, we have to check if all ranks are unique. If they are, we have all information we need to compute the suffix array SA. We can compute the suffix array by sorting the tuples by the second component. Then, the first component corresponds to the SA. Otherwise (if not all ranks are unique), we continue.

Now, we want to compute the new ranks using the old ones (unless we have already sorted all suffixes).

PD3 Construct rank-triples (i, r, r'), where (i, r) is the previously considered rank-tuple and r' is the rank of the tuple with index $i + 2^k$ (or 0 if $i + 2^k \geq n$). Here, we change from the 2^k-order to the 2^{k+1}-order by combining the ranks.

Computing triples $(i, r_i, r_{i+2^{k-1}})$ in iteration k is not a totally new idea. This approach to generate new names has already proven to work well in external memory [Dem+08a].

PD4 Next, we sort the rank-triples by (r, r') and compute new rank-tuples (i, r_{new}). Using two such triples one can take the step from 2^{k-1}-ranks to 2^k-ranks: consider $(i, r_i, r_{i+2^{k-1}})$ and $(j, r_j, r_{j+2^{k-1}})$ with $r_i = r_j$. This means that suffixes i and j start with the same 2^{k-1} characters, $T[i..n) =_{(2^{k-1})} T[j..n)$. By comparing $r_{i+2^{k-1}}$ and $r_{j+2^{k-1}}$, we can determine the lexicographic order of the next 2^{k-1} characters, and hence compute new ranks. Then, increase k by one and continue with Step PD2.

Most notably, we do not recompute all ranks during each iteration by comparing the prefixes, but use the ranks from the previous iteration, i. e., we use the 2^k-groups to compute the 2^{k+1}-groups without further text access.

5.2.1 Prefix Doubling in Thrill

Since the canonical prefix doubling algorithm that we describe above only requires scanning and sorting arrays, we can easily implement it using Thrill. In this section, we describe two different approaches (one is based in the inverse suffix array and the other one is based on sorting) for obtaining the triples that are necessary to compute the new ranks given the limitations of Thrill. Then, we show how we can significantly improve the sorting variant by discarding already sorted suffixes. In addition to the doubling algorithms, we also implemented quadrupling algorithms that have the same general idea but quadruple the length of the considered suffixes by employing four ranks instead of two to generate the new ranks.

Algorithm 5.1 describes the basic structure of the prefix doubling algorithms in Thrill presented in this section. The whole algorithm requires one DIA N storing pairs and one DIA S storing triples. For the first iteration, S contains the triples $(i, T[i], T[i+1])$ for all $i \in [0, n)$ (line 1). These triples contain a text position and the *rank-tuple* for that position, i. e, the two ranks that are required to compute the new rank for the suffix starting at the text position. For bootstrapping the first iteration $k = 1$, we can simply use the characters as 1-ranks.

In our actual implementation, we accelerated the first iteration by computing the effective alphabet first. Quite often, the input string does not use the whole alphabet range, and we can apply the effective alphabet before computing the suffix array, because the input string and the mapped string have the same suffix array, as the lexicographic order of suffixes does not change. In our implementation, we then *pack* as many mapped characters as possible into an integer index.

Algorithm 5.1. Generic Prefix Doubling algorithm.

Input : Text $T \in \mathrm{DIA}\langle\Sigma\rangle$.
Output : Suffix array SA.

```
1 S ≔ T.Window_2((i, [t_0, t_1]) ↦ (i, t_0, t_1))   // Initial triples (combining PD1–PD3).
2 for k ≔ 1 to ⌈log_2 |T|⌉ − 1 do
3 |   S ≔ S.Sort((i, r_0, r_1) by (r_0, r_1))          // Sort triples by ranks.
4 |   N ≔ S.FlatWindow_2((i, [a, b]) ↦ CName(i, a, b))   // Map to ranks 0 or i.
5 |   if N.Filter((i, r) ↦ (r = 0)).Size() = 1 then   // If all ranks are unique, then
6 |   |   return SA ≔ N.Map((i, r) ↦ i)               // return ranks as suffix array,
7 |   N ≔ N.PrefixSum((i, r), (i', r') ↦ (i', max(r, r')))   // else calculate new ranks
8 |   S ≔ Generate new rank-tuples using N            // and run next iteration.
```

Algorithm 5.2. CName—Identifications of h-groups.

Input : $j \in \mathbb{N}_0$, $(i, r_0, r_1), (i', r_0', r_1') \in N$.
Output : Tuple marking h-groups.

```
1 if j = 0 then
2 |   emit (i, 0)                                   // First DIA item has no offset.
3 emit { (i', j)   if (r_0, r_1) ≠ (r_0', r_1'),   // Add sentinel if rank pairs alter.
       { (i', 0)   otherwise.                      // T[i, n) and T[i', n) get the same new name.
```

For subsequent iterations, we continue on line 3 and sort S with respect to the rank-tuple, which brings *equal* 2^{k-1}-ranks together. These entries with equal 2^{k-1}-rank need to be extended to prefix depth 2^k. The new 2^k-ranks are calculated using a $\mathrm{FlatWindow}_2$ on S (line 4) and the function CName(), which takes the current position i in S and the items $S[i]$ and $S[i+1]$ as input and emits a tuple consisting of a text position and a new rank (Algorithm 5.2). We know that the suffixes are sorted with respect to their rank-tuples. Therefore, we can scan S and mark every position where the rank-tuples differs from its predecessor. CName() marks these non-unique rank-tuple by giving them the rank 0. All unique rank-tuples get a rank equal to their current position in S. If there is only one suffix with rank 0, then we know that all ranks differ and that we have finished the computation, see line 5. Otherwise, we can use the DIA operation PrefixSum() with a *max* operator to set the rank of each tuple to the largest preceding rank (line 7). The sequence of ranks is initialized by emitting an arbitrary first rank (zero in Algorithm 5.2) as first item in the DIA. With this extra item, the rank array N always contains n items. Now each suffix has a refined rank.

The next step (line 8) is to identify the ranks of the suffixes required for the next doubling step. During the k-th doubling step, we fill S with one triple for each index $i \in [0, n)$ that contains the current name of the suffix at position i and the current name of the suffix at position $i + 2^{k-1}$. Next, we discuss approaches for doing so.

Algorithm 5.3. Generating Names using the ISA in Thrill.

Input : $k \in \mathbb{N}_0$.
Output : S contains the triples that are used to compute new ranks.

1 $N \coloneqq N.\text{Sort}((i, r) \text{ by } i)$ // *Compute* ISA^{2^k}.

2 $S \coloneqq N.\text{Window}_{2^k+1}((j, [\,(i, r), \ldots, (i', r')\,]) \mapsto \begin{cases} (i, r, r') & \text{if } j + 2^k < |T|\,, \\ (i, r, 0) & \text{otherwise.} \end{cases})$

Generating Ranks using the Inverse Suffix Array

We can obtain the h-names of the required suffixes using the inverse h-suffix array. This approach is based on the qsufsort algorithm [LS07] and was pioneered in a distributed setting using MPI by Flick and Aluru [FA15].

Algorithm 5.3 shows in line 1 how we can sort pairs in N based on their position in the text, such that we get the inverse 2^k-suffix array ISA^{2^k} in iteration k. This inverse 2^k-suffix array contains the current 2^k-name of each suffix. For each position i, we need the name of the $(i + 2^k)$-th suffix. To get this name, we can simply scan over the DIA N with a Window() operation of width $2^k + 1$, i. e., the same as shifting the inverse 2^k-suffix array by 2^k positions and appending 0s until its length is n.

While our experiments show that this approach is faster than prefix doubling using sorting (described in the next subsection), it is obvious that it only works as long as the Window() size $2^k + 1$ fits into the RAM of each worker. The second solution using sorting does not suffer from this limitation and can be used as a fallback method.

Generating Ranks using Sorting

Next, we adapt an external memory prefix doubling algorithm by Crauser and Ferragina [CF02] to Thrill. The idea is to compute the new rank pairs by sorting the old ranks with respect to the starting position of the suffix, as shown in Algorithm 5.4. During each iteration, we know for each suffix the suffix index whose current rank is required to compute the next refined rank. Hence, we can sort the tuples containing the starting positions of the suffixes and their current rank in such a way that if there is another rank required for a rank pair, then it is the rank of the succeeding tuple (line 1). To do so, we use the following comparison operator: $<^k_{\text{op}}\colon (\mathbb{N}_0, \mathbb{N}_0) \times (\mathbb{N}_0, \mathbb{N}_0) \to \texttt{bool}$ in Algorithm 5.4:

$$(i, r) <^k_{\text{op}} (i', r') = \begin{cases} i \textbf{ div } 2^k < i' \textbf{ div } 2^k & \text{if } i \equiv i' \pmod{2^k}\,, \\ i \textbf{ mod } 2^k < i' \textbf{ mod } 2^k & \text{otherwise.} \end{cases} \tag{5.1}$$

This relation orders pairs (i, n) first by the k least significant bits and then by the $w - k$ most significant bits of i, where w is the number of bits used to store i, i. e., we group all positions that are 2^k positions apart (modulo) and sort those in text order (division).

Algorithm 5.4. Generating Ranks using Sorting in Thrill.

Input : $k \in \mathbb{N}_0$.
Output : S contains the triples that are used to compute new ranks.
1 $N := N.\text{Sort}(<_{\text{op}}^{k})$ *// Sort such that ranks of i and $i + 2^k$ are consecutive.*
2 $S := N.\text{Window}_2((j, [\,(i, n_0, n_1), (i', n_0', n_1')\,]) \mapsto \begin{cases} (i, n_0, n_0') & \text{if } i + 2^k = i', \\ (i, n_0, 0) & \text{otherwise.} \end{cases}\Big)$

For example, $<_{\text{op}}^{2}$ reorders $[0\,..\,8)$ to $[\,0, 4, 1, 5, 2, 6, 3, 7\,]$. The modulo operations and divisions with powers of two are fast in practice as they can be realized using bit masks. After sorting using the $<_{\text{op}}^{k}$-comparator, we need to ensure that two consecutive ranks are the ones required to compute the new rank, since the required rank may not exist due to the length of the text, i. e., during the k-th iteration each suffix beginning at a text position greater than $n - 2^k$. In this case, we use the sentinel rank 0, which compares smaller than any valid rank (line 2). We return one triple for each position, consisting of a text position, the current rank of the suffix beginning at that position and the rank of the suffix 2^k positions to the right (if it exists and 0 otherwise).

Example 5.5 shows the suffix array construction for $T =$ `bdacbdacb` using the approach described above. Comments of the form $x.y$ translate to "line y in Algorithm x is responsible for the change." We give a data-flow graph of this in Figure 5.1.

Example 5.5. Example of prefix doubling using sorting in Thrill.

$T = [\mathtt{b}, \mathtt{d}, \mathtt{a}, \mathtt{c}, \mathtt{b}, \mathtt{d}, \mathtt{a}, \mathtt{c}, \mathtt{b}]$
$S = [(0, \mathtt{b}, \mathtt{d}), (1, \mathtt{d}, \mathtt{a}), (2, \mathtt{a}, \mathtt{c}), (3, \mathtt{c}, \mathtt{b}), (4, \mathtt{b}, \mathtt{d}), (5, \mathtt{d}, \mathtt{a}), (6, \mathtt{a}, \mathtt{c}), (7, \mathtt{c}, \mathtt{b}), (8, \mathtt{b}, \$)]$ *// 5.1.1*
k = 1 *// 5.1.2*
$S = [(2, \mathtt{a}, \mathtt{c}), (6, \mathtt{a}, \mathtt{c}), (8, \mathtt{b}, \$), (0, \mathtt{b}, \mathtt{d}), (4, \mathtt{b}, \mathtt{d}), (3, \mathtt{c}, \mathtt{b}), (7, \mathtt{c}, \mathtt{b}), (1, \mathtt{d}, \mathtt{a}), (5, \mathtt{d}, \mathtt{a})]$ *// 5.1.3*
$N = [(2, 0), (6, 0), (8, 2), (0, 3), (4, 0), (3, 5), (7, 0), (1, 7), (5, 0)]$ *// 5.1.4*
4 items with rank 0 *// 5.1.5*
$N = [(2, 0), (6, 0), (8, 2), (0, 3), (4, 3), (3, 5), (7, 5), (1, 7), (5, 7)]$ *// 5.1.7*
$N = [(0, 3), (2, 0), (4, 3), (6, 0), (8, 2), (1, 7), (3, 5), (5, 7), (7, 5)]$ *// 5.4.1*
$S = [(0, 3, 0), (2, 0, 3), (4, 3, 0), (6, 0, 2), (8, 2, 0), (1, 7, 5), (3, 5, 7), (5, 7, 5), (7, 5, 0)]$ *// 5.4.2*
k = 2 *// 5.1.2*
$S = [(6, 0, 2), (2, 0, 3), (8, 2, 0), (0, 3, 0), (4, 3, 0), (7, 5, 0), (3, 5, 7), (1, 7, 5), (5, 7, 5)]$ *// 5.1.3*
$N = [(6, 0), (2, 1), (8, 2), (0, 3), (4, 0), (7, 5), (3, 6), (1, 7), (5, 0)]$ *// 5.1.4*
2 items with rank 0 *// 5.1.5*
$N = [(6, 0), (2, 1), (8, 2), (0, 3), (4, 3), (7, 5), (3, 6), (1, 7), (5, 7)]$ *// 5.1.7*
$N = [(0, 3), (4, 3), (8, 2), (1, 7), (5, 7), (2, 1), (6, 0), (3, 6), (7, 5)]$ *// 5.4.1*
$S = [(0, 3, 3), (4, 3, 2), (8, 2, 0), (1, 7, 7), (5, 7, 0), (2, 1, 0), (6, 0, 0), (3, 6, 5), (7, 5, 0)]$ *// 5.4.2*
k = 3 *// 5.1.2*
$S = [(6, 0, 0), (2, 1, 0), (8, 2, 0), (4, 3, 2), (0, 3, 3), (7, 5, 0), (3, 6, 5), (5, 7, 0), (1, 7, 7)]$ *// 5.1.3*
$N = [(6, 0), (2, 1), (8, 2), (4, 3), (0, 4), (7, 5), (3, 6), (5, 7), (1, 8)]$ *// 5.1.4*
1 item with rank 0 *// 5.1.5*
Result: $[6, 2, 8, 4, 0, 7, 3, 5, 1]$ *// 5.1.6*

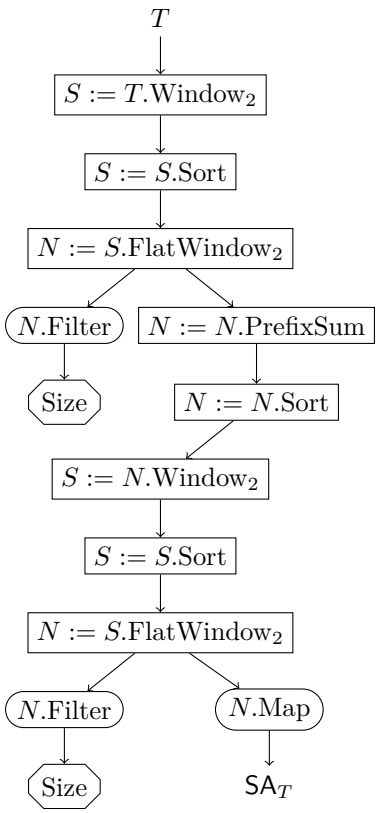


Figure 5.1. DIA data-flow graph of two iterations of prefix doubling with sorting, exported by Thrill.

Distributed External Prefix Doubling with Discarding

Both prefix doubling variants presented in the previous two sections have large I/O costs from repeatedly re-ranking suffixes whose final rank is already known. These are included in each distributed sorting operation and cause needless overhead. Crauser and Ferragina [CF02] and Dementiev et al. [Dem+08a] presented a method called *discarding* to alleviate this by omitting all suffixes no longer needed from sorting operations. To this end, we classify suffixes into three categories:

1. Suffixes that do not yet have a unique rank are called *not unique*, which is also the initial state,
2. suffixes that have a unique rank, but are required to compute another rank pair for a suffix that does not yet have a unique rank, are called *unique*, and finally
3. unique suffixes that are no longer needed for any other can be *discarded*.

Algorithm 5.6. Prefix Doubling with Discarding.

Input : $k \in \mathbb{N}_0$.
Output : Suffix array $\mathsf{SA}[.]$

1 $S \coloneqq T.\text{Window}_2((i, [\,t_0, t_1\,]) \mapsto (i, t_0, t_1))$ *// Create initial triples $(i, T[i], T[i+1])$.*
2 $S \coloneqq S.\text{Sort}((i, r_0, r_1) \text{ by } (r_0, r_1))$ *// Sort triples by rank pairs.*
3 $N \coloneqq S.\text{FlatWindow}_2((i, [\,a, b\,]) \mapsto \text{CName}(i,\ a,\ b))$ *// Map ranks to 0 or i.*
4 $N \coloneqq N.\text{PrefixSum}(((i, r), (i', r')) \mapsto (i', \max(r, r')))$ *// Calculate initial ranks.*
5 **for** $k \coloneqq 1$ **to** $\lceil \log_2 |T| \rceil$ **do**
6 $\quad P \coloneqq N.\text{FlatWindow}_3((i, [\,a, b, c\,]) \mapsto \text{Unique}(a, b, c, i))$ *// Get states of items.*
7 $\quad P \coloneqq \text{Union}(P, U).\text{Sort}(<_{\text{op}}^{k})$ *// Concatenate undiscarded items and sort them.*
8 $\quad P \coloneqq P.\text{FlatWindow}_3((i, [\,a, b, c\,]) \mapsto \text{NPairs}(i, a, b, c, k))$ *// Compute new rank*
9 $\quad D' \coloneqq P.\text{Filter}((i, r_0, r_1, s)) \mapsto (s = \mathtt{d})$ *// pairs and update state. Then find and*
10 $\quad D \coloneqq \text{Union}(D, D').\text{Map}((i, r_0, r_1, s) \mapsto (i, a.r_0)))$ *// update discarded items.*
11 $\quad U' \coloneqq P.\text{Filter}((i, r_0, r_1, s) \mapsto (s = \mathtt{u}))$ *// Separate already unique items and*
12 $\quad U \coloneqq U'.\text{Map}((i, r_0, r_1, s) \mapsto (i, r_0, s))$ *// items that still need to be sorted. Former*
13 $\quad I' \coloneqq P.\text{Filter}((i, r_0, r_1, s) \mapsto (s = \mathtt{n}))$ *// are only needed to compute the rank pairs*
14 $\quad I \coloneqq I'.\text{Map}((i, r_0, r_1, s) \mapsto (i, r_0, r_1))$ *// and stored in U. Latter are stored in I.*
15 $\quad$ **if** $I.\text{Size}() = 0$ **then** *// If all items are unique*
16 $\quad\quad$ **return** $\mathsf{SA} \coloneqq D.\text{Sort}((i, r) \text{ by } r).\text{Map}((i, r) \mapsto i)$ *// return SA.*
17 $\quad M \coloneqq I.\text{FlatWindow}_2((i, [\,a, b\,]) \mapsto \text{RankDiscarding}(i, a, b))$ *// Form ranks*
18 $\quad M \coloneqq M.\text{PrefixSum}(((i, r_0, r_1, r_2), (i', r_0', r_1', r_2')) \mapsto$
$\quad\quad (i', \max(r_0', r_0'), \max(r_1', r_1'), r_2'))$
19 $\quad N \coloneqq M.\text{Map}((i, r_0, r_1, r_2) \mapsto (i, r_2 + (r_1 - r_0)))$ *// complying with old ranks.*

Using this classification, we can extend the prefix doubling algorithm using sorting for generating new ranks to exclude unique and discarded suffixes from expensive sorting operations. To be more precise, we can ignore discarded suffixes during all sorting operations, and unique suffixes are only required during the computation of the new ranks (Algorithm 5.4). Here, they are needed as second rank of the triple (n_0' in line 2). We do not emit a triple for an index corresponding to a unique suffix. Instead, we just store the unique pair. The algorithm terminates when all suffixes are either unique or discarded. To compute the final suffix array, we concatenate the unique and discarded pairs and sort them by their rank.

Since we ignore discarded suffixes during the computation of the new ranks, we must compute the new rank based on the old rank instead of the position among all other suffixes, as we did before. This can be done using multiple prefix sum operations, see lines 18 and 19 in Algorithm 5.6. In addition, we must keep track of the unique and discarded suffixes, but the total overhead is small compared to the savings during the sorting operations.

Initially, Algorithm 5.6 behaves like the generic prefix doubling algorithm (see Figure 5.2 for the data-flow graph). We compute rank pairs for consecutive text positions (line 1) and compute the ranks for all suffixes the same way we do in the

generic algorithm. Next, we add a *state* to the triples (i, r_1, r_2), i. e., creating 4-tuples (i, r_1, r_2, s), indicating whether a rank pair is *unique* ($s = \texttt{u}$) or *not unique* ($s = \texttt{n}$) (see Function Unique, Algorithm 5.7). All 4-tuples that are unique do not need a new rank but they may still be required to compute the new rank of another suffix. Hence we add a third state, a 4-tuple that is unique gets the state *discarded* (`d`) if it is not required for the computation of a different rank. Those tuples can easily be identified by looking at three consecutive tuples after they have been sorted using the less-comparator described in Equation (5.1). Let $a = (i, r_1, r_2, s), b = (i', r'_1, r'_2, s')$ and $c = (i'', r''_1, r''_2, s'')$ be three continuous tuples with s'' being unique. If either s or s' is unique, then c can be discarded because both a and b will get a unique rank pair during this iteration. Otherwise (if s or s' is not unique), c cannot be discarded as a will not get a unique rank pair during this iteration and we require the rank of c during the next iteration to compute the rank pair (see Function NPairs, Algorithm 5.7, lines 7–18). While computing the final state we create new rank pairs required for the new rank if the state is not unique, as otherwise the rank is final.

Since we do not consider all tuples during the course of Algorithm 5.6, we need to change the renaming based on the rank pairs. Up to now, we were able to assign ranks starting at 0 and continue based on the (preliminary) position in the suffix array. If we discard tuples, this approach is not feasible anymore as we need to consider the ranks of already discarded tuples. During the k-th iteration, all suffixes that do not have a unique rank form consecutive intervals in the suffix array. Within these intervals, all suffixes that cannot be distinguished by their first 2^k characters share the same rank. These ranks are extended, i. e., increased such that the new rank is always at least as great as the previous rank and greater than the rank of the first preceding suffix that can be distinguished using the first 2^k characters of the suffixes (lines 17–19). At the beginning of the next iteration, we add all unique ranks to the new ranks and check if they can be discarded. When all ranks are unique (line 15) we can compute the suffix array by sorting the discarded tuples by their ranks (line 16).

Prefix Quadrupling

The idea of prefix doubling can be generalized. In the prefix doubling algorithms described above, during the k-th doubling step, we consider substrings of length 2^k. However, we can also consider length-a^k substrings for any integer a with $a > 1$. To this end, we need a ranks to compute the new one (two for our prefix doubling algorithms in Step PD3). Using a ranks to compute new ranks—using a-tuples—obviously results in more data that we have to sort. In external memory, prefix doubling algorithms using sorting are I/O optimal for 5-tuples and in practice using 4-tuples, i. e., prefix quadrupling as shown by Dementiev et al. [Dem+08a]. Here, prefix quadrupling has the advantage that less memory is required for storing the tuples and that the I/O-volume is just 1.5 % worse compared to prefix quintupling. The change within our distributed algorithms can be kept to a minimum as we just require rank quadruples instead of rank pairs, as we can still use the same comparison operator (Equation (5.1)) for sorting. Our prefix quadrupling algorithms also employ the discarding technique.

Algorithm 5.7. Prefix Doubling with Discarding (Additional Functions)

Function *Unique*($j \in \mathbb{N}_0$, $(i, r), (i', r'), (i'', r'') \in N$)
  if $j = 0$ **then**
    emit $\begin{cases} (i, r, \mathtt{u}) & \text{if } r \neq r', \text{ // First item is unique} \\ (i, r, \mathtt{n}) & \text{otherwise. // if its ranks differ from its successor.} \end{cases}$
  else if $j + 2 = l$ **then**
    emit $\begin{cases} (i'', r'', \mathtt{u}) & \text{if } r' \neq r'', \text{ // Final item is unique} \\ (i'', r'', \mathtt{n}) & \text{otherwise. // if its ranks differs from its precursor.} \end{cases}$
  emit $\begin{cases} (i', r', \mathtt{u}) & \text{if } r \neq r' \text{ and } r' \neq r'', \text{ // An item is} \\ (i', r', \mathtt{n}) & \text{otherwise. // unique if its ranks are unique.} \end{cases}$

Function *NPairs*($j \in \mathbb{N}_0$, $(i, r, s), (i', r', s'), (i'', r'', s'') \in P$, $k \in \mathbb{N}_0$)
  if $j = 0$ **then**
    emit $\begin{cases} (i, r, 0, \mathtt{d}) & \text{if } s = \mathtt{u}, \text{ // The first two items can be discarded} \\ (i', r', 0, \mathtt{d}) & \text{if } s' = \mathtt{u}. \text{ // if they are unique. Emit} \leq 2 \text{ items.} \end{cases}$
  else if $j + 2 = l$ **then**
    if $s' = \mathtt{n}$ **then**
      emit $\begin{cases} (i', r', r'', \mathtt{n}) & \text{if } i' + 2^k = i'', \text{ // If the last two items of the} \\ (i', r', 0, \mathtt{n}) & \text{otherwise. // DIA are undecided, then we need} \end{cases}$
    if $s'' = \mathtt{n}$ **then**
      emit $(i'', r'', 0, \mathtt{n})$ // to fuse the ranks required for renaming.
  if $s = \mathtt{n}$ **then**
    emit $\begin{cases} (i, r, r', \mathtt{n}) & \text{if } i + 2^k = i', \text{ // The ranks for renaming are} \\ (i, r, 0, \mathtt{n}) & \text{otherwise. // consecutive and fused accordingly.} \end{cases}$
  if $s'' = \mathtt{u}$ **then**
    emit $\begin{cases} (i'', r'', 0, \mathtt{d}) & \text{if } s = \mathtt{u} \text{ or } s' = \mathtt{u}, \text{ // Unique items are dis-} \\ (i'', r'', 0, \mathtt{u}) & \text{otherwise. // carded if uncalled-for in future renaming.} \end{cases}$

Function *RankDiscarding*($j \in \mathbb{N}_0$, $l \in \mathbb{N}_0$, $(i, r_0, r_1), (i', r_0', r_1') \in I$)
  if $j = 0$ **then**
    emit $(i, 1, 1, r_0)$ // The new ranks must comply with the old ones.
  emit $\begin{cases} (i', j + 2, j + 2, r_0') & \text{if } r_0 \neq r_0' \text{ and } r_1 \neq r_1', \text{ // The first rank de-} \\ (i', 1, j + 2, r_0') & \text{else if } r_0 = r_0', \text{ // termines the group and new} \\ (i', 1, 1, r_0') & \text{otherwise. // names are consistent within groups.} \end{cases}$

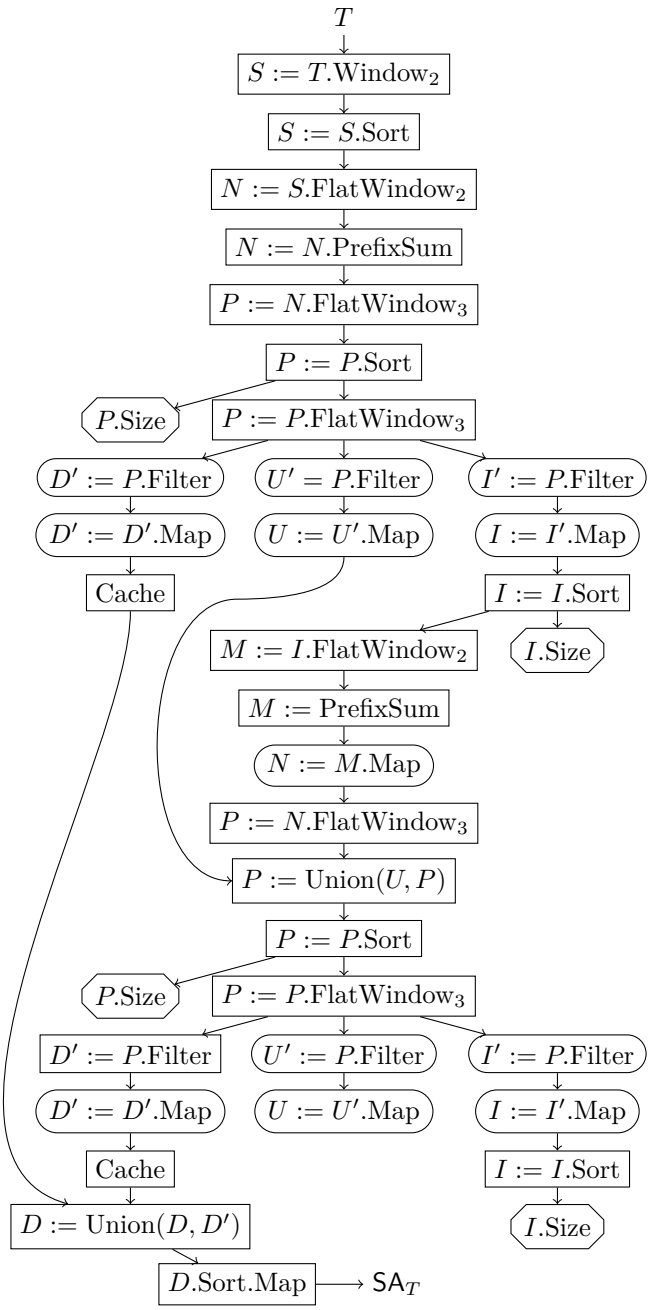


Figure 5.2. DIA data-flow graph of two iterations of prefix doubling with discarding that can easily be exported by Thrill.

5.2.2 Prefix Doubling in MPI

As mentioned before, our prefix doubling algorithms that we implemented using Thrill are not the first distributed suffix array construction algorithms, and also not the first distributed prefix doubling suffix array construction algorithms. There exists a distributed prefix doubling suffix array construction algorithm by Flick and Aluru [FA15] that is also realized using MPI.

Using MPI, we implemented the same ideas that we described in the previous section. When we use MPI, the main difference is that we can access arbitrary elements, which is not possible using DIAs in Thrill. This makes the *generation of ranks using the inverse suffix array* much easier, as we do not have to use a huge Window() operation but can just retrieve the required element from any processing element. Our approach to use *sorting to generate ranks* does not differ when we use MPI instead of Thrill; we can use the same idea and just have to realize sorting using MPI, which we describe below. The same is true for *discarding*. We did not consider prefix quadrupling in MPI, due to its high memory footprint.

Sample Sort Using MPI

Before we analyze the BSP cost of our distributed prefix doubling algorithms, we analyze our distributed sample sorting algorithm. When we sort data during our prefix doubling algorithm, we have keys of fixed length, i. e., the rank-tuples or triples. Hence, we can employ a distributed sample sort [Ble+96] to sort them and do not rely on a distributed string sorter (as we do in Section 5.4). Now, we want to sort m elements using p processing elements of size w bytes each. We assume that the elements are distributed such that there are $\Theta(m/p)$ elements on each processing element. Then, we can sort the data in five steps in parallel.

1. In the beginning, we sort the data locally in $\mathcal{O}(\frac{m}{p} \lg \frac{m}{p})$ time and choose $p-1$ local splitters from the local data, such that the p partitions that are implicitly given by the splitters have the same size (up to rounding).

2. Next, we gather all local splitters (on all processing elements) to determine $p-1$ global splitters (in the same way as in Step 1) in $\mathcal{O}(p \lg p + pwG + L)$ time.

3. We then use the global splitters, which are available on all processing elements, to partition the local data in $\mathcal{O}(\frac{m}{p} \lg p)$ time.

4. Now that we have all local data sorted and partitioned, we distribute these partitions in $\mathcal{O}(\frac{mw}{p}G + L)$ time, such that for any two processing elements i, j with $i < j$, all elements on processing element i are not larger than all elements on processing element j.

5. Finally, we merge the received partitions locally in $\mathcal{O}(\frac{m}{p} \lg p)$ time to finish the distributed sample sort.

Note that we can easily prevent an imbalance of data that is sent to one processing element in advance, by checking if any processing element would receive too much

data, as the sizes of the partitions are known, and resize the partitions accordingly. If we employ the sample sort as described above, we obtain the following lemma.

Lemma 5.1. *Using p processing elements, we can sort m elements each of size w bytes in $\mathcal{O}(\frac{m}{p}\left(\lg\frac{m}{p}+\lg p\right)+p\lg p+G\left(\frac{m}{p}+p\right)w+L)$ time.*

Cost of Distributed Prefix Doubling

During each iteration, the only difference is the computation of the names. We list the pseudocode for the generation of the ranks in Algorithms 5.3 and 5.4. There, it is easy to see that both approaches require the tuples to be sorted followed by a scan of the sorted data. However, even when using MPI, we would need the Window() operation, which requires $\mathcal{O}(nG)$ communication time in the worst case. We now analyze the prefix doubling algorithm *using sorting* in the following.

Given a text T of size n and p processing elements, distributed prefix doubling requires $\mathcal{O}(\lg n)$ iterations. During each iteration, we have to sort and scan the input a constant number of times. We distribute the input such that each of the processing element has a consecutive slice of size $\Theta(n/p)$ of the text.

Initially, in Step PD1, we just have to scan the text on each processing element. Since the rank of the suffix corresponds to the rank of its first character, this requires $\mathcal{O}(n/p)$ time. Next, in Step PD2, we have to check if all ranks are unique. To this end, we first sort the suffixes by their rank. Using Lemma 5.1 (with $m=n$), we can do so in time $\mathcal{O}(\frac{n}{p}(\lg\frac{n}{p}+\lg p)+p\lg p+(\frac{n}{p}+p)wG+L)$. Checking with consecutive processing elements requires additional $\mathcal{O}(wG+L)$ time. To compute the rank-triples in Step PD3, we simply scan the local data again and consider the consecutive rank-tuples that are on other processing elements and can be obtained in $\mathcal{O}(wG+L)$ time, which requires $\mathcal{O}(n/p+wG+L)$ time. Last, in Step PD4, we have to sort the triples, which again requires $\mathcal{O}(\frac{n}{p}(\lg\frac{n}{p}+\lg p)+p\lg p+(\frac{n}{p}+p)wG+L)$ time. Since we have to repeat this at most $\mathcal{O}(\lg n)$ times, we get the following lemma.

Lemma 5.2. *Using p processing elements, we can compute the suffix array of a text T of length n in $\mathcal{O}(\frac{n\lg n}{p}(\lg\frac{n}{p}+\lg p)+p\lg p\lg n+G(\frac{n}{p}+p)w\lg n+L\lg n)$ time using our prefix doubling algorithm based on sorting.*

Space Requirements. In addition to the BSP cost, we also analyze the space requirements of our distributed prefix doubling algorithm. The most space is required when we sort the rank-triples, as we have to (in the worst case) send and receive all rank-triples. Therefore, the total space required to hold all rank triples is $2wn$ bytes (as wn bytes are already required for the suffix array). Since we need send and receive buffers, this adds up to $5wn$ bytes. For the splitters we require $pw+p^2w$ bytes. Here, pw bytes are used for the local splitters and p^2w bytes in total. The space for the global splitters can also be used as receive buffer. In addition, we must keep track of all partitions, which requires additional p^2w bytes in total, i. e., pw bytes per processing element. Summing all of this up, we require $5wn+pw+2p^2w$ bytes for our distributed prefix doubling algorithm.

5.3 DISTRIBUTED RECURSIVE SUFFIX SORTING

In 2003, the *skew* aka *DC3* suffix sorting algorithm was proposed by Kärkkäinen and Sanders [KS03], and later generalized to *DC* by Kärkkäinen et al. [Kär+06]. They employ recursion on a subset of the suffixes to reach linear running time in the sequential RAM model. The algorithms were later implemented for external memory [Dem+08a], and DC3 for distributed memory using MPI [KS07]. We mention that this section is mainly the work of our co-authors in our publication [Bin+18], and we mainly mention it for completeness.

The key notion of DC is to recursively calculate the ranks of suffixes in only a *difference cover* [Sin38] of the original text. A set $D \subseteq \mathbb{N}_0$ is a difference cover for $v \in \mathbb{N}_0$, if $\{(i-j) \bmod v \mid i,j \in D\} = \{0,\ldots,v-1\}$. Examples of difference covers are $D_3 = \{1,2\}$ for $v = 3$, $D_7 = \{0,1,3\}$ for $v = 7$, and $D_{13} = \{0,1,3,9\}$ for $v = 13$. In general, a difference cover of size $\mathcal{O}(\sqrt{v})$ can be calculated for any v in $\mathcal{O}(\sqrt{v})$ time [Kär+06]. With respect to suffix sorting, the difference cover has the interesting property that it *samples* suffixes for recursive sorting such that the rank of all samples allows one to order the non-sampled suffixes using a *constant-time* comparison operation. The basic steps of the DC3 algorithm are the following:

(D1) Calculate ranks of all suffixes starting at positions in the difference cover $D_3 = \{1,2\}$ modulo 3. This is done by sorting the triples $(T[i], T[i+1], T[i+2])$ for $(i \bmod 3) \in D_3$, calculating lexicographic names, sorting the names back to string order, and recursively calling a suffix sorting algorithm on a reduced string T_R of size $\frac{2}{3}|T|$, if necessary. This reduced string represents *two* concatenated copies of the input string using the lexicographic names: the first copy are all names for suffixes with $i = 1 \bmod 3$ followed by a second copy for all suffixes with $i = 2 \bmod 3$. Hence, each character in T_R embodies three characters in T. Step D1) calculates two arrays, R_1 and R_2, containing the ranks of suffixes $i = 1 \bmod 3$ and $i = 2 \bmod 3$, which are computed by inverting the recursively constructed suffix array of T_R.

(D2) Scan the text T and rank arrays R_1 and R_2 to generate three arrays: S_0, S_1, and S_2, where array S_j contains one tuple for each suffix i with $i = j \bmod 3$. For each suffix i, the arrays store one tuple containing the two *following* ranks from R_1 and R_2 and all characters from T *up to* the next ranks. This is exactly the information required such that the following merge step is able to deduce the suffix array correctly. Due to the difference cover property the following rank for each suffix i is among the three elements $R_1[i]$, $R_1[i+1]$, and $R_1[i+2]$ for R_1, and analogously for R_2.

(D3) Sort S_0, S_1, and S_2 and merge them using a custom comparison function which compares the suffixes represented in the tuples using characters and ranks. Only a constant number of characters and ranks need to be accessed in each comparison. Output the suffix array using the indices stored in tuples.

The first two steps of the DC3 algorithms can be seen as preparation for the final merge in step D3). Step D1) delivers ranks for all suffixes $(i \bmod 3) \in D_3$ in R_1 and

R_2. In step D2) tuples are created in S_0, S_1, and S_2 which are constructed from the recursively calculated ranks and characters from the text. The tuples are designed such that the comparison function can fully determine the final suffix array.

The DC3 algorithm generalizes to DC using a difference cover D for any ground set size $v \geq 3$. DC constructs a recursive subproblem of size $\lceil (|T|/v)|D| \rceil$, which, considering $|D| = \mathcal{O}(\sqrt{v})$, is of size $\Theta(\frac{|T|}{\sqrt{v}})$. The algorithm has at most $\log_v |T|$ recursion levels and only one recursion branch. At every level of the recursion, only work with sorting complexity is needed, and a straight-forward application of the Master theorem to the recurrence $Z(|T|) = Z(\Theta(\frac{|T|}{\sqrt{v}})) + \mathcal{O}(\text{sort}(|T|))$ shows that the whole algorithm has sorting complexity due to the small recursive subproblem. For our distributed scenario, DC3 has the same complexity as sorting and merging.

Distributed Difference Cover Algorithms with Thrill

The complete DC3 implementation in Thrill algorithm code is shown as Algorithm 5.8. In the algorithm pseudocode we omitted some details on padding and sentinels for inputs that are not a multiple of the difference cover size, but our actual implementation in Thrill covers all these edge cases.

The goal of lines 2–20 is to calculate R_1 and R_2 as an interleaved array I_R. This is done by performing the following steps:

1. Scan the text T using a FlatWindow operation and create triples (i, c_0, c_1, c_2) for all indices $(i \bmod 3) \in D_3 = \{1, 2\}$ (lines 2–4).

2. Sort the triples as S, scan S and use a prefix sum to calculate lexicographic names N (lines 5–11). The lexicographic names are constructed in the prefix sum from 0 and 1 indicators. The value 0 is used if two lexicographic consecutive triples are equal, which means they are assigned the same lexicographic name; the value 1 increments the name in the prefix sum and assigns unequal triples a new names.

3. Check if all lexicographic names are different by comparing the highest lexicographic name against the maximum possible (lines 12–13).

4. If all lexicographic names are different, then I_S, which contains the indices of S, is already the suffix array of the suffixes in D_3 (lines 19–20). Hence, R_1 and R_2 can be created directly: the suffix array I_S only needs to be inverted and split modulo 3. However, instead of constructing R_1 and R_2 as separate DIAs, we *interleave* them in I_R using a Sort operation such that they are balanced on the distributed system, as we will be needing pairs of mod 1/2 ranks.

5. Otherwise, prepare a recursive subproblem T_R to calculate the ranks. First, sort the lexicographic names back into string order such that $T_R = T_1 \oplus T_2$ where $\oplus$ is string concatenation (line 14). T_1 represents the complete text T using the lexicographic names of all triples $i = 1 \bmod 3$, and T_2 is another complete copy of T with triples $i = 2 \bmod 3$. By replacing the triples with lexicographic names,

the original text is reduced by $\frac{2}{3}$. Second, recursively call any suffix sorting algorithm (e.g. DC3) on T_R (line 15). Last, invert the permutation SA_R to gain ranks R_1 and R_2 of triples of T in D_3, again interleave I_R such that R_1 and R_2 are distributed on the workers after the Sort operation.

With R_1 and R_2 interleaved in I_R from step D1) (lines 2–20), the objective of step D2) is to create S_0, S_1, and S_2 in line 22. Each suffix i has exactly one representative in the array S_j where $j = i \bmod 3$. Its representative contains the recursively calculated ranks of the two following suffixes in the difference cover from R_1 and R_2 (two consecutive items from I_R), and the characters $T[i], T[i+1], T[i+2]$ up to (but excluding) the next known rank.

For DC3 these are $T[i]$, $T[i+1]$, $I_R[\frac{2i}{3}]$, and $I_R[\frac{2i}{3}+1]$ for a suffix $i = 0 \bmod 3$ in S_0. $I_R[\frac{2i}{3}] = R_1[\frac{i}{3}]$ is the rank of the suffix $T[i+1..n)$ and $I_R[\frac{2i}{3}+1] = R_2[\frac{i}{3}]$ is the rank of suffix $T[i+2..n)$, which are both in the difference cover. We write the tuple as (i, c_0, c_1, r_1, r_2) where the indices are interpreted relative to $i \bmod 3$. Each suffix $i = 1 \bmod 3$ in S_1 stores $T[i]$, $R_1[\frac{i-1}{3}]$, and $R_2[\frac{i-1}{3}]$ and we write the tuples as (i, c_1, r_1, r_2) where the indices again are relative to $i \bmod 3$. And lastly, each suffix $i = 2 \bmod 3$ in S_2 stores $T[i]$, $T[i+1]$, $R_1[\frac{i-2}{3}+1]$, and $R_2[\frac{i-2}{3}]$ because $R_1[\frac{i-2}{3}+1]$ is the rank of suffix $T[i+2..n)$.

In the Thrill code we construct the tuples by zipping pairs from I_R, and three consecutive characters from T together. The ZipWindow Z' (line 22) delivers $(c_0, c_1, c_2, r_1, r_2)$ for each index $i = 0 \bmod 3$. To construct the tuples in S_i two adjacent tuples need to be used because S_2's element are taken from the next tuple. This can be done in Thrill using a Window operation of size 2 (line 23). Thus to construct S_0, S_1, and S_2, we take $(c_0, c_1, c_2, r_1, r_2)$ for each index $i = 0 \bmod 3$ and $(\bar{c}_0, \bar{c}_1, \bar{c}_2, \bar{r}_1, \bar{r}_2)$ for the next index $i \bmod 3{+}3$, and output $(3i{+}0, c_0, c_1, r_1, r_2)$ for S_0, $(3i{+}1, c_0, c_1, r_1, c_2, r_2)$ for S_1, and $(3i+2, c_2, r_2, \bar{c}_0, \bar{r}_1)$ for S_2, as described above (lines 24–26). The three arrays are then sorted and merged. Here, the comparison function compares two representatives characterwise until a rank is found. The difference cover property guarantees that such a rank is found for every pair S_i, S_j during the Merge (lines 27–31).

Most of the previous discussion on DC3 can be extended to DC7 straightforwardly: Sort by seven characters instead of three, construct $T_R = T_0 \oplus T_1 \oplus T_3$ in case not all character tuples are unique, and have step D1) deliver R_0, R_1, and R_3 containing the ranks of all suffixes $(i \bmod 7) \in D_7$.

Cost of Distributed Difference Cover. The BSP cost of the distributed DC3 algorithms is the following:

Lemma 5.3 ([Kär+06]). *The BSP cost of is* $\mathcal{O}(\frac{n \lg n}{p} + G(\frac{n \lg n}{p \lg(n/p)} w) + L \lg^2 p)$

Compared with the BSP cost of distributed prefix doubling (Lemma 5.2), DC3 requires less local computation and is slightly more communication efficient (by a factor of $1/\lg(n/p)$). Additionally, the number of synchronizations only depends on the number of processing elements, whereas the number of synchronizations of prefix doubling depends on the input size, which in practice is better.

Algorithm 5.8. DC3 Algorithm in Thrill.

```
1  Function DC3(T ∈ DIA⟨Σ⟩)
2      T_3 := T.FlatWindow_3((i, [c_0, c_1, c_2]) ↦ Triple(i, c_0, c_1, c_2))
3      with Function Triple(i ∈ ℕ_0, c_0, c_1, c_2 ∈ Σ)
4          if i ≠ 0 mod 3 then emit (i, c_0, c_1, c_2)
5      S := T_3.Sort((i, c_0, c_1, c_2) by (c_0, c_1, c_2))
6      I_S := S.Map((i, c_0, c_1, c_2) ↦ i)
7      N' := S.FlatWindow_2((i, [p_0, p_1]) ↦ CTriple(i, p_0, p_1))
8      with Function CTriple(i ∈ ℕ_0, (c_0, c_1, c_2), (c'_0, c'_1, c'_2))
9          if i = 0 then emit 0
10         emit (if (c_0, c_1, c_2) = (c'_0, c'_1, c'_2) then 0 else 1)
11     N := N'.PrefixSum()
12     n_sub = ⌈2|T|/3⌉,    n_mod1 = ⌈|T|/3⌉
13     if N.Max() + 1 ≠ n_sub then
14         T'_R := Zip([I_S, N], (i, n) ↦ (i, n))
                .Sort((i, n) by (i mod 3, i div 3))
15         SA_R := DC3(T'_R.Map((i, n) ↦ n))
16         I'_R := SA_R.ZipWithIndex((r, i) ↦ (r, i))
17         I_R := I'_R.Sort((r, i) by (r mod n_mod1, r))
18     else
19         R := I_S.ZipWithIndex((r, i) ↦ (r, i))
20         I_R := R.Sort((r, i) by (r div 3, r))
21     I_R := I_R.Map((r, i) ↦ (i + 1))
22     Z' := ZipWindow_[3,2]([T, I_R],
         (i, [c_0, c_1, c_2], [r_1, r_2]) ↦ (c_0, c_1, c_2, r_1, r_2))
23     Z := Z'.Window_2((i, [(z_1, z_2)]) ↦ (i, z_1, z_2))
24     S_0 := Z.Map((i, (c_0, c_1, c_2, r_1, r_2), (c̄_0, c̄_1, c̄_2, r̄_1, r̄_2)) ↦
         (3i + 0, c_0, c_1, r_1, r_2)).Sort((i, c_0, c_1, r_1, r_2) by (c_0, r_1))
25     S_1 := Z.Map((i, (c_0, c_1, c_2, r_1, r_2), (c̄_0, c̄_1, c̄_2, r̄_1, r̄_2)) ↦
         (3i + 1, c_1, r_1, r_2)).Sort((i, c_1, r_1, r_2) by (r_1))
26     S_2 := Z.Map((i, (c_0, c_1, c_2, r_1, r_2), (c̄_0, c̄_1, c̄_2, r̄_1, r̄_2)) ↦
         (3i + 2, c_2, r_2, c̄_0, r̄_1)).Sort((i, c_2, r_2, c̄_0, r̄_1) by (r_2))
27     return Merge([S_0, S_1, S_2], CmpDC3).Map((i, ...) ↦ i) with Function
         CmpDC3(z_1, z_2)
28         (c_0, r_1)      < (c'_1, r'_2)     if z_1 = (i, c_0, c_1, r_1, r_2) ∈ S_0,
                                                 z_2 = (i', c'_1, r'_1, r'_2)   ∈ S_1,
29         (c_0, c_1, r_2) < (c'_2, c̄'_0, r̄'_1) if z_1 = (i, c_0, c_1, r_1, r_2) ∈ S_0,
                                                 z_2                            =
                                                 (i', c'_2, r'_2, c̄'_0, r̄'_1)  ∈ S_2,
30         (r_1)           < (r'_2)           if z_1 = (i, c_1, r_1, r_2)    ∈ S_1,
                                                 z_2                            =
                                                 (i', c'_2, r'_2, c̄'_0, r̄'_1)  ∈ S_2,
31         and symmetrically if z_1 ∈ S_i, z_2 ∈ S_j with i > j .
```

5.4 DISTRIBUTED INDUCED COPYING

In this section, we take a look at a different technique to compute the suffix array: *induced copying*. As shown in Section 4.1, this technique is used in (one of) the fastest main memory suffix array construction algorithms, *DivSufSort* by Mori [Mor06], which we explained in detail in Section 4.2.

In this section, we give a distributed variant of DivSufSort, which we also implemented using MPI. First, in Section 5.4.1, we extend some of the definitions that we initially introduced to describe the main memory *DivSofSort* in Section 4.2. Then, we give an overview of the distributed algorithm and introduce the distributed array, which we use to describe and analyze our algorithm, in Section 5.4.2. In Sections 5.4.3–5.4.5, we describe the different steps of our algorithm, before we give a detailed analysis of space requirements and BSP cost in Section 5.4.6.

5.4.1 Extended Classification of Suffixes

Since we adapt DivSufSort to work in distributed memory, we extend the definitions that we introduced in Section 4.2. To give a little bit more context for our new definitions, we repeat some of the prior definitions. We mark everything that we have already defined previously in grey (■). Again, we use the classification introduced by Itoh and Tanaka [IT99] to distinguish between two *classes* of suffixes in combination with a notation established by Kärkkäinen et al. [Kär+17] for a similar classification [Non+11] that we now extend to better fit our distributed setting.

Let T be a text of size n over an alphabet of size σ. Following Definition 4.1, we classify all suffixes in one of two classes:

$$(C1)\ i \in C^- \iff T[i] > T[i+1] \text{ or } (T[i] = T[i+1] \text{ and } i+1 \in C^-) \text{ or } i = n-1,$$

$$(C2)\ i \in C^+ \iff T[i] < T[i+1] \text{ or } (T[i] = T[i+1] \text{ and } i+1 \in C^+).$$

Using these classes we defined one sub-class, which contains all suffixes that we have to sort to be able to induce all other suffixes in Definition 4.2.

$$(C3)\ i \in C^{+\triangleright} \iff i \in C^+ \text{ and } i+1 \in C^-.$$

In the following, we are also interested in another sub-class that is based on C^--suffixes, which allows us to define the suffix array in a more fine-grained way later on.

Definition 5.1. *Let T be a text of length n and a text position $i \in [0, n)$, then*

$$(C4)\ i \in C^{-\triangleright} \iff i \in C^- \text{ and } i+1 \in C^+.$$

Suffixes in this sub-class are not sorted but induced later on. However, we use them to reduce the number of entries of the suffix array that we have to scan during the inducing phase. See Section 5.4.5 for more details. The definitions of the fine grained intervals remain the same and work analogously for our new sub-class. In Figure 5.3a, we give an example of the extended classification. Remember that we say "a suffix $T[i..n)$ is in C" if $i \in C$, and $C^{-\odot} \coloneqq C^- \setminus C^{-\triangleright}$, where C can denote any (sub-)class.

i	0	1	2	3	4	5	6	7	8	9
T[i]	a	b	b	c	a	b	a	b	c	a
class	+	+	+	−	+	−	+	+	−	−
sub-class	⊙	⊙	▷	▷	▷	▷	⊙	▷	⊙	▷

(a)

i	0	1	2	3	4	5	6	7	8	9
SA[i]	9	4	0	6	5	1	7	2	8	3
$C_{\alpha\beta}$	$	ab			ba	bb	bc		ca	
class	−	+	+	+	−	+	+	+	−	−
sub-class	▷	▷	⊙	⊙	▷	⊙	▷	▷	⊙	▷

(b)

Figure 5.3. Classification of suffixes in text order (a) and suffix array order (b).

This still allows us to implicitly sort all suffixes lexicographically based on their class and first (two) characters; we can extend Lemma 4.1 as follows.

Lemma 5.4. *Let T be a text of size n over an alphabet Σ and $i, j \in [0, n)$, then*

1. *$T[i..n) < T[j..n)$, if $i \in C^{-}_{\alpha\beta}$ and $j \in C^{+}_{\alpha\beta}$ for any $\alpha, \beta \in \Sigma$,*
2. *$T[i..n) < T[j..n)$, if $i \in C^{+\triangleright}_{\alpha\beta}$ and $j \in C^{+\odot}_{\alpha\beta}$ for any $\alpha, \beta \in \Sigma$, and*
3. *$T[i..n) > T[j..n)$ if $i \in C^{-\triangleright}_{\alpha\beta}$ and $j \in C^{-\odot}_{\alpha\beta}$ for any $\alpha, \beta \in \Sigma$.*

Proof. We have already proven the first two statements in Section 4.2 where we introduced classification. The third statement can be proven analogously to the second one by simply changing the classes of the suffixes and thus the lexicographical order of the characters that distinguish the suffixes. □

Remember that $\overrightarrow{C}$ contains the starting positions of the suffixes in (sub-)class C in lexicographical order. Before, we could express the suffix array of a text over an alphabet $\Sigma = [0, \sigma)$ as $\mathsf{SA} = \overrightarrow{C^{-}_{00}}\overrightarrow{C^{+\triangleright}_{00}}\overrightarrow{C^{+\odot}_{00}}\overrightarrow{C^{-}_{01}}\overrightarrow{C^{+\triangleright}_{01}}\overrightarrow{C^{+\odot}_{01}} \dots \overrightarrow{C^{-}_{\sigma-1\sigma-1}}\overrightarrow{C^{+\triangleright}_{\sigma-1\sigma-1}}\overrightarrow{C^{+\odot}_{\sigma-1\sigma-1}}$, see Observation 4.1. Using Lemma 5.4 we can split the C^{-}-suffixes and get the following observation, which we also visualize in Figure 5.3b.

Observation 5.1. *We can express the suffix array of a text over an alphabet $\Sigma = [0, \sigma)$ as follows:*

$$\mathsf{SA} = \overrightarrow{C^{-\odot}_{00}}\overrightarrow{C^{-\triangleright}_{00}}\overrightarrow{C^{+\triangleright}_{00}}\overrightarrow{C^{+\odot}_{00}}\overrightarrow{C^{-\odot}_{01}} \dots \overrightarrow{C^{-\odot}_{\sigma-1\sigma-1}}\overrightarrow{C^{-\triangleright}_{\sigma-1\sigma-1}}\overrightarrow{C^{+\triangleright}_{\sigma-1\sigma-1}}\overrightarrow{C^{+\odot}_{\sigma-1\sigma-1}}.$$

5.4.2 General Overview

Using the classification, we can compute the suffix array in three steps. Before that, we have to consider how we distribute the text on all processing elements. For the remaining section, we use the following notations for the text and its suffixes. Given a text T of size n, we want to compute the suffix array using p processing elements. We assume that T is distributed among all processing elements, such that each processing element holds a consecutive slice T' of size $n' = \Theta(n/p)$. Thus, $T'[j] \coloneqq T[i\lfloor n/p \rfloor + j]$ on the i-th processing element for $i \in [0, p)$. Similarly, S'_j denotes the j-th suffix of T' with respect to the whole text, i. e., on processing element i we have $S'_j = T[i\lfloor n/p \rfloor + j..n)$ for $i \in [0, p)$. Now, our distributed DivSufSort requires three steps:

1. On each processing element, we compute $C^{+\triangleright}$ and the sizes of $C_{\alpha\beta}$ for all $\alpha, \beta \in \Sigma$ and (sub-)classes C for T'. The results are communicated to get those sizes for the whole text T. We describe this step in Section 5.4.3.

2. Next, in Section 5.4.4, we sort all suffixes in $C^{+\triangleright}$ lexicographically to compute $\overrightarrow{C^{+\triangleright}}$ using a distributed string sorting algorithm, which we describe comprehnsively in Section 5.5.

3. Last, we induce $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$ using $\overrightarrow{C^{+\triangleright}}$ and then $\overrightarrow{C^{-\odot}}$ using $\overrightarrow{C^{-\triangleright}}$, which we describe in Section 5.4.5.

Distributed Arrays. During the computation of the suffix array, we make heavy use of *distributed arrays* that distribute data similar to how we distributed the text above but provide additional functionality. We use distributed arrays to store the suffixes in the different (sub-)classes $C_{\alpha\beta}$ for $\alpha, \beta \in \Sigma$. Here, each processing element holds a consecutive slice of the array of fixed size, i. e., given a distributed array C of fixed length ℓ, each processing element holds $\Theta(\ell/p)$ elements, such that on the i-th processing element the j-th local element is the $j + i\lfloor \ell/p \rfloor$-th global element.

A distributed array supports two operations: *pushback* and *pushfront* put data in the rightmost or leftmost unused space, respectively. The space is reserved beforehand. Executing one operation takes one superstep, independently of the amount of data stored. Since the operation can be called from multiple processing elements during one superstep, the data is stored in an order depending on the rank of the processing elements that send the data. When we insert data into a distributed array (using *pushfront*) originating from processing element i and processing element j with $i < j \in [0, p)$, then all data sent by processing element i will have a smaller index in the distributed array than any data sent by processing element j (in the same superstep).

The operations are executed *delayed*, meaning that elements are not stored immediately, but buffered until *communicate()* is called, which we use in Algorithm 5.9. We only insert data using the operations described above or access already stored data. When we say in "reverse order", we access all elements stored in the distributed array from right to left. We indicate two concatenated arrays using $\otimes$. In this case, the whole arrays are concatenated, not just the local slices.

5.4.3 Identifying Suffixes in Distributed Memory

We first need to identify those suffixes that are in $C^{+\triangleright}$, which can be done by a right to left scan of the text. The last suffix is in C^- and thus we only need to look at two consecutive characters to identify the type of a suffix.

Observation 5.2. *Let $i \in [0, n-1)$. We know that $n-1 \in C^-$. If $T[i] > T[i+1]$, then $i \in C^-$ and if $T[i] < T[i+1]$, then $i \in C^+$. Last, if $T[i] = T[i+1]$, then $i \in C^- \Leftrightarrow i+1 \in C^-$.*

Hence, if we know the type of the last suffix on each processing element, we can classify all suffixes. Alas, in our distributed setting, only for the $p-1$-th processing element the class of the last suffix is known. Hence, we cannot simply scan T' right to left on any processing element but the last. Instead, we identify the first suffix S_i that is definitely in C^-, i. e., the rightmost position where $T'[i] > T'[i+1]$ with $i < n'-1$. Starting at this suffix, we can use Observation 5.2 to classify all suffixes S'_j with $j < i$. Next, we identify the classes of all remaining suffixes. To this end, each processing element sends $T'[0]$ and the class of S'_0 to all other processing elements. The class on the i-th processing element can be *unknown*, i. e., there has been no suffix that is definitely in C^-. In this case, we can conclude the type of all suffixes on processing element i using the class and the first character of the first suffix on processing element $i+1$. Since the class is known on the $p-1$-th processing element, we can resolve the class of all received suffixes and thus, we can classify all suffixes that have not been classified, yet. Within those suffixes, there is at most one suffix in $C^{+\triangleright}$.

In total, we scan the local text at most twice, send $\mathcal{O}(1)$ computer words and receive $\mathcal{O}(p)$ computer words in one communication phase. In practice, the communication overhead is very small ($p \ll n$) and there are (again, only in practice) no unknown suffixes. Still, we could further reduce the communication in exchange for more supersteps using a prefix sum-like approach to resolve unknown classes. This results in costs of $\mathcal{O}(n/p + pG + \lg pL)$, which leads to the following lemma.

Lemma 5.5. *Let T be a text of length n. Using p processing elements, identifying all suffixes in $C^{+\triangleright}$ costs $\mathcal{O}(n/p + pG + \lg pL)$ time.*

In addition to identifying the suffixes in $C^{+\triangleright}$, we compute the number of suffixes in all other (sub-)classes without any overhead in running time. These sizes are needed to determine the size of the distributed arrays we need for inducing the suffix array in Section 5.4.5 based on the different sub-clases as described in Observation 5.1.

Space Requirements. We need w bytes to store an index position in the suffix array, usually $w = 4$ for smaller texts and $w = 5$ for larger texts (up to 1 TiB). In theory, $\lceil \lg n \rceil$ bits are sufficient. In practice, we use a multiple of one byte for faster access. Here, we assume that we need one byte per character, i. e., $\sigma < 256$.

We need at most $wn/2$ bytes to store $C^{+\triangleright}$ in a distributed array. Since we need to communicate the suffixes, this requires twice the amount of space, resulting in wn bytes. Storing all those positions requires $2\sigma^2 w$ bytes space on each processing elements, $\sigma^2 w$ bytes for all suffixes in C^- and the same amount for the suffixes in C^+.

Substring s_1	a	b	a	a	b
Substring s_2	a	b	a	b	

Figure 5.4. Two $C^{+\triangleright}$-substrings. The underlined characters correspond to positions in $C^{+\triangleright}$. If we consider only the substrings starting and ending at those positions, s_2 is lexicographically smaller than s_1, as it is a prefix of s_1. This can be avoided by considering one additional character.

5.4.4 Sorting of Suffixes in Distributed Memory

We compute $\overrightarrow{C^{+\triangleright}}$ in two steps. The first step is the same as in main memory. We sort the substrings between two adjacent positions in $C^{+\triangleright}$ (in text order). Formally, let $\text{next}(i) = \min\{j > i \colon j \in C^{+\triangleright} \cup \{n\}\}$. This allows us to define the $C^{+\triangleright}$-*substrings* as $T_i^{+\triangleright} = T[i..\min\{\text{next}(i) + 2, n\})$, see Definition 4.3. (The additional two characters are important to correctly sort the $C^{+\triangleright}$-substrings, see Figure 5.4 for an example.) We sort the $C^{+\triangleright}$-substrings using a distributed string sample sort. A detailed description of the sorting is given in Section 5.5, where we describe general purpose distributed string sorting algorithms. Sorting $C^{+\triangleright}$-substrings is just a special case.

Here, the number $C^{+\triangleright}$-substrings m is at most $n/2$ and the *distinguishing prefix size* D, i. e., the number of characters that must be compared to sort the $C^{+\triangleright}$-substrings lexicographically, is in practice roughly the same at each processing element (which is confirmed by our experiments, see Table 5.1). Hence, we can sort the $C^{+\triangleright}$-substrings in $\mathcal{O}(n \lg \sigma/p + nG/p + L)$ time, using our distributed string sorter, which employs sample sort and sorts the strings locally using multi-key radix sort.

Having sorted the $C^{+\triangleright}$-substrings, we sort all suffixes in $C^{+\triangleright}$ by using the ranks of the $C^{+\triangleright}$-substrings. We use an approach similar to prefix doubling (see Section 5.2 for details on prefix doubling) with one difference: Instead of using the original input text, we use the ranks of the $C^{+\triangleright}$-substrings (in text order) as input T^{ranks} for the prefix doubling algorithm. During each iteration we double the size of the considered prefixes in T^{ranks}. Thus, we implicitly double the number of considered consecutive $C^{+\triangleright}$-substrings in T, which is similar to this part of DivSufSort in main memory. Then, we compute the new ranks using the old ones until all ranks are unique, as described in the prefix doubling suffix array construction algorithms in Section 5.2.

Since the prefix doubling algorithm relies on indices in the range from 0 to m, where m is the number if considered suffixes/substrings, we must transform the text positions of the $C^{+\triangleright}$-substrings accordingly. When all suffixes in $C^{+\triangleright}$ are sorted, we reverse the transformation by first sorting the rank-tuples in text order, and then identifying all suffixes in $C^{+\triangleright}$ during a single scan of the text. Using Lemma 5.2 we get the following lemma.

Lemma 5.6. *Sorting all suffixes in $C^{+\triangleright}$ lexicographically, i. e., computing $\overrightarrow{C^{+\triangleright}}$, costs $\mathcal{O}((\lg \frac{n}{p} + \lg p)n \lg n/p + p \lg p \lg n + G(n/p + p)w \lg n + L \lg n)$ time.*

Algorithm 5.9. Distributed Inducing the Suffix Array

```
1  for α = σ − 1 down to 0 do
2  |  for β = σ − 1 down to α do
3  |  |  for i ∈ C^{+▷}_{αβ} ⊗ C^{+⊙}_{αβ} in reverse order do
4  |  |  |  if i > 0 and T[i − 1] ≤ α then
5  |  |  |  |  C^{+⊙}_{T[i−1]α}.pushfront(i − 1)
6  |  |  |  else if i > 0 then
7  |  |  |  |  C^{−▷}_{T[i−1]α}.pushfront(i − 1)
8  |  |  communicate()
9  C^{−⊙}_{T[n−1]0}.pushback(n − 1)
10 for α = 0 to σ − 1 do
11 |  for β = 0 to α do
12 |  |  for i ∈ C^{−⊙}_{αβ} ⊗ C^{−▷}_{αβ} do
13 |  |  |  if i > 0 and T[i − 1] ≥ α then
14 |  |  |  |  C^{−⊙}_{T[i−1]α}.pushpack(i − 1)
15 |  |  communicate()
```

Space Requirements. The space requirements for this phase of our algorithm is the same as for distributed prefix doubling (similar to the BSP cost). However, we know that there are at most $n/2$ suffixes that we need to sort. While this does not change the asymptotical BSP cost, we now only require wn bytes for the two ranks. Since we can use the space for the suffix array for both sending and receiving, we only require additional $2wn + pw + 2p^2w$ bytes for this phase, i. e., in addition to the space that is already required for the suffix array, which is not used at this point in time.

5.4.5 Inducing the Suffix Array

Now, we compute the suffix array by inducing all other suffixes using only $\overrightarrow{C^{+\triangleright}}$ and T, without any sorting. First, we induce $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$ from $\overrightarrow{C^{+\triangleright}}$ and the already induced suffixes in $\overrightarrow{C^{+\odot}}$. Then, we add the last suffix to its correct position in $C^{-\triangleright}$. Last, we induce $\overrightarrow{C^{-\odot}}$ from $\overrightarrow{C^{-\triangleright}}$, see Algorithm 5.9.

We assume that the sorted suffixes in $\overrightarrow{C^{+\triangleright}_{\alpha\beta}}$ are stored in distributed arrays for all $\alpha, \beta \in \Sigma$. In general, all C-objects in the algorithm are distributed arrays. The distributed arrays are named after the (sub-)classes to which the suffixes belong. The algorithm runs on all processing elements, and each processing element only considers its own slice of the distributed arrays when reading from it (but the concatenation in lines 3 and 12 still affects the whole distributed array). We first induce from right to left (first inducing phase), i. e., in decreasing lexicographical order (see loop starting at line 1). Here, we induce all suffixes in $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$. Next, we add the last suffix (line 9) before starting the second inducing phase. Last, we induce the suffixes in increasing lexicographical order (see loop starting at line 10).

During this step, we require access to the text (from arbitrary processing elements), since we need to identify the bucket we induce the suffix into. To this end, we distribute the text as before. Now, when we require the i-th character, we know that it is stored on processing element i/ℓ and is the $i\%\ell$-th character in the local slice, where % denotes the *modulo* operator. Then, before we induce the next suffixes (pushfront or pushback operation in Algorithm 5.9), we retrieve the first character of *all* suffixes that are induced during this step in *one* communication phase, which allows us to induce into the correct distributed array.

In practice, we can also make use of the property that not all classes contain suffixes. The inner loops (see lines 2 and 11) are implemented such that they skip distributed arrays that cannot contain any (relevant) suffixes, which are characterized by the following lemma:

Lemma 5.7. *Let be $\alpha, \beta \in \Sigma$, then*

1. $\alpha < \beta \Rightarrow C^{-}_{\alpha\beta} = \emptyset$,
2. $\alpha > \beta \Rightarrow C^{+}_{\alpha\beta} = \emptyset$, *and*
3. $\alpha = \beta \Rightarrow C^{-\triangleright}_{\alpha\beta} \cup C^{+\triangleright}_{\alpha\beta} = \emptyset$.

Proof. Due to the definition of $C^{-}_{\alpha\beta}$ and $C^{+}_{\alpha\beta}$ ($i \in C^{-} \Rightarrow T[i] \geq T[i+1]$ and $i \in C^{+} \Rightarrow T[i] \leq T[i+1]$), the first two statements are true. To prove the third statement we assume that $i \in C^{+\triangleright}$. Therefore, $T[i..n) < T[i+1..n)$ and $i+1 \in C^{-}$. This leads to $T[i] = T[i+1] \geq \cdots > T[i+j]$ with $T[i+j]$ being the first character strictly smaller than $T[i+1]$. This contradicts our initial assumption. The proof of the last case ($\alpha = \beta \Rightarrow C^{-\triangleright}_{\alpha\beta} = \emptyset$) works analogously. □

For each pair of characters there is a communication phase (lines 8 and 15). This would be sufficient if we did not insert into distributed arrays that we are currently traversing. Unfortunately, there is one case where this can happen. We describe how to handle this special cases in the next paragraph.

The Special Case

A *run* of length r denotes a substring $T[i..i+r)$ with $T[i] = T[i+1] = \cdots = T[i+r-1]$ for $i \in [0, n-r)$. The algorithm (as just described) cannot handle length-3 or longer runs, as this would require to induce in the same distributed array that we are currently traversing (lines 3 and 12). Since the arrays are updated just before the next character combination is considered, we never use the newly induced suffixes to further induce any suffix. Fortunately, handling those runs is easy in the distributed setting. If multiple runs of the same character occur, suffixes that are induced by the rightmost suffix in the run are interleaved. E. g., if we induce suffixes i and j in runs from suffix array positions k and ℓ with $k < \ell$ during the first inducing step, we know that $i-1$ occurs left of $j-1$ in the suffix array. This repeats until one or both runs end and can be generalized for an arbitrary number of runs, see Figure 5.5 for an example.

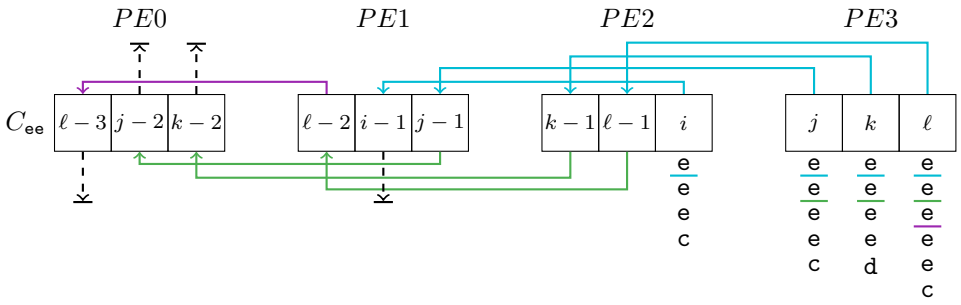


Figure 5.5. The distributed array C_{ee} of size 12 on four processing elements. Initially, the text positions i, j, k and ℓ are contained in C_{ee} (the text starting at those positions up to the first mismatching character is given below these entries). In this example, we induce from right to left. The arrows indicate the interleaved occurrences of the induced suffixes that can be induced by just the length of the run. The colors of the arrows indicate the starting position of the suffix (see the horizontal bars in the text). Arrows ending in a bar indicate that the suffix is induced into a different distributed array.

Hence, we can compute the part of the suffix array where the runs occur using only their length and the suffix array position they have been induced from.

To this end, we first determine all runs that must be contained in the currently considered distributed array and compute their lengths. Next, we communicate this information among all processing elements. Now, each processing element can determine which entries must be stored in its locals slice of the distributed array, by simply unrolling the runs similar to the example given in Figure 5.5.

The Cost of Inducing

In total, we consider each entry of the suffix array exactly once. Since we use distributed arrays, we know that the number of suffix array positions on all processing elements is the same up to rounding. Also, the maximum number of computer words sent and received is (asymptotically) the same. While there can be communication phases where only one processing element receives data, all processing elements must receive the same amount of data at some point (as the content of the distributed array is stored equally among all processing elements). The additional costs of the steps required by the special case are dominated by the costs described above. Since we need a communication phase for each pair of characters, we require $\mathcal{O}(\sigma^2)$ supersteps. All this requires wn bytes to store the induced suffixes and at most twice as much to send the positions, when we store them in a distributed array. In total, we can induce all remaining suffixes in the following time.

Lemma 5.8. *We can induce all suffixes in $C^{-\triangleright}, C^{-\odot}$ and $C^{+\odot}$ in $\mathcal{O}(\frac{n}{p} + \frac{n}{p}G + \sigma^2 L)$ time.*

Last, we need to transform the distributed arrays, such that all suffixes on processing element i are lexicographically smaller than all suffixes on processing element j if $0 \leq i < j < p$, i. e., compute the final suffix array. To this end, we compute the number of elements (in each distributed array) that we must sent to each processing element, then during one large communication phase, we send them accordingly. The memory required during this phase is $2w\frac{n}{p}$ bytes per processing element.

5.4.6 Space and Time Requirements

The most memory is required during the sorting of the suffixes in $C^{+\triangleright}$, where we need $3wn$ bytes of memory in addition to the text (n bytes) and $2\sigma^2 wp$ bytes for the size of the (sub-)classes. During the classification, we need $wn + 2\sigma^2 wp$ bytes and the text. Last, when inducing all other suffixes, we need wn bytes in addition to the text. This results in a maximum $3wn/p + 2\sigma^2 w$ bytes per processing element, when we distribute all data equally among all processing elements. Using Lemmas 5.5, 5.6, and 5.8 we get the following:

Corollary 5.1. *Using p processing elements, our distributed induced copying algorithm can compute the suffix array of a text T of length n over an alphabet of size σ in $\mathcal{O}(\frac{n \lg n}{p}(\lg \frac{n}{p} + \lg p) + p \lg p \lg n + (\frac{n}{p} + p)w \lg nG + (\lg n + \sigma^2)L)$ time, using $3wn + 2\sigma^2 wp$ bytes of space.*

Here, we want to note that the factor of σ^2 in the space and in the costs for the synchronization steps implies that this algorithm is only applicable to at most medium-sized alphabets. However, as we show in our experiments in Section 5.6, it is totally reasonable for byte alphabets.

5.5 DISTRIBUTED STRING SORTING

In this section, we describe a distributed variant of *string sample sort* [Bin+17], which we used in Section 5.4.4 to sort the $C^{+\triangleright}$-substrings. Usually, *atomic* keys (keys that can be compared with a single comparison) are considered when sorting data. Longer strings, on the other hand, cannot be compared with a single comparison, but have to be compared character by character. Techniques like *word packing* allow us to compare as many characters at once as fit into one hardware register, but still require non-constant time in general.

Let $S \coloneqq \{s_0, \ldots, s_{m-1}\}$ be a set of m strings and let D be the *distinguishing prefix size* of S, i. e., the number of characters that must be compared to sort S lexicographically. Formally, for a set $S \coloneqq \{s_0, \ldots, s_{m-1}\}$ of strings, $D \coloneqq 1 + \sum_{i=0}^{m-1} \max\{\mathsf{lcp}(s_i, t) : t \in S \setminus s_i\}$, where $\mathsf{lcp}(s, t) = \max\{i \colon s[0..i) = t[0..i)\}$. Further, let $M \coloneqq \sum_{i=0}^{m-1} |s_i|$ be the total length of all strings in S.

The strings are distributed among all processing elements such that each processing element holds roughly the same number of characters (if possible). For simplicity, we assume that on each processing element the local distinguishing prefix size, i. e., the number of characters that must be compared to sort all strings that are stored at the

processing element, is $D' = \Theta(D/p)$ and that on each processing element there are strings of total length of $m' = \Theta(M/p)$ characters.

Those assumptions are feasible in our scenario as we focus on the sorting of $C^{+\triangleright}$-substrings that all have a similar (short) length, see Table 5.1 for practical measurements confirming this simplifying assumption. Splitters are chosen in the same fashion they are chosen in our distributed sample sort for atomic keys, which is described in Section 5.2.2:

1. We first sort all strings locally on all processing elements and determine the local splitters. These splitters are then shared among all processing elements, and a common set of $p - 1$ global splitters is chosen.

2. Using the global splitters, on each processing element we determine p intervals (on the locally sorted strings) that have the global splitters as borders (the first and last interval has only one global splitter as upper and lower border, respectively).

3. We distribute the strings in these intervals among all processing elements, such that the strings in the i-th interval on any processing element are sent to the i-th processing element.

4. Since all strings that have been sent to any processing element are sorted, we simply merge the received intervals locally to obtain the final sorting.

The most time consuming task is the sorting in Step 1. Steps 2–4 work similar to the distributed sample sort described in Section 5.4.4. The only difference is that we need to consider strings instead of atomic keys.

Note that the sorting as described here differs from the canonical sample sort, as we first sort locally on each processing element before we determine the splitters. Since we sort the strings locally, we only have to merge them later. Hence, this approach could also be denoted as a sample and merge sort hybrid.

5.6 EXPERIMENTAL EVALUATION

Due to the number of algorithms that we evaluate in this section, we split this evaluation in three parts:

1. In Section 5.6.1, we discuss the distributed suffix array construction algorithms that we implemented using Thrill. We consider them separately, as they achieve an obviously lower throughput than the algorithms implemented using MPI. This, however, can to some extend be explained with our experimental setup. To highlight this, we present some results from [Bin+18], where we had a setup that works better with the Thrill implementations.

2. Next, in Section 5.6.2, we present the results for our distributed suffix array construction algorithms that are implemented using MPI.

3. Finally, in Section 5.6.3, we give the results for distributed string sorting algorithms. This evaluation is also used to determine the string sorting algorithm that we use for our distributed suffix sorting algorithm based on DivSufSort.

The setup for all experiments in this section is the same. We conducted the experiments using LiDO.small nodes (Section 1.4.1) and the inputs described in Section 1.4.2. The code for the suffix sorting algorithms implemented using Thrill (1) is part of the Thrill framework and available at https://github.com/thrill/thrill. The implementations that use MPI (2) and (3) are available at `www.kurpicz.org/dsaca`. We compiled the code using GCC 9.2.0 with flags `-O3` and `-march=native`. As in all previous experiments, we start the timing as soon as the input in available in the main memory of all processing elements and stop the timing as soon as all suffixes (1 and (2)) or strings (3) are sorted. All results are the average of five executions of the algorithm.

5.6.1 Evaluation of Distributed Suffix Sorting using Thrill

In this section, we present the results of our distributed suffix sorting algorithms implemented using Thrill. The algorithms are described in Sections 5.2 and 5.3. Here, *T.DC3* and *T.DC7* denote the difference cover algorithm with difference cover sizes of 3 and 7. The prefix doubling algorithms are denoted by *T.PD-Discarding* and *T.PD-Window*. The former is the algorithm that discards tuples that are not required any more and the latter is the one that uses a window of size 2^k in the k-th iteration to determine the new ranks. Finally, *T.PQ* denotes our prefix quadrupling algorithm that also makes use of discarding.

We show the throughput of the algorithms in Figure 5.6. We conducted a weak scaling experiment, i. e., we use 256 MiB and 512 MiB of input per node. Surprisingly, the throughput is higher when using one node than when using two. The algorithms achieve a higher throughput than on one node only when we use more than 16 nodes. If we consider only two or more nodes, the algorithms seem to scale well. Later, we show results from experiments on different hardware where the algorithms scale well on any number of nodes.

Nevertheless, T.PD-Window is the slowest of the algorithm on all inputs, which comes without surprise, as larger windows are expensive to compute. Next, T.DC3 is the second slowest algorithm on all inputs. On `CommonCrawl` and `Wiki`, T.DC7 is the fastest algorithm. On texts with smaller alphabet, T.PD-Window is the fastest algorithm. It is always the fastest on `Prot` and on `DNA` (512 MiB per node). When using 256 MiB of input per node, T.DC7 is faster up to 8 nodes, using 16 nodes or more, T.PD-Discarding becomes faster. Using prefix quadrupling (T.PQ) is never the fastest nor the slowest suffix sorting algorithm.

We show the memory requirements of all algorithms in Figure 5.7. As expected, the memory peak of all algorithms does not depend on the input. All algorithms but T.PD-Discard show a slight increase of the memory peak when comparing the results on 1 and 4 nodes. Using more nodes does not increase the memory peak. In general, T.PD-Window is the most memory efficient algorithm and T.DC3 has a 10 % higher memory peak. On all instances, T.PQ has the highest memory peak.

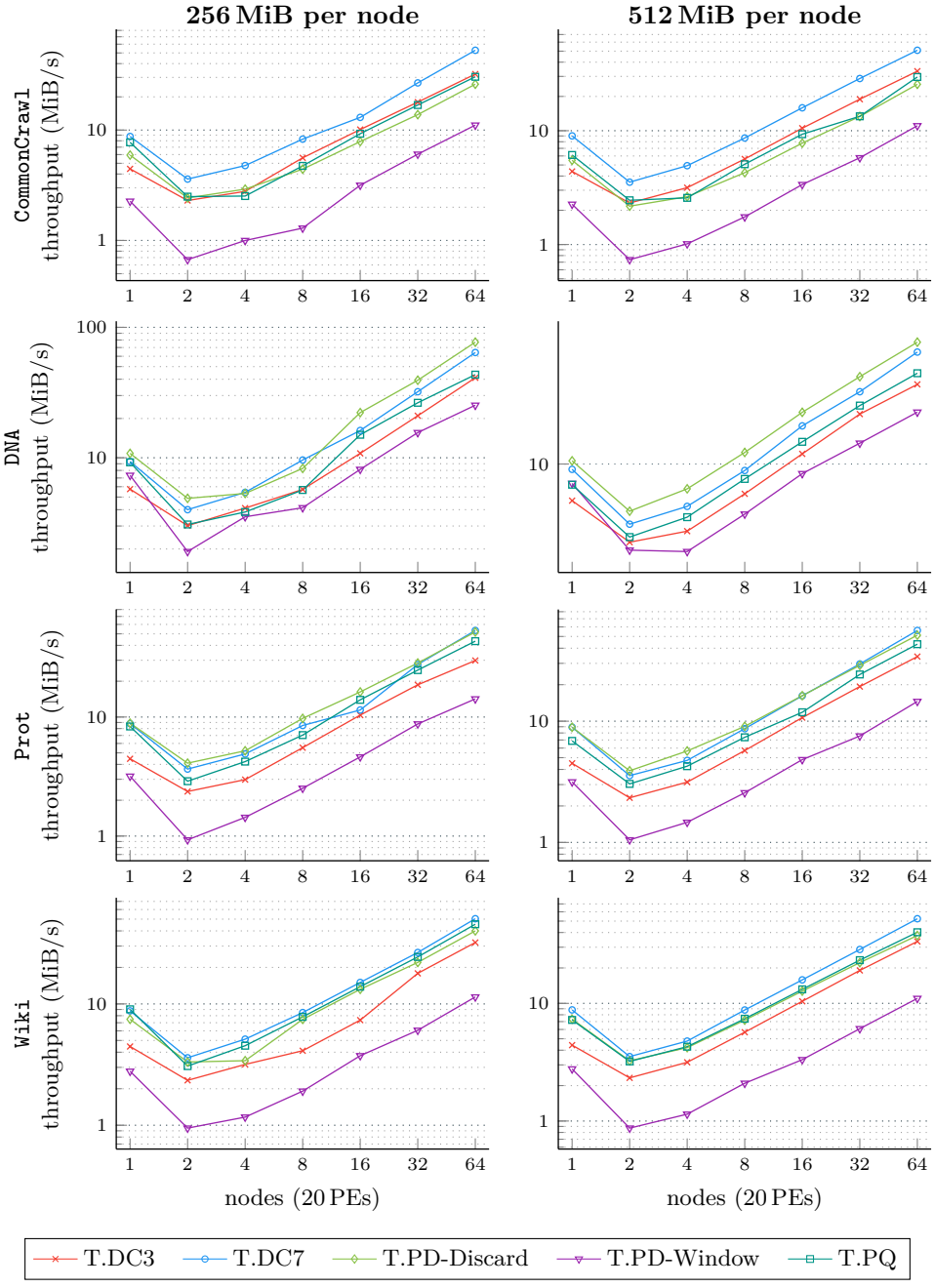


Figure 5.6. Throughput of distributed suffix sorting algorithms (Thrill).

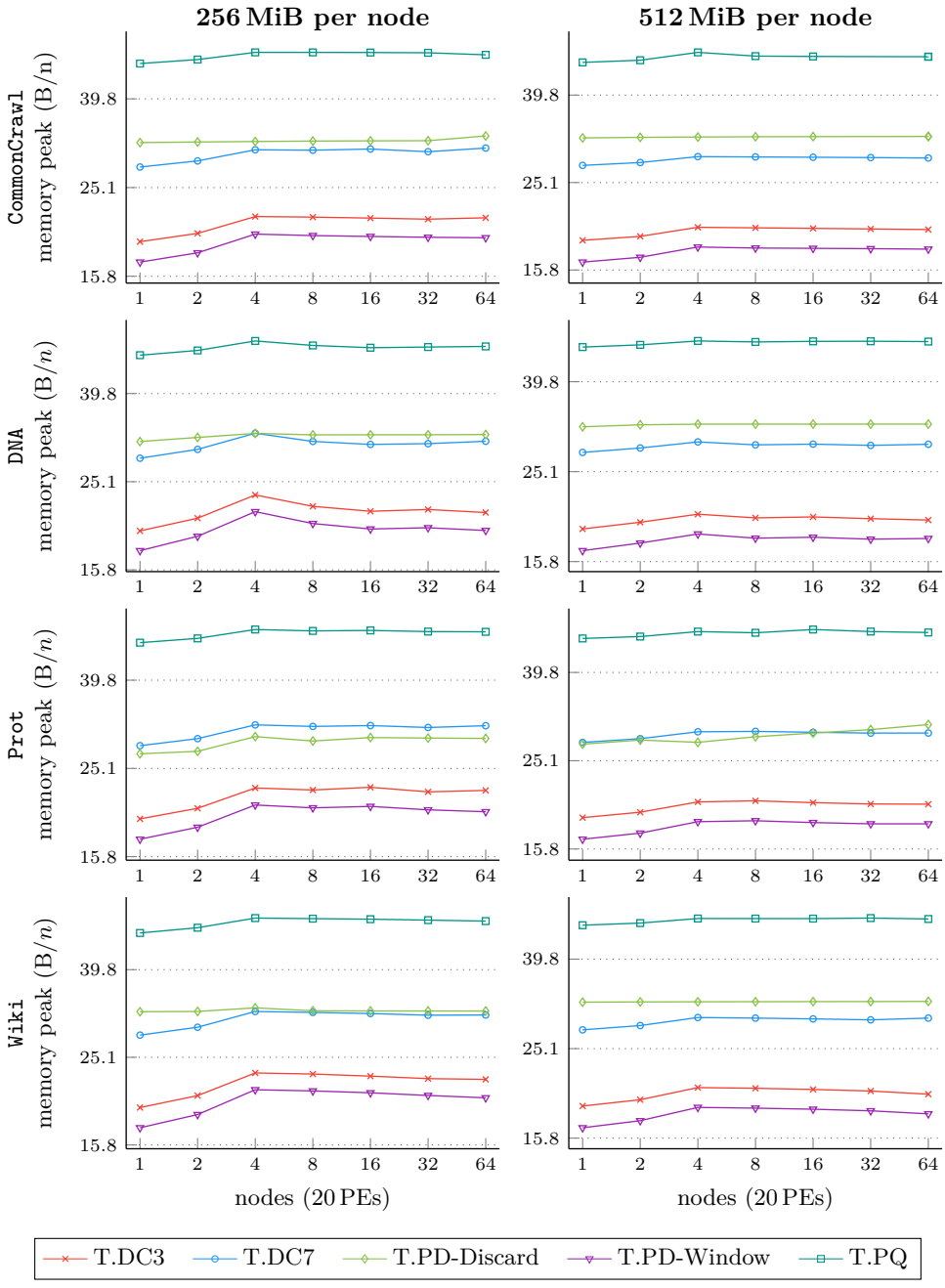


Figure 5.7. Memory peak of distributed suffix sorting algorithms (Thrill).

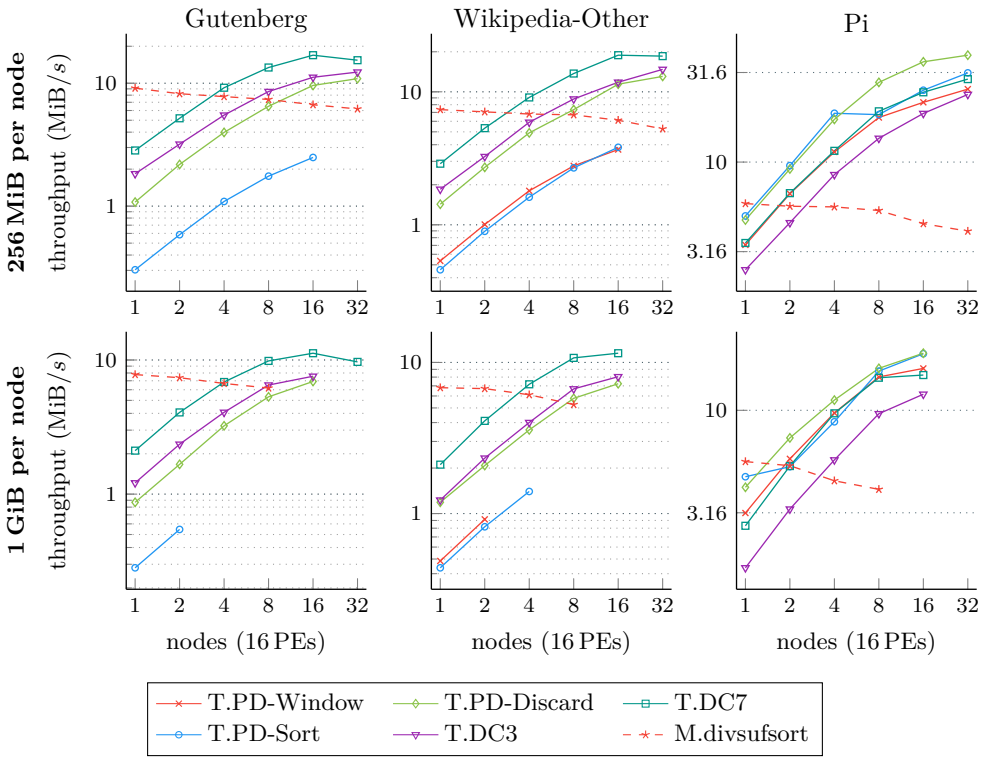


Figure 5.8. Weak scaling plots of distributed and of the fastest non-distributed suffix sorters run on one host with the same input size.

Evaluating Thrill on Different Hardware

As mentioned earlier, the experimental evaluation of the algorithms on LiDO.small nodes shows weak results. The algorithms do not scale well and become slower as soon as the algorithms have to communicate over the network. In Figure 5.8, we show results from another experiment that we describe in detail in [Bin+18]. Here, we conducted the experiments on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) using *i3.4xlarge* instances. Each node has 16 Intel Xeon E5-2686 v4 Broadwell vCPUs with 2.30 GHz clock speed, 122 GB RAM, and 2×1.9 TB Non-Volatile Memory Express (NVMe) SSDs. All experiments were run with the Thrill master branch version from January 19th, 2018, compiled with `g++` 5.4.1 an Ubuntu Linux 16.04 "xenial" using Linux 4.4.0-1052-aws.

We limited the available RAM on each host to 8 GB using the kernel option `mem=8G`. This leaves about 7 GB for Thrill, since the kernel reserves itself a portion. This limitation is extreme, but demonstrates that Thrill can efficiently utilize disk space. For the comparison with the sequential DivSufSort we removed the memory limit.

Since we do not have access to the experimental setup anymore, we can only describe our previous results. Here, we used the following inputs:

Gutenberg is a concatenation of all text documents from Project Gutenberg by document id as available in September 2012. These total 23 GiB in size and contain a version of the human genome as a subsequence.

Wikipedia-Other is a 125.6 GiB XML dump of the English Wikipedia date *enwiki-201701*.

Pi are the decimals of π, written as ASCII digits and starting with "3.1415."

In Figure 5.8, it is easy to see that all algorithms scale well. Here, T.DC7 is the fastest algorithm on Gutenberg and Wikipedia-Other. On Pi, however, T.PD-Discard becomes faster when using at least 8 nodes. The other difference cover algorithm T.PD3 is the second fastest algorithm on all inputs but Pi. There, T.DC7 is still faster. The other prefix doubling versions (T.PD-Window and T.PD-Sort, which is the same as the discarding algorithm without the discarding) are the slowest two algorithms on Gutenberg and Without-Other. On Pi, they are faster than T.DC3.

Now, we consider the COST (Section 1.3.3) of the parallelization. To this end, we use DivSufSort (Section 4.2)—the fastest sequential suffix sorting algorithm. Here, DivSufSort gets the same input as our distributed suffix sorting algorithms. However, since DivSufSort requires $9n$ bytes working space, it can only compute the suffix array for inputs up to 8 GiB. The fastest algorithm T.DC7, requires 4 nodes to be faster than DivSufSort on Gutenberg and Wikipedia-Other. On Pi, T.DC7 is faster when we use two or more nodes. In general, the COST is relatively high in this setting.

We cannot determine the reason for the behavior of the algorithms on LiDO.small nodes. However, compared with the results described above (even though the inputs are different) the throughput on one node is very high. All algorithms on one LiDO.small node achieve roughly a three times higher throughput than in this setup. Therefore, we assume that it must be a combination of the initial high throughput and the network.

5.6.2 Evaluation of Distributed Suffix Sorting using MPI

Now, we evaluate our distributed suffix sorting algorithms implemented using MPI. Here, we compare the two algorithms that we presented in this dissertation: Our distributed prefix doubling algorithms that uses discarding is denoted by *MPI.PD-Discard* (Section 5.2.2) and our distributed suffix sorting algorithm based on DivSufSort is denoted by *MPI.divsufsort* (Section 5.4).

We compare both algorithms with the state-of-the-art distributed suffix sorting algorithm *MPI.psac* [FA15] that is based on prefix doubling and uses sorting to compute the new ranks. Instead of discarding, already sorted intervals of the suffix array are skipped. We also include difference cover implementations by Bingmann [Bin12] (*MPI.DC3*, *MPI.DC7*, and *MPI.DC13*). However, those implementations can handle at most inputs of size 4 GiB as they use 32-bit indices.

We are also aware of different suffix array construction algorithms that are part of *cloudSACA* [Abd+14; Met+16]. However, these algorithms cannot compute the suffix array for inputs of the size that we are considering due to their high memory requirements, which was also shown by Flick and Aluru [FA15]. Therefore, we omit cloudSACA in this evaluation.

Construction Time. We show the algorithm's throughput in our weak scaling experiment in Figure 5.9. Here, we used 512 MiB, 1024 MiB, and 1536 MiB per node as input, e. g., when using 8 nodes, we compute the suffix array for inputs of size 4 GiB, 8 GiB and 12 GiB.

First, we look at the difference cover algorithms, because they are not able to compute the suffix array for all input sizes. We see MPI.DC13 is always faster than MPI.DC7, and that MPI.DC7 is always faster than MPI.DC3. On `CommonCrawl` and `Prot`, MPI.DC13 is the fastest algorithm. However, due to the memory requirements (which we describe in the next section) larger inputs cannot be handled by this algorithm. In general, the difference cover algorithms scale well and have a high throughput (at least MPI.DC7 and MPI.DC13) compared with the other algorithms.

Next, we compare the other three algorithms. Here, MPI.divsufsort and MPI.PD-Discard are the only two algorithms that can handle inputs of size 1536 MiB per node. All other algorithms require to much memory to compute the suffix array for these inputs (as we show in the next section). Comparing MPI.divsufsort and MPI.PD-Discard, we see that on fewer nodes MPI.divsufsort is faster, but as soon as we use 16 or more nodes, MPI.DP-Discard becomes faster, i. e., it scales better. MPI.psac scales even better. It is always the fastest when using 64 nodes. Most impressive is the speed of MPI.psac on `DNA`, where it is always the fastest algorithm and up to twice as fast as MPI.PD-Discard.

We also want to mention the throughput of MPI.divsufsort and MPI.PD-Discard on large inputs (1536 MiB per node) on `CommonCrawl` and `Wiki`. Here, the throughput of MPI.divsufsort and MPI.PD-Discard is significantly lower than on all other tested inputs. Our only explanation is that the distributed (string) merge sort algorithms that we use in both become a bottleneck.

Therefore, MPI.psac remain the fastest and best scaling distributed suffix array construction algorithm. However, our new algorithms (MPI.divsufsort and MPI.PD-Discard) can process inputs of sizes that MPI.psac cannot.

Memory Peak. We give the memory peaks of the algorithms in Figure 5.10. Again, let us first look at the difference cover algorithms. As expected, their memory peak depends solely on the size of the difference cover. Hence, MPI.DC13 requires more memory than MPI.DC7, and MPI.DC7 requires more memory than MPI.DC3. MPI.DC7 has nearly the same memory peak as MPI.psac. The memory peak of MPI.psac is the reason why the algorithm cannot handle larger inputs. MPI.psac requires roughly 63 Byte per character of the input. On `DNA`, it requires only 54 Byte per character of the input. This may be the reason for its high throughput on this input.

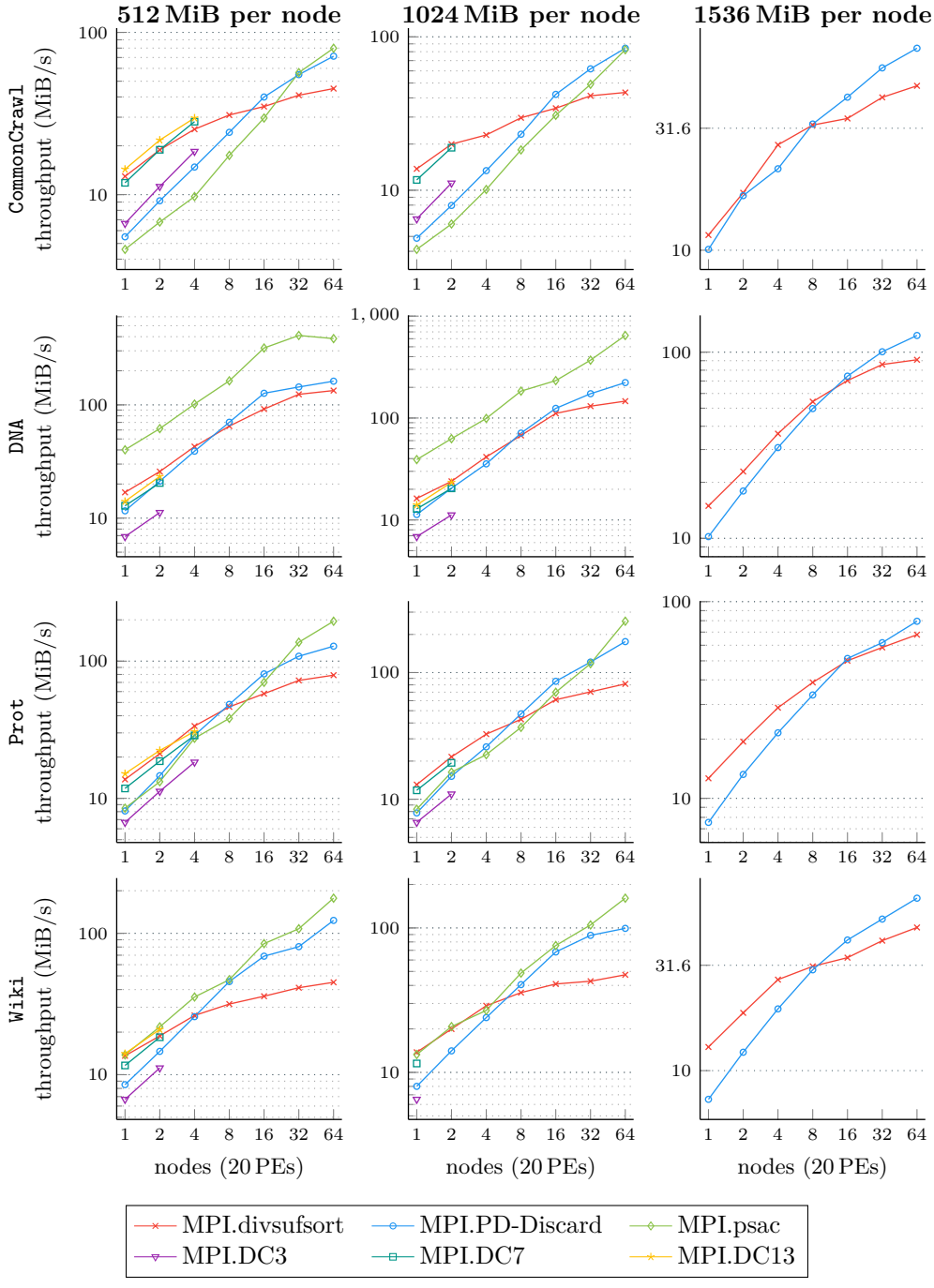


Figure 5.9. Throughput of distributed suffix sorting algorithms (MPI).

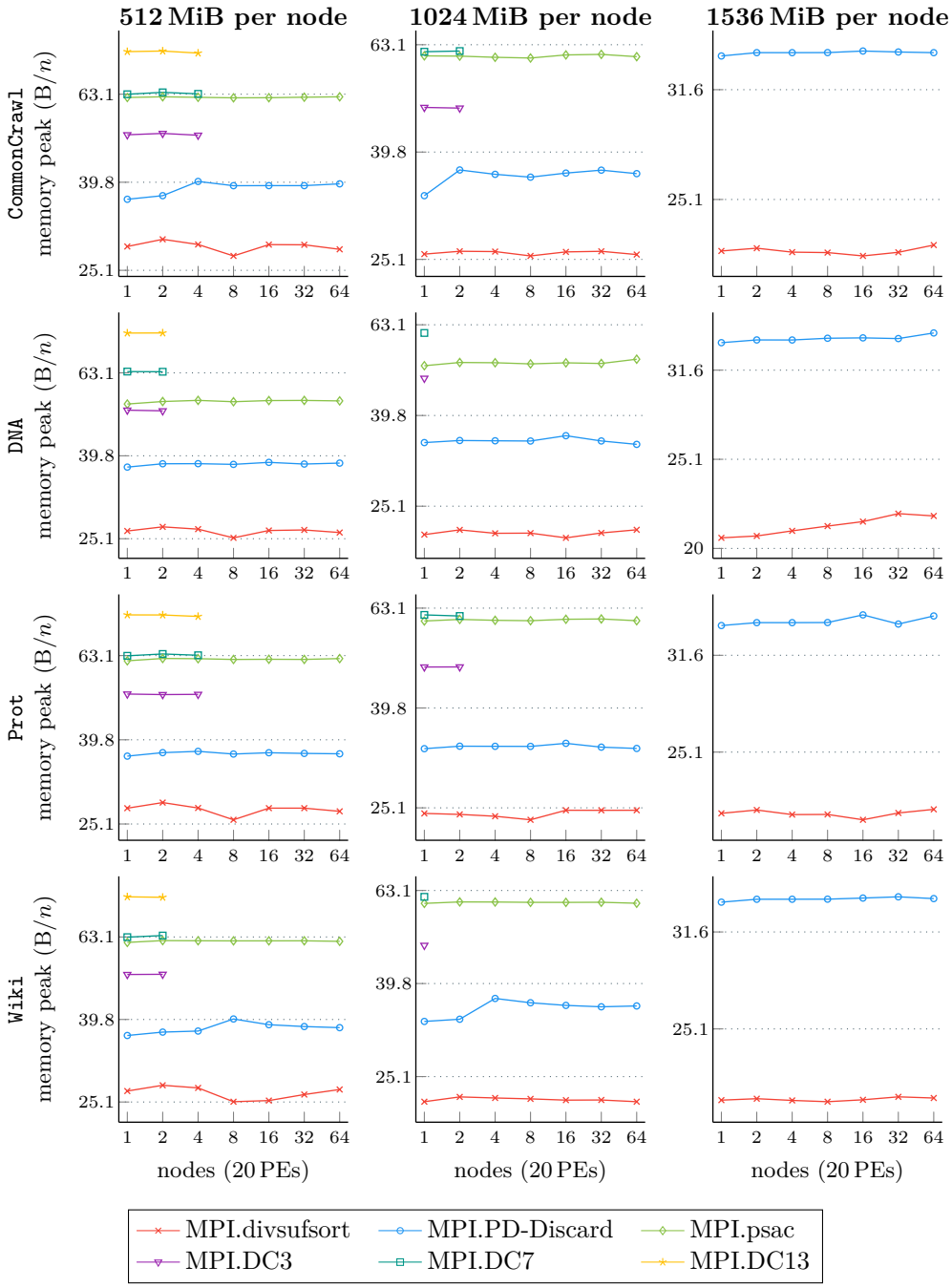


Figure 5.10. Memory peak of distributed suffix sorting algorithms (MPI).

Nevertheless, our two new algorithms require significantly less memory during the construction of the suffix array. MPI.PD-Discard require roughly 39 Bytes per character of the input. The difference between MPI.PD-Discard and MPI.psac is that we use 40-bit integers, whereas MPI.psac uses 64-bit integers. This perfectly explains the difference in the memory peak of both algorithms, as $\frac{63}{8} \cdot 5 = 39.375$. It should be noted that we were not able to easily make MPI.psac work with 40-bit integer.

Our second new algorithm requires even less memory. Here, it is not just the usage of 40-bit integers but the design of the algorithm that results in the low memory requirements. MPI.divsufsort requires slightly more than 25 Bytes per character of the input and is the most memory efficient distributed suffix array construction algorithm.

Overall, this gives us a good trade-off between speed and memory peak. MPI.psac is the fastest distributed suffix sorting algorithm (especially running on more than 32 nodes) but also requires the most memory. Our distributed prefix doubling algorithm (MPI.PD-Discard) requires less memory, but it is slower than MPI.psac. On some instances significantly slower. Finally, MPI.divsufsort is the most memory efficient algorithm. However, it does not scale well and is even slower than MPI.PD-Discard on more than 16 nodes. When using less nodes MPI.divsufsort is faster.

COST of Parallelization. Finally, we consider the COST of parallelization (Section 1.3.3) of the distributed string sorting algorithms. To this end, we conducted a strong scaling experiment. We use 512 MiB, 1024 MiB, and 1536 MiB as input for the distributed suffix sorting algorithm and also for the sequential DivSufSort. DivSufSort is the fastest sequential suffix sorting algorithm, which we have shown in Section 4.1.

We show the throughput of the algorithms in this strong scaling experiment in Figure 5.11. In this experiment, we see MPI.psac's speed on `DNA` again. Here, it has a COST of three on `DNA`, which is not shown in the figure. Because, for a better overview, we only show the results for 5 to 20 processing elements, which is sufficient to show the COST of most algorithms.

In general, the COST on `CommonCrawl` is high. MPI.PD-Discard and MPI.psac have a cost of 30 and 35, respectively. Here, MPI.divsufsort has a COST of 15. On `Prot` and `Wiki`, MPI.PD-Discard has the worst COST, whereas MPI.divsufsort has a better COST of at most 8. On the former MPI.psac has worse COST than MPI.divsufsort (at most 9), and on the latter it has better COST of at most 7.

While there is no algorithm that has the best COST on all inputs, overall, the COSTs are reasonable. Most algorithms achieve a higher throughput than the best sequential suffix sorting algorithm (seq.divsufsort) using at most 20 processing elements (one node in our setup). Only on `DNA`, MPI.DC3, MPI.PD-Discard, and MPI.psac require more than one node to achieve a higher throughput. Hence, we can conclude that the algorithms do not only scale well due to a bad sequential running time of the distributed algorithm.

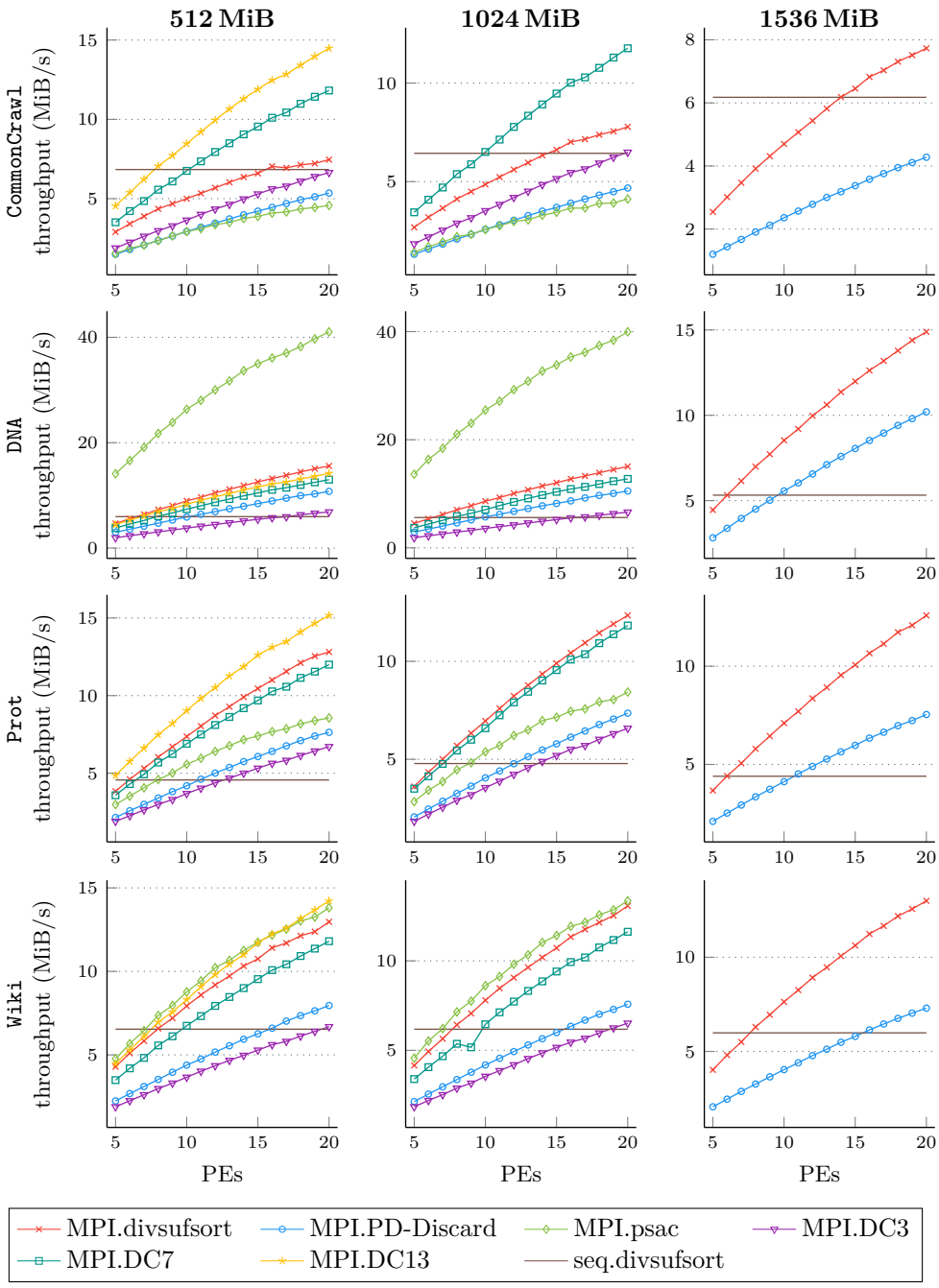


Figure 5.11. Throughput of distributed suffix sorting algorithms (MPI) and the fastest sequential suffix sorting algorithm seq.divsufsort.

Table 5.1. Average length and standard deviation of those lengths of the $C^{+\triangleright}$-ending substrings used in the experiments.

input	average length	standard deviation
`CommonCrawl`	6.001	0.913
`DNA`	6.652	1.583
`Prot`	6.124	0.661
`Wiki`	6.387	1.493

5.6.3 Evaluation of Distributed String Sorting

In Section 5.5, we describe a distributed string sorting algorithm that we use in MPI.divsufsort to sort the $C^{+\triangleright}$-ending substrings. In this section, we present an experimental evaluation of this algorithm. To the best of the author's knowledge, this was the first practical implementation of a distributed string sorting algorithm that we denote by *DSS*. The only other work on distributed string sorting that the author is aware of is presented by Bingmann et al. [Bin+20] (based on Schimek's [Sch19] Master's thesis). They present distributed string sorting algorithms (and different configurations of those): (i) *hQuick* is a multi-key Quicksort, (ii) *PDMS* is based on prefix doubling and merge sort).

Sorting $C^{+\triangleright}$-Ending Substrings in Practice. Our distributed string sorting algorithm DSS is based on a merge and sample sort hybrid. Hence, we can use any sequential string sorting algorithm to sort the strings locally. Since we use a merge sort, we can use any sequential string sorting algorithm to sort the strings locally, before we distribute and merge them. We tested over 130 implementations from Bingmann et al. [Bin+17] and Kärkkäinen and Rantala [KR08], but only show running times for the choices where our DSS is fastest on average. We show results for MSD radix sort, burstsort, multi-key Quicksort, and string sample sort that we denote by *DSS.MSD*, *DSS.BS*, *DSS.MKQS*, and *DSS.SSS*, respectively. To be specific, we use `bingmann_msd_CI3`, `burstsort_sampling_vector`, `multikey_cache8`, and `bingmann_sample_sortBTCUI`. A framework to test *all* sequential string sorting algorithms is part of the code that we provide.

In Figure 5.12, we show the throughput of the of the distributed string sorting algorithms mentioned above. DSS has a similar throughput on all inputs (and input sizes) independent from the used local sorting algorithm. However, DSS.MSD is slightly faster than all other DSS variants on `CommonCrawl`, `Prot`, and `Wiki`. On these instances DSS.BS is the slowest of our DSS variants. Therefore, we chose DSS.MSD as distributed string sorting algorithm in MPI.divsufsort.

Comparing DSS.MSD with the new distributed string sorting algorithms by Bingmann et al. [Bin+20], we see that hQuick is slower on all instances (where it finished sorting, which is not the case for more then eight processing elements). However,

PDMS are significantly faster than DSS.MSD on all inputs and input sizes. We cannot explain the behaviour of PDMS in the large tests on `DNA` and `Prot`. Nevertheless, it scales better than all other tested algorithms. Therefore, PDMS is the fastest distributed string sorting algorithm.

Sorting other Types of Strings. As seen above, the distributed string sorting algorithms MS and PDMS outperform our implementation on all instances tested in this dissertation. A result that coincides with the findings of Bingmann et al. Therefore, we do not conduct further experiments as it has already been shown that *PDMS* is several times faster than our fastest distributed string sorting algorithm [Bin+20]. All their other algorithms but hQuick are also at least as fast as our fastest one.

5.7 CONCLUSION AND FUTURE WORK

First, we presented five different algorithms that we implemented using Thrill. The results we achieved using our experimental setup were underwhelming. Not only do the algorithm not scale well, they become slower when using more than one node. This is a behavior that we cannot explain. We also showed results from other experiments that were conducted on different (cheaper commodity) hardware. Here, the same algorithms scale well.

Next, we presented two different distributed suffix sorting algorithms that we implemented using MPI. These algorithms are of similar speed compared with the fastest distributed suffix sorting algorithm (MPI.psac). However, the big advantage of our algorithms is that they are very memory efficient. This allows us to compute the suffix array for larger inputs on the same hardware.

Finally, we showed the first practical distributed string sorting algorithm. We tested a multitude of sequential string sorting algorithms to determine the fastest local string sorting algorithm for our setting, i. e., sorting $C^{+\triangleright}$-ending substring. However, recently more efficient string sorting algorithms have been presented by Bingmann et al. [Bin+20], making ours obsolete, as they are always worse.

Future Work. Currently, there is only one distributed LCP array construction algorithm. This algorithm is part of MPI.psac [FA15] and computes the LCP array while computing the suffix array. In main memory, it is faster and more memory efficient to compute the suffix array first and the LCP array based on the suffix array afterwards, see Section 4.4. It remains open if computing both arrays successively is also a better approach in distributed memory.

Also, the distributed string sorting algorithms by Bingmann et al. [Bin+20] can be used as distributed string sorter in our distributed suffix sorting algorithm, which should improve the throughput of the However, we do not expect the resulting algorithm to be faster than the ones based on prefix doubling. This leads to the further questions: Are there a other techniques for distributed suffix sorting that can be used to efficiently compute the suffix array in distributed memory? Even if those techniques are not optimal theoretically or not working well in main memory?

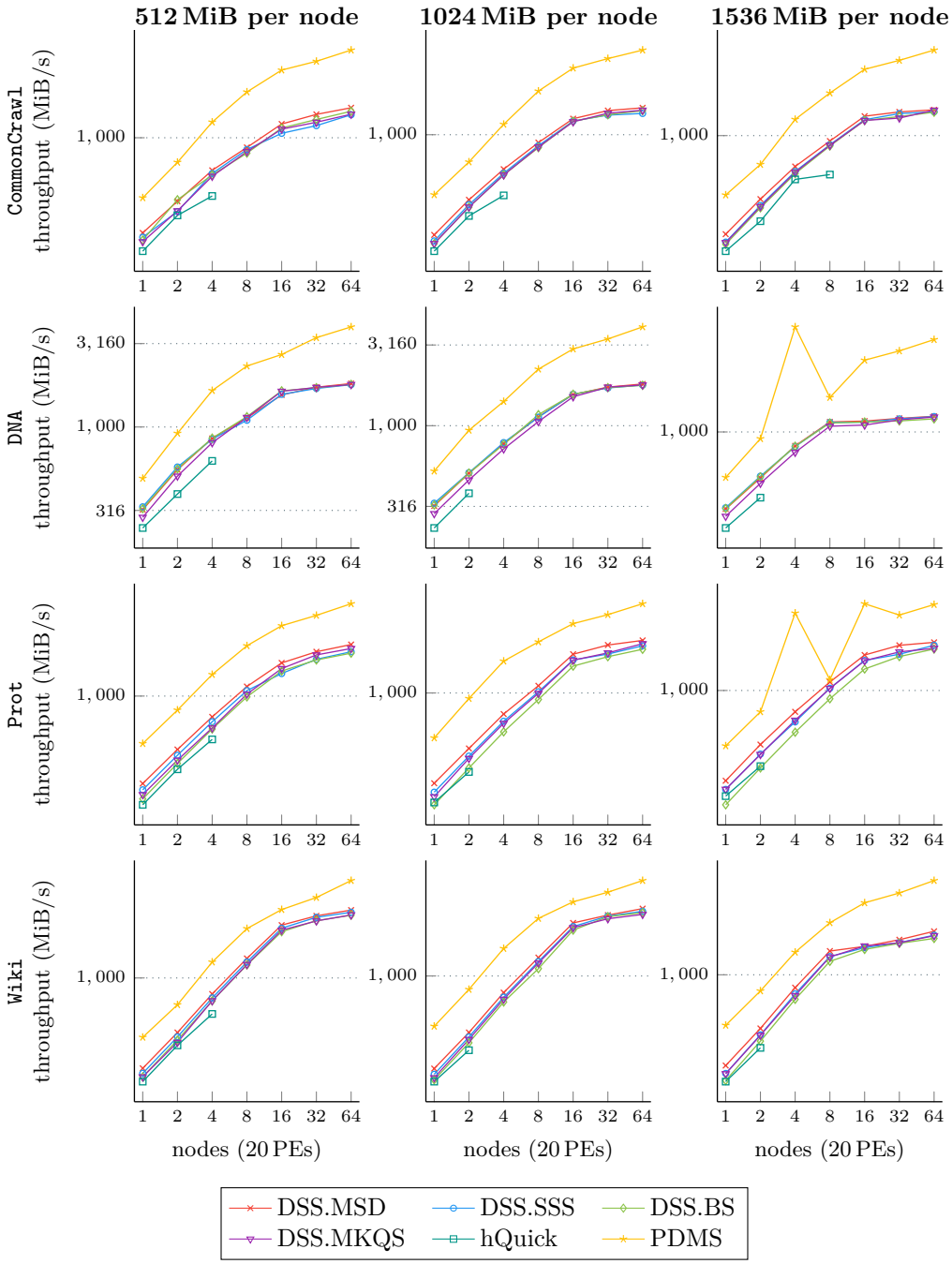


Figure 5.12. Throughput of distributed string sorting algorithms for the $C^{+\triangleright}$-ending substrings with logarithmic axes.

CHAPTER 6

THE DISTRIBUTED PATRICIA TRIE

In this chapter, we present the distributed Patricia trie, a distributed full-text index build on top of the suffix array consisting of two levels. We use the distributed Patricia trie to answer three types of pattern matching queries: *existential* queries (does the pattern occur in the text), *counting* queries (how often does the pattern occur in the text), and *enumeration* queries (where in the text does the pattern occur). We also use *succinct data structures* to represent the tree structures, which reduces the memory requirements of the distributed Patricia trie.

This chapter is based in the publication [Fis+17]. We now give a brief overview of other full-text indices that are related to our work, in Section 6.1. Next, in Section 6.2, we introduce notations that we use in the rest of this chapter. Most notably, we introduce *succinct data structures*, i. e., data structures that require space close to information-theoretic lower bound. Then, in Section 6.3.1, we present the construction algorithm for the distributed Patricia trie, and in Section 6.3.2, we show how to use it to answer queries. Finally, in Section 6.4 we show an experimental evaluation of the distributed Patricia trie.

6.1 RELATED WORK

First, we mention different distributed text indices. Arroyuelo et al. [Arr+14] consider a large variety of distributed suffix array data structures with various trade-offs between replication, number of remote data accesses, and load balance. However, they always use explicit binary search in the suffix array, leading to logarithmically higher costs than our approach. Our index can be viewed as an improvement of the *global* approach presented in [Arr+14], where we add two levels of tries that allow us to answer each query exchanging only a constant number of messages. We show that our index scales better especially for larger text sizes and a larger number of processing elements.

Recently, Flick and Aluru [FA19] present a two-level distributed full-text index that requires the suffix array and the LCP array. The main difference of their approach is that they (1) use a look-up table to match the first few characters, (2) a dynamic top-level trie that balances the subtries, and (3) a space efficient representation of the local subtries.

There exists also theoretical work by Ferragina and Luccio [FL99] that discusses a distributed Patricia trie. Their approach is only good when answering *long* queries, for example, existential queries "does the pattern occur in the text?" of length $m \geq p$, where p is the number of processing elements and m the length of the pattern. Those queries can be answered optimally with respect to computation and communication.

A complementary (theoretical) approach is described by Mäkinen et al. [Mäk+04] and is good for short patterns. Using backwards search, a query can be answered using m communication steps. The problem is that at most σ processing elements can be used and, in the worst case, some processing elements might need space $\Omega(n)$.

Russo et al. [Rus+10] (theoretically) describe distributed compressed indices. Their approach partitions the text between the processing elements and works with local indices. The consequence is that queries have to be processed on every processing element, contrary to our goal to have total work independent of p.

There are several results that assume that the input text is replicated over all processing elements (e. g. the distributed suffix tree by Clifford [Cli05]). This makes index construction and search much easier but severely limits scalability, therefore, we do not consider this approach any further.

While not a distributed text index, the reduced-space on-disk suffix array (RoSA) index by Gog et al. [Gog+14b] is a two-leveled index that also makes use of blind tries. Most notably, the RoSA index is build on top of the Burrows–Wheeler transform [BW94] of the text, which is a transformation of the text that is (1) easy to compress and (2) can be reversed to obtain the original text.

Also, multi-level full-text indices have been considered for external memory. The *String B-Tree* by Ferragina and Grossi [FG99] utilizes Patricia tries at each level to reduce the I/O volume.

Similar to our distributed Patricia trie, many distributed indices rely on suffix arrays and sometimes also on the corresponding LCP array. As we described in great detail in Chapter 5, suffix array construction in distributed memory is well researched. However, Flick and Aluru [FA15] give the only distributed algorithm for computing both the suffix array and the LCP array; their approach is within a factor of $\mathcal{O}(\log n)$ from the optimal. Thus, we can use the suffix array and the LCP array as the starting point for our distributed index construction.

6.2 PRELIMINARIES

In this section, we first introduces tries in Section 6.2.2, which we use in both levels of our distributed Patricia trie. Then, in Section 6.2.1, we give different succinct representations for trees, i. e., we present how we can represent a tree with n nodes using only $2n$ bits. Using additional $o(n)$ bits of space allows us to navigate in the succinct trees.

6.2.1 Tries

We build an index on top of the suffix array. This index is a trie, a special case of trees. Given a labeled tree $G = \langle V, E \rangle$ with root $r \in V$, we denote the label of a node or an edge $x \in V \cup E$ by $\mathit{label}(x)$ and the concatenation of all edge labels on the path from the root to any node v by $\mathit{pathlabel}(v)$. The *out-degree* of a node v is denoted by $\delta^+(v)$. The *leaf rank* of a leaf $\ell \in V$ is the number of leaves visited before ℓ in a preorder traversal of the tree.

Let $R = \{R_1, R_2, \ldots, R_k\}$ be a set of strings over the alphabet Σ such that all strings are distinct and no string is the prefix of another string in R. The *trie* of R is an ordered tree with root r, where the edge labels are characters and the leaves represent string numbers from $[1, k]$ such that:

1. for each node $v \in V$, the labels of the outgoing edges $\mathit{label}((v, \cdot)) \in \Sigma$ are distinct,
2. for each string $R_i \in R$, there is a leaf $\ell \in V$ with $R_i = \mathit{pathlabel}(\ell)$ and $\mathit{label}(\ell) = i$, and
3. for each leaf $\ell \in V$ there is a string $R_i \in R$ such that $R_i = \mathit{pathlabel}(\ell)$ and $\mathit{label}(\ell) = i$.

The *compressed trie* is a trie where each path $e_1, e_2, \ldots, e_\ell$ with $\ell > 1$ consisting only of nodes with out-degree 1 is replaced by a single edge e such that $\mathit{label}(e) = \mathit{label}(e_1)\,\mathit{label}(e_2)\ldots\mathit{label}(e_\ell)$. Still, all outgoing edges of a node v start with a different character. The *string depth* of a node v is $\mathit{sd}(v) := |\,\mathit{pathlabel}(v)\,|$, i. e., the length of the longest common prefix of all strings represented leaves below v. To find all occurrences of a pattern P in a compressed trie, we start at the root r and follow the edge e such that $\mathit{label}(e) = P[1..|\,\mathit{label}(e)\,|]$. At each node v, the length of the pattern matched up to this point equals $\mathit{sd}(v)$. We then follow the edge e with $\mathit{label}(e) = P[\mathit{sd}(v) + 1..|\,\mathit{label}(e)\,| + \mathit{sd}(v)]$. This process is repeated until we have matched the whole pattern at the edge $(\cdot, v)$. Then, all leaves that are successors of v correspond to strings in R that are prefixed by P. If at any point, there is no edge to follow, the pattern P does not occur in the trie.

The *Patricia trie* (or *blind trie*) [Mor68] of a text T is a compressed trie for all suffixes of T, where each node v just stores the first character and the string depth $\mathit{sd}(v)$. Due to this limitation, finding all occurrences of a pattern requires two steps—a *blind search* followed by a comparison to a substring of T (which has been determined by the blind search):

(BS1) For the blind search, we start at the root and follow the edge matching the pattern at the position corresponding to the string depth, i. e., at a node v we follow the edge e with label $\mathit{label}(e) = P[\mathit{sd}(v)]$. We repeat this until we have reached a node v such that $\mathit{sd}(v) \geq |P|$ or there is no feasible edge to follow. In the first case, we retrieve a prefix of length $|P|$ of a suffix corresponding to any leaf w that is a successor of v and compare that prefix with our pattern P. In the second case (there is no edge to follow) P does not occur in T.

(BS2) Next, we compare P and $T[i..i+|P|-1]$ where i is the label of the leaf w (that has been identified during the blind search). If the strings are equal, then all leaves that are below v correspond to an occurrence of P in T. Otherwise, P does not occur in T.

The Patricia trie can be constructed from the suffix array, LCP array, and the text in linear time, i. e., scanning the suffix array and the LCP array once and considering each entry at most twice. The text is required for the edge labels and each position is accessed at most once. We give a detailed construction algorithm for the distributed Patricia trie in Section 6.3.1. Later, in Section 6.4, we also compare the construction time for the tries needed by our index with the time required for the construction of the suffix array and the LCP array.

6.2.2 Succinct Data Structures

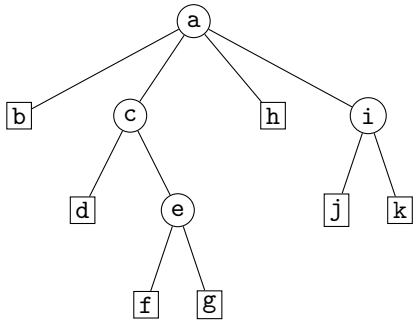
We can represent a tree containing ℓ nodes using a bit vector $\mathsf{BV} \in \{0,1\}^{2\ell}$. The bits represent parentheses; a 1 represents an open parenthesis "(" and a 0 represents a closing parenthesis ")". To navigate in the tree, we require additional operations on the bit vector. If it is clear from context, indicate on which bit vector the operations are conducted. We already defined rank and select queries in Section 2.1. As a small reminder, $\text{rank}_0(i)$ asks for the number of 0's up to position $i-1$ and $\text{select}_0(i)$ returns the position of the i-th 0. The operations work analogously for rank_1 and select_1. Additionally, we need the operation $find\_close(i)$, which gives the position of the matching closing parenthesis for an open parenthesis at position i. All these operations can be answered in constant time [Cla97; Jac89]. In the following, we explain three different succinct representations of trees. An example of these representations is depicted in Figure 6.1.

The first two representations use parenthesis to describe the tree structure. Here, parentheses are used for an easier understanding of the structure. Since there are only have two of those (either an opening or a closing parenthesis) we can represent them as bits in our implementation—but as parentheses in our figures and descriptions.

Balanced Parenthesis

We first start with the *balanced parenthesis* (BP) representation that was introduced by Munro and Raman [MR01]. This succinct representation used parenthesis to represents the tree. It can be constructed by traversing the tree in *preorder*, i. e., we first visit the current node (starting with the root) and then recursively visit all children starting with the first. We add an opening parenthesis "(" to the bit vector whenever we visit a node for the first time and we add a closing parenthesis ")" to the bit vector whenever we visit a node for the last time.

In theory, BP also allows an access of the i-th child in constant time [NS14]. However, in the implementation used by us (see Section 6.4 for more details), BP does not support a direct access to the i-th child. Instead, one has to access the first child of the node at position x (position $x+1$) and then go to the next child ($find\_close(x)+1$)



(a)

	ab cd ef g h ij k
BP	(()(()(())))(()()))
	a bc de fghi jk
DFUDS	((((())(()))(()))))(()))
	ab ch id ejkfg
LOUDS	11110011001100100000

(b)

Figure 6.1. An exemplary tree in (a) where each node has a label and the leaves are marked as boxes. In (b), we show the succinct representation of the tree in all three different representations that we consider in this chapter. There, we also mark the positions where the nodes are represented.

until the i-th child is reached in $\mathcal{O}(i)$ time. In our implementation, the first two representations allow an access of the i-th child in constant time, whereas it takes $\mathcal{O}(i)$ time in BP.

Depth First Unary Degree Sequence

The next succinct tree representation is the *depth first unary degree sequence* (DFUDS) introduced by Benoit et al. [Ben+05]. This representation is also based on parentheses is obtained by traversing the tree in preorder and (like in LOUDS) append $\delta^+(v)$ opening parentheses followed by a closing parenthesis whenever we visit a node v for the first time. To make the sequence balanced, we prepend an opening parenthesis. The position of the i-th child of the node at position x is identified by $find\_close\big(\text{select}_{\text{“)”}}(\text{rank}_{\text{“(”}}(x)+1)+1\big)+1$ in constant time.

Level Ordered Unary Degree Sequence

Finally, we want to mention the *level ordered unary degree sequence* (LOUDS) introduced by Jacobson [Jac89]. When we represent a tree using LOUDS, we actually use bits to represent the tree and do not make use of parentheses. Hence, there is no access to the $find\_close$ operation. A tree that is encoded using LOUDS is represented level-wise, i. e., starting with the root, we visit all nodes v of a level from left to right and add $\delta^+(v)$ 1's followed by a 0 to the bit vector. The position of the i-th child of the node at position x is identified by $\text{select}_0(\text{rank}_1(x)+i-1)+1$ in constant time.

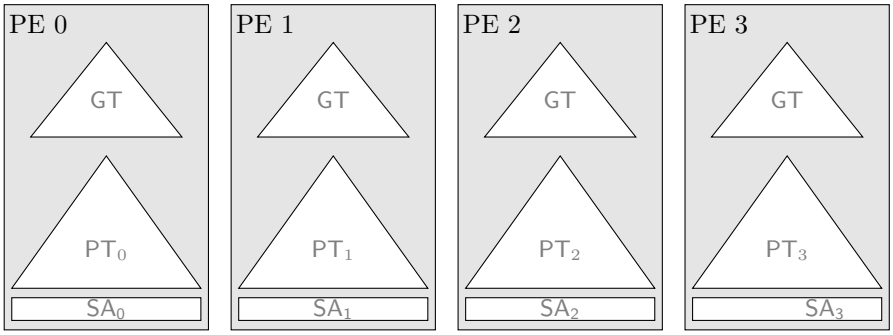


Figure 6.2. Structure of the distributed Patricia trie on four processing elements. GT is the same on all processing elements, whereas PT_i differs for each processing element $i \in [0, 4)$.

6.3 DISTRIBUTED PATRICIA TRIE

Now, we present the construction algorithm for the distributed Patricia trie and show how to answer queries with it. For simplicity, we assume that the size of the text n is divisible by number of processing elements p. Then we distribute the suffix array SA and LCP array LCP in a consecutive fashion, such that the i-th processing element holds $\mathsf{SA}_i \coloneqq \mathsf{SA}[i \cdot n/p..(i+1)n/p)$ and $\mathsf{LCP}_i \coloneqq \mathsf{LCP}[i \cdot n/p..(i+1)n/p)$. In addition, each processing element holds a part of the text as described in the next section.

Our proposed data structure, the *distributed Patricia trie* (DPT), is a two level index consisting of an index GT for query distribution (first level) and several indices PT_i that can find all occurrences of a pattern P that starts at text positions held by the local SA_i on processing element i (second level). In this case we say that the the i-th processing element is *responsible* for P. The index GT is replicated at every processing element. This allows queries to arrive at arbitrary processing elements and then to be sent to the responsible processing elements in the next step. There, the query is processed utilizing PT_i. This index is unique for each processing element, see also Figure 6.2.

6.3.1 Construction of the Distributed Patricia Trie

In this section we show how to construct the distributed Patricia trie in linear time. We start with the construction of the local Patricia tries (PT) as we use the information about their smallest and greatest element for the construction of the global tries (GT).

Local Tries

The construction is the same at each processing element, even though the local tries differ. Our construction algorithm is the extension of an algorithm to compute the *suffix tree*, i. e., a compressed trie of all suffixes of a text T. We modified the suffix insertion algorithm presented in [Mäk+15, p. 143] such that the Patricia trie can be

constructed by scanning the suffix array and the LCP array from left to right. The pathlabel of the rightmost path in a (Patricia) trie is the lexicographically largest pathlabel in the trie. Since all suffixes in the suffix array are in lexicographical order, each suffix that is added to the Patricia trie is lexicographical greater that all previously inserted suffixes and will form the new rightmost path. Therefore, at each point of time during the construction, only nodes on the rightmost path can be changed. All other nodes are considered as *final*. The inner nodes on the rightmost path, i. e., the nodes that can still be changed, are kept on a stack. Since we compute a Patricia trie, each node v knows its string depth $sd(v)$.

Initially, we have a stack containing a node with string depth 0 and no children. We start by adding the first inner node v, with $sd(v) = \mathsf{LCP}[2]$, two children (the left child represents $\mathsf{SA}[1]$ and the right child represents $\mathsf{SA}[2]$) with edge labels $T[\mathsf{SA}[1]+\mathsf{LCP}[2]]$ and $T[\mathsf{SA}[2] + \mathsf{LCP}[2]]$, resp. If $sd(v) = 0$, the node replaces the initial one that has been on the stack. Otherwise, v will be a child of the initially created node. We now continue to scan the SA and LCP-array from left to right. Whenever we read a new position i in the LCP-array, we remove nodes from the stack until the node v on top of the stack has $sd(v) \leq \mathsf{LCP}[i]$. If $sd(v) < \mathsf{LCP}[i]$, we create a new inner node w with $sd(w) = \mathsf{LCP}[i]$, i. e., we *branch below* node v. The left child of w (edge label $T[\mathsf{SA}[i-1] + \mathsf{LCP}[i]]$) is the former rightmost child of v, and the right child of w is a new leaf referring to $\mathsf{SA}[i]$ and has edge label $T[\mathsf{SA}[i] + \mathsf{LCP}[i]]$. Next, w becomes the new rightmost child of v and is put on the stack. If $sd(v) = \mathsf{LCP}[i]$, v just gets a new rightmost child (edge label $T[\mathsf{SA}[i] + \mathsf{LCP}[i]]$) referring to $\mathsf{SA}[i]$, i. e., v gets a new *leaf*. Following these operations, we can compute each local PT in $\mathcal{O}(n/p)$ time.

With respect to practical application, we also want to construct succinct representations of the tries. It is possible to compute a succinct representation using its pointer based representation. Using the approach described above, we can also compute a succinct trie representation directly, i. e., reducing the required memory peak for the construction.

We compute the DFUDS representation of a trie by storing all final nodes and their subtrees in DFUDS representation. Whenever we remove a node from the stack, we add it and its subtree at the end of the already computed DFUDS representation of the previously removed final nodes. This is possible because we construct the trie in the same order as a depth first search traversal visits all nodes (which is the order in which the nodes are represented in DFUDS).

Up to now, we have simply named the characters that correspond to the edge labels. Since all local PTs are on different processing elements, we cannot assure that the text position required for an edge label is locally available. We have to retrieve all edge labels during one communication phase. The number of characters stored at each processing element is $\Theta(n/p)$. During the construction of the local Patricia trie PT_i, we scan the arrays SA_i and LCP_i to determine the first mismatching text positions of two lexicographically consecutive suffixes. These characters will then be used for the edge labels later on. If we create a new leaf, we only require *one* character label.

For a simpler and more realistic analysis of the costs for constructing our local indexes, we assume that all mismatching characters are stored at the same processing

element where the corresponding suffix starts, i. e., we assume that $T[\mathsf{SA}[i]..\mathsf{SA}[i] + \max(\mathsf{LCP}[i], \mathsf{LCP}[i+1])]$ is stored on one processing element for all $1 \leq i \leq n$. This is usually the case if the text T is composed of a number of smaller documents such that all documents reside on a *single* processing element, but all processing elements still have $\Theta(n/p)$ characters. (If this is not the case, one could still replicate parts of the text on each processing element such that the processing elements hold overlapping parts of the text.) Under this assumption, each processing element needs to *send* $\mathcal{O}(n/p)$ characters as edge labels. Further, each processing element also *receives* at most $\mathcal{O}(n/p)$ characters, as the local Patricia trie has less than $2n/p$ edges. Finally, we note that the construction takes one superstep (construct the tree and store the text positions, then retrieve the characters at those positions).

Lemma 6.1. *Given that all mismatching characters are stored at the same processing element where the corresponding suffix starts, the* SA*, and the* LCP *array, constructing the Patricia tries costs* $\mathcal{O}(\frac{n}{p} + G\frac{n}{p} + L)$*. Each* PT_i *requires* $\mathcal{O}(\frac{n}{p}(\lg n + \lg \sigma))$ *bits of space in addition to the size of the tree structure.*

Global Trie

Next, we consider the construction of the *global trie* (GT), which allows us to distribute queries without accessing the text. GT is the same at every processing element, which allows arbitrary processing elements to initially process any query. To identify all processing elements that are responsible for a pattern, we require the smallest and largest suffix that is represented by each processing element and their LCP values.

Using the set of suffixes $\mathcal{S} = \{T[\mathsf{SA}_1[1]..n), \ldots, T[\mathsf{SA}_1[n/p]..n), \ldots, T[\mathsf{SA}_p[1]..n), \ldots, T[\mathsf{SA}_p[n/p]..n)\}$ to construct GT, we can use the following observation to identify all processing elements that are responsible for a pattern.

Observation 6.1. *Processing element* i *is responsible for a pattern* P *if and only if* $T[\mathsf{SA}_i[1]..\mathsf{SA}_i[1] + |P|) \leq P$ *and* $P \leq T[\mathsf{SA}_i[\frac{n}{p}]..\mathsf{SA}_i[\frac{n}{p}] + |P|)$*.*

Obviously, there can be patterns for which multiple processing elements are responsible. Depending on the type of query, we need to use the second level index of at most two processing elements to answer a query. Communication with more processing elements may be necessary, see Section 6.3.2 for more details.

The global trie can be constructed similar to the local Patricia trie construction described above. The suffixes required for the construction are known (all suffixes in $\mathcal{S}$). We still require the size of the longest common prefixes of those suffixes. For two lexicographically consecutive suffixes the size is in the LCP array. The size of the longest common prefix of the other suffixes is the string depth of the root of the corresponding local PT. We can propagate all these values during one communication phase, where each processing element sends the two text positions and LCP values to all other nodes, sending $\mathcal{O}(p)$ messages of constant size. At the end of the phase each processing element has a temporary SA and a temporary LCP array each of size $2p$. Using these arrays we use the algorithm described above, only handling edge labels differently.

The task of the global trie GT is to distinguish all elements in $\mathcal{S}$ without accessing T. Therefore, the edge labels may consist of more than one character. The first character of an edge (v, w) is the same character we would store if we constructed a Patricia trie. Let the text position of this character be i. Instead of storing only this character and the string depth, we now need to store the substring $T[i..i + sd(w)]$ as the edge label of (v, w). Hence, it is not necessary to store the string depth at the nodes. In addition, we construct the trie with respect to a maximum pattern size $|P|_{\max}$. We can usually assume that $|P|_{\max}$ is constant (chosen during construction) such that the size $|P|$ of each pattern P is at most $|P|_{\max}$. Thus, the total size of all edge labels is bounded by $2p|P|_{\max}$. During the construction, we store references to the edge labels, i. e., the text position and length. Therefore, at each processing element it is known which substrings need to be communicated (as edge labels). The edge labels are distributed among the processing elements in two supersteps. First, each processing element sends an equal amount of different labels to each processing element. The cost for this superstep (including the construction of the tree structure) is $\mathcal{O}(p + Gp|P|_{\max} + L)$. In the next superstep, each processing element distributes the received labels to each other processing element, costing $\mathcal{O}(Gp|P|_{\max} + L)$.

To prevent that a requested substring spans over more than one processing element we pad the locally stored text with the next $|P|_{\max}$ characters. Since we build the trie for $2c$ substrings, this requires $\mathcal{O}(p)$ time, which leads to the following Lemma.

Lemma 6.2. *Given the suffix array and LCP array, constructing the global trie costs $\mathcal{O}(p + Gp|P|_{\max} + L)$. The trie requires $\mathcal{O}(p|P|_{\max} \lg \sigma)$ bits of space in addition to the space required by the tree structure.*

Reducing the Memory Overhead. Now we show how we can reduce the memory overhead by increasing the number of supersteps required during construction. The whole index is kept in main memory, therefore, we want the overhead during the construction to be as small as possible. First, note that we can stream the suffix array and LCP array, since we just need to scan them once for the construction. Second, we look at the size of the indices, as usually a text position requires more space than a character. Since we need characters but obtain text positions, we need to store them until the next communication phase. Usually, $\lg n > \lg \sigma$ (i. e., factor of five to ten in practice), thus the text positions consume more memory than the labels will later on. If we only compute s required text positions during a superstep and then retrieve them, we need $\mathcal{O}(\frac{n}{sp})$ supersteps. Thus, we can decrease the memory overhead by increasing the number of supersteps that are required during the construction. This yields the following space-time trade-off (regarding the maximum amount of memory required during construction).

Corollary 6.1. *Given the suffix array and the LCP array, the cost of constructing the Patricia trie is $\mathcal{O}(\frac{n}{p} + G\frac{n}{p} + L\frac{n}{sp})$ if we only allow $s \lg n$ bits additional space.*

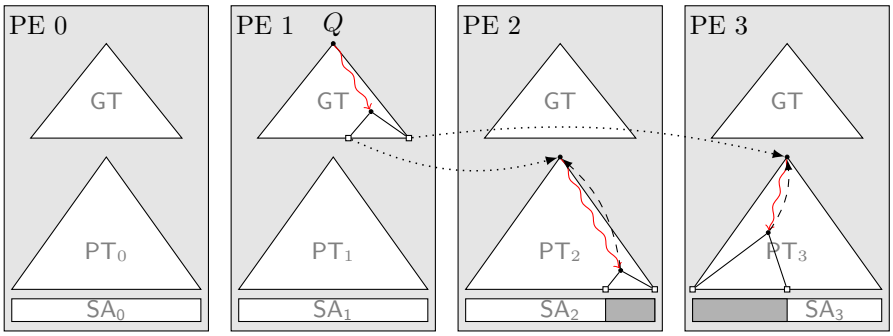


Figure 6.3. Answering a query Q with a distributed Patricia trie on four PEs.

6.3.2 Querying a Distributed Index

The global trie GT is constructed for the set $\mathcal{S} = \{T[\mathsf{SA}_1[1]..n), \ldots, T[\mathsf{SA}_1[n/p]..n), \ldots, T[\mathsf{SA}_p[1]..n), \ldots, T[\mathsf{SA}_p[n/p]..n)\}$ and a maximum pattern length of $m_{\max}$. For any $i \in [1, c]$ the $2i$-th leaf corresponds to the lexicographically smallest suffix and the $2i+1$-th leaf corresponds to the greatest suffix represented by processing element i. Querying GT is different from querying a trie as we do not want to find all occurrences of a pattern P, but want to find all processing elements that represent P. We still follow the edges according to their label and the corresponding position in P until we have matched P at a node u or have a mismatch with the label of an edge (v, w).

In the first case and if v is an internal node, we need to identify the leftmost and rightmost leaves below v. Let k and ℓ be the leaf ranks of those leaves, resp. Then all processing elements j with $j \in [\lfloor \frac{k}{2} \rfloor, \lfloor \frac{\ell}{2} \rfloor]$ contain positions where the pattern occurs, as the processing elements cannot be distinguished by P. If (in the first case) v is a leaf with leaf rank k, then processing element $\lfloor \frac{k}{2} \rfloor$ can be responsible for P. In this case we cannot be sure, as $\mathit{pathlabel}(v)$ may be a prefix of P. Therefore, we send P to processing element $\lfloor \frac{k}{2} \rfloor$ and use the local Patricia trie $\mathsf{PT}_{\lfloor \frac{k}{2} \rfloor}$ to determine whether P occurs.

In the second case (there was a mismatch), P can still occur. Let α and β be the mismatching characters of the label and the pattern, resp. If $\alpha >_{\text{lex}} \beta$ we look at the leftmost leaf below w. If the leaf rank k is even, P does not occur in any processing element, as it is smaller than the lexicographically smallest suffix represented by processing element $\lfloor \frac{k}{2} \rfloor$ and greater than the lexicographically greatest suffix represented by processing element $\lfloor \frac{k}{2} \rfloor - 1$ because otherwise another edge would be followed in the beginning. If the rank is odd, P may occur in processing elements $\lfloor \frac{k}{2} \rfloor$. In the other case ($\alpha <_{\text{lex}} \beta$), we need to get the rightmost leaf below w and check the leaf rank. There may be an occurrence if the leaf rank is even and there cannot be an occurrence if the leaf rank is odd (with the same type of argumentation given before). All processing elements that are responsible for a pattern P form a consecutive interval that we denote by $\mathsf{GT}(P) = [\ell, r]$.

Lemma 6.3. *Given* GT *and a pattern* P*. Let* $\mathsf{GT}(P) = [\ell, r]$*, if* $\ell \neq r$ *then* P *occurs at least once in the processing elements* ℓ *and* r *and* $\frac{n}{p}$ *times in processing elements* j *for all* $j \in (\ell, r)$*.*

Now we take a look at how to answer *pattern matching* queries in the local Patricia tries. First, we look at the processing of a single query. Later, we show how the index can be used to answer a batch of queries.

Pattern Matching Queries.

We now look at three different types of pattern matching queries that we can answer using the distributed Patricia trie.

Existential Queries : Given a pattern P, we want to know whether the pattern P occurs in the text T.

Counting Queries : Given a pattern P, we want to know how often the pattern P occurs in the text T.

Enumeration Queries : Given a pattern P, we want to know all text positions in T where P occurs.

First, we look at an existential query P of length m that arrives at processing element i. We can answer the query in three supersteps (see also Figure 6.3):

(EX1) At processing element i, we identify all processing elements that are responsible for P, i. e., all processing elements j with $j \in \mathsf{GT}(P)$. If $\ell \neq r$ we know that P occurs in T (see Lemma 6.3), else we send P to processing element ℓ.

(EX2) Next, we perform a blind search in PT_ℓ. If the blind search fails, we know that P does not occur in T. Otherwise, the blind search returns a text position q. During the communication phase we retrieve $T[q..q + m - 1]$.

(EX3) Using $T[q..q + m - 1 - 1]$ we can verify the existence of P in T in the third superstep. When a query can be answered at a processing element, we do not send it somewhere else, as the target depends on the application the index is used for.

The cost of an existential query is the following. During the first superstep we identify all processing elements that can answer the query and send it to one processing element costing $\mathcal{O}(t_{\text{trie}}(P) + mG + L)$. We let $t_{\text{trie}}(P)$ denote the time required to search for P in a trie. Depending on the implementation this requires $\mathcal{O}(m \lg \sigma)$ time (binary search) or $\mathcal{O}(m + \lg \lg \sigma)$ time [FG15]. In the second superstep we perform a blind search and retrieve a substring of length m. This costs $\mathcal{O}(t_{\text{trie}}(P) + mG + L)$. During the last superstep we just compare two strings of length m in $\mathcal{O}(m)$ time.

Counting all occurrences of a pattern can be seen as an extension of the existential query and can be answered similarly (requiring four supersteps). Let P be a counting query of length m arriving at processing element i.

(CO1) First, we identify all processing elements that are responsible for P, i. e., all processing elements j with $j \in \mathsf{GT}(P)$. Let all processing elements j with $j \in [\ell, r]$ be responsible for P. If $\ell \neq r$ we know that P occurs in all those processing elements (see Lemma 6.3). During the communication phase we send two queries Q_ℓ and Q_r to processing element ℓ and r, resp. The former asks for the lexicographically smallest occurrence of P in PT_ℓ and the latter asks for the lexicographically largest occurrence of P in PT_r.

(CO2) In the next step, we perform one blind search in PT_ℓ and one blind search in PT_r. If $\ell \neq r$ we know that the blind searches will return two text positions q_ℓ and q_r that are the lexicographically smallest and largest occurrences of P in T. If one of the blind searches fails we know that the processing element is not responsible for P and we can send that there are no occurrences at the processing element. During the communication phase, we retrieve $T[q_\ell..q_\ell + m - 1]$ and $T[q_r..q_r + m - 1]$.

(CO3) Using $T[q_\ell..q_\ell + m - 1]$ and $T[q_r..q_r + m - 1]$ we can verify the existence of P in T (only necessary if $\ell \neq r$) and also find the number of occurrences at processing elements ℓ and r using the leaf ranks. We send the number to processing element i.

(CO4) We know the number of occurrences occ_ℓ and occ_r of P in processing elements ℓ and r, resp. We also know that P has to occur $\frac{n}{p}$ times at each processing element j for $j \in (\ell, r)$. Thus the total number of occurrences of P is $occ_\ell + occ_r + \max(0, r - \ell - 1)\frac{n}{p}$.

The first superstep costs $\mathcal{O}(t_{\text{trie}}(P) + Gm + L)$ as we need to identify the processing elements that are responsible for the pattern and send it to two processing elements. In the second superstep we perform a blind search at two processing elements and retrieve two substrings of length m. This costs $\mathcal{O}(t_{\text{trie}}(P) + Gm + L)$. The third superstep consist of comparing the retrieved substrings with the pattern and send the number of occurrences of the pattern to the processing element where the pattern arrived initially, i. e., processing element i. This costs $\mathcal{O}(m + G + L)$. During the last superstep we need to compute the total number of occurrences at processing element i which costs $\mathcal{O}(1 + L)$.

Last, we consider enumeration queries. Let P be a enumeration query of length m arriving at processing element i. During the first two supersteps answering an enumeration query does not differ from answering a counting query, i. e., Steps (CO1) and (CO2). The remaining steps are the following.

(EN3) Using $T[q_\ell..q_\ell + m - 1]$ and $T[q_r..q_r + m - 1]$ we can verify the existence of P in T (only necessary if $\ell \neq r$) and also find all positions where the pattern occurs. We send all these positions to processing element i.

(EN4) We have received all occurrences of P from the processing elements ℓ and r. Next, we need to retrieve all occurrences, i. e., the local suffix array from all processing elements j for $j \in (\ell, r)$.

The first two supersteps are the same as for a counting query. Therefore, the costs of the first two supersteps are the same. The third superstep is very similar to the third superstep for answering a counting query. The only difference is that we need to send the text positions of all occurrences to the processing element where the query arrived initially. This costs $\mathcal{O}(m + G \cdot occ + L)$, where occ denotes the number of occurrences of P at processing element ℓ and processing element r. Last, we need to retrieve the text positions of all occurrences of P in processing elements j with $j \in (\ell, r)$, which costs $\mathcal{O}((r - \ell)\frac{n}{p}G + L)$ if $\ell < r + 1$.

Answering any type (existential, counting or enumeration) of query has (asymptotically) the same cost for the first two supersteps, as we send at most twice as many queries to the second level (for counting and enumeration queries). To answer counting queries we send the leaf ranks during third superstep yielding a cost of $\mathcal{O}(m + G + L)$. In the last superstep we just need to add up the number of occurrences in $\mathcal{O}(1)$ time. When we consider enumeration queries, we need to report all text positions where P occurs. Let occ be the maximum number of occurrences of P in a processing element j for $j \in [\ell, r]$, then the cost of the third superstep is $\mathcal{O}(m + occ \cdot G + L)$. In the fourth superstep we need to retrieve all positions from the processing elements j for all $j \in (\ell, r)$ costing $\mathcal{O}(1 + Gp \cdot occ + L)$. All in all we get the following costs.

Lemma 6.4. *Let P be a pattern of size m. Then, answering an existential query costs $\mathcal{O}(t_{trie}(P) + Gm + L)$, answering a counting query costs $\mathcal{O}(t_{trie}(P) + Gm + L)$, and answering an enumeration query costs $\mathcal{O}(t_{trie}(P) + G\,(m + occ) + L)$, where occ denotes the total number of occurrences of P.*

Batched Queries and Load Balancing. When we process a batch of q queries at once rather than a single query, the number of supersteps does not increase, i. e., we can amortize the startup latencies of the BSP model over a large number of queries. Moreover, if the local work and communication volume is well balanced over the processing elements, the query throughput scales linearly with p. Balancing the queries itself can be achieved using any standard load balancing technique, i. e., assuming that $\mathcal{O}(q/p)$ queries arrive at each processing element is unproblematic.

Balancing how many queries get directed at each local trie is more difficult, since certain patterns might be more popular than others. However, we can use *virtualization*—we split the corpus into $p' \gg p$ pieces and distribute them randomly to the actual processing element. Similarly, some documents might be more popular than others. However, by randomly permuting the documents in the corpus, we can at least ensure that it is unlikely that many popular documents are assigned to the same processing element.

When all these balancing conditions are fulfilled, a batch Q of queries can be completed in time

$$\mathcal{O}(\frac{1}{p}\left(\sum_{P \in Q} t_{\text{trie}}(P) + G(|P| + occ(P))\right) + L),$$

where $occ(P)$ denotes the number of occurrences of P for an enumeration query (and 0 else).

Comparison to the Distributed Suffix Array. Using the (multiplexed) distributed suffix array (DSA) [Arr+14], a batch Q of q counting queries can be answered in time

$$\mathcal{O}(\frac{1}{p}\left(\sum_{P\in Q} t_{\text{Bin}}(P) + G(|P|\lg\frac{n}{q} + \lg p)\right) + L\lg\frac{pn}{q}),$$

where $t_{\text{Bin}}(P)$ denotes the time to identify the occurrences of the pattern P in the suffix array, i. e., $|P|\lg n$. Distributing the queries costs $\mathcal{O}(\sum_{P\in Q}|P| + G|P| + L)$ and is dominated by the costs of answering the batch of queries.

Comparing the costs of the DSA with our DPT we get the following result: The maximum time used for computation by each processing element is $\mathcal{O}(\frac{1}{p}\left(\sum_{P\in Q} t_{\text{trie}}(P)\right))$ using the DPT since we look in GT and at most two local PTs for each query. The computation time required by the DSA is $\mathcal{O}(\frac{1}{p}\left(\sum_{P\in Q} t_{\text{Bin}}(P)\right))$ and results from the binary searches (local and inter-processing element). Hence, the time used for computation by each processing element differs with respect to the time required for searching the corresponding suffix array interval for each pattern using a trie and using binary search. Usually, we can assume that $t_{\text{trie}}(P)$ is smaller than $t_{\text{Bin}}(P)$.

The cost of communication is $\mathcal{O}(\frac{1}{p}\left(\sum_{P\in Q} G(|P|)\right))$ using the DPT, as we just send each pattern to at most two processing elements and retrieve a substring of the length of the pattern. For the DSA the cost of communication is higher, i. e., $\mathcal{O}(\frac{1}{p}G\left(\sum_{P\in Q}|P|\lg\frac{n}{q} + \lg p\right))$ because more substrings need to be retrieved during the binary search. This effect can be moderated by storing pruned suffixes for each position of suffix array. Still, the DPT requires only a constant number of substrings to be retrieved for each query.

Last, the synchronization using DPT is constant, i. e., $\mathcal{O}(L)$, but using the DSA synchronization costs $\mathcal{O}(\lg\frac{pn}{q}L)$. Due to the constant number of messages being sent using the DPT, the synchronization cost is optimal and a logarithmic factor worse using the DSA.

Therefore, if we assume an optimal distribution of the queries and of the documents, the DPT is theoretically faster than the DSA. This difference in cost can also be seen in practice. However, the multiplexed DSA is very strong against query bias, whereas the DPT can be affected by query bias resulting in a load imbalance.

6.4 EXPERIMENTAL EVALUATION

We implemented the distributed Patricia trie. For the representations of bit vectors and the operations rank, select, and find_close, we use the succinct data structure library (SDSL) [Gog+14a]. Our implementation is available at `www.kurpicz.org/dpt`.

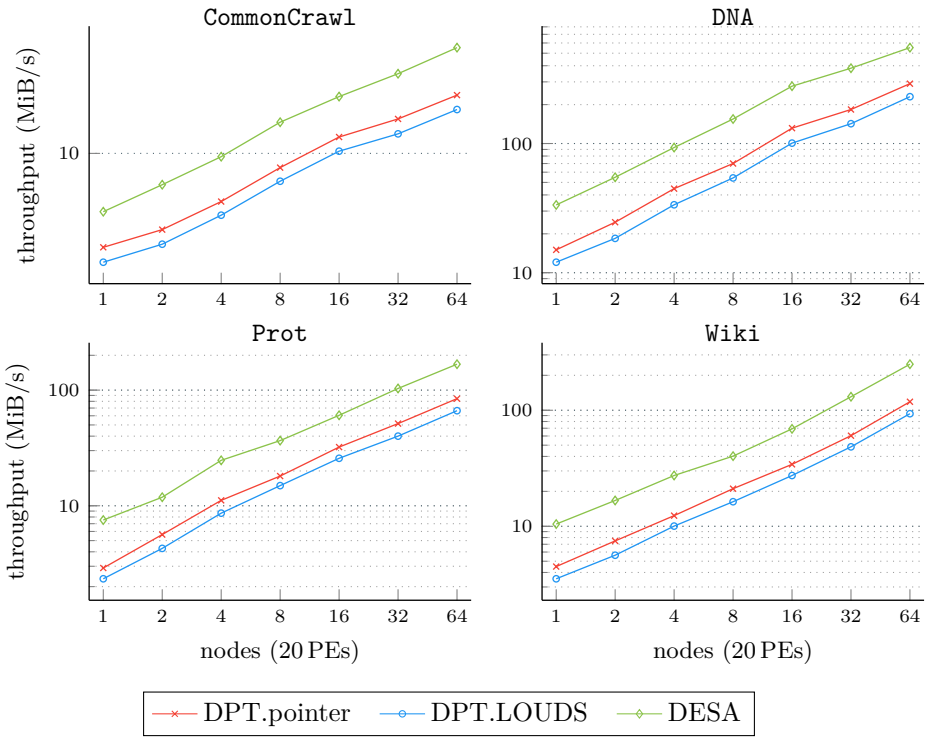


Figure 6.4. Throughput of distributed text index construction (512 MiB per node).

We compare the distributed Patricia trie with the recently presented distributed index [FA19] that we denote by *DESA*. Unfortunately, we were not able to compile the distributed suffix array by [Arr+14] on the cluster where we conducted the experiments. However, we refer to our previous experiments that show that the distributed patricia trie can answer queries faster than the distributed suffix array [Fis+17].

We conducted the experiments on LiDO.small nodes (Section 1.4.1), used the inputs presented in Section 1.4.2, and compiled the code using GCC 9.2.0 with flags `-O3` and `-march=native`. All reported running times are the average of five executions.

Construction Time. We first look at the construction times of both indices. Fortunately, both indices require the suffix array, LCP array, and text as input. Therefore, we can easily compare the construction times. We compare two variants of the DPT: DPT.pointer and DPT.LOUDS (represent the tree structure succinctly using LOUDS). We chose LOUDS over the other representations, because it is the fastest in practice.

Timing starts as soon as the suffix array, LCP array, and text are available in main memory on all processing elements. (Not the whole text is available at each processing element, just a slice of the text.) We end the timing as soon as the index is constructed.

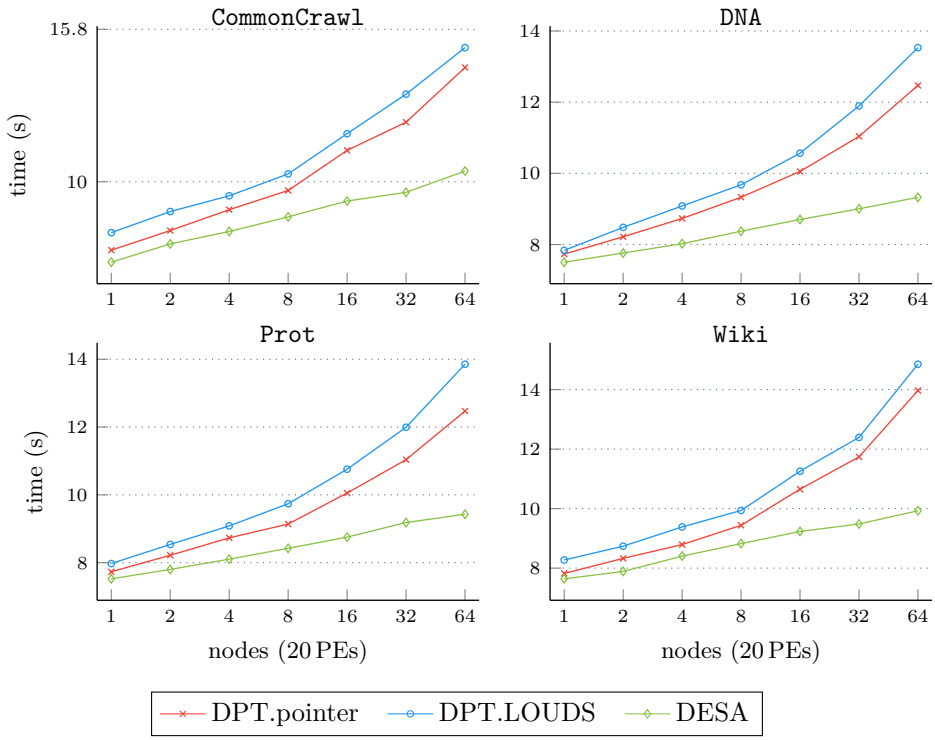


Figure 6.5. Running time for answering 10 million existential queries per PE.

The construction times are depicted in Figure 6.4. All algorithms behave similarly on all inputs. DESA can be constructed between 1.8 and 2.9 times faster than DPT.pointer (given the suffix and LCP array). DPT.LOUDS is the slowest to construct (around 20 % slower than DPT.pointer). All algorithms scale very well, but overall DESA achieves the highest speedup.

Query Time. Now, we measure the time the indices need to answer queries. We chose a setup similar to the one used by Flick and Aluru [FA19]: We generated 10 000 random queries of length 20 from the inputs and ran the queries 1 000 times. Overall we obtain 10 million queries for each input. (Flick and Aluru generated 1 000 queries and ran the queries 10 000 times, which leads to fewer different queries)

First, we consider existential queries, as DESA can only answer those. We show the existential query times in Figure 6.5. In this weak scaling experiment we see that the time required to answer 10 million queries per processing element. Initially, DESA and DPT.pointer answer existential queries at a similar speed. However, DESA balances the queries better, leading to a better scaling algorithm, which shows when using more than four nodes. It takes more time to answer queries for larger alphabets.

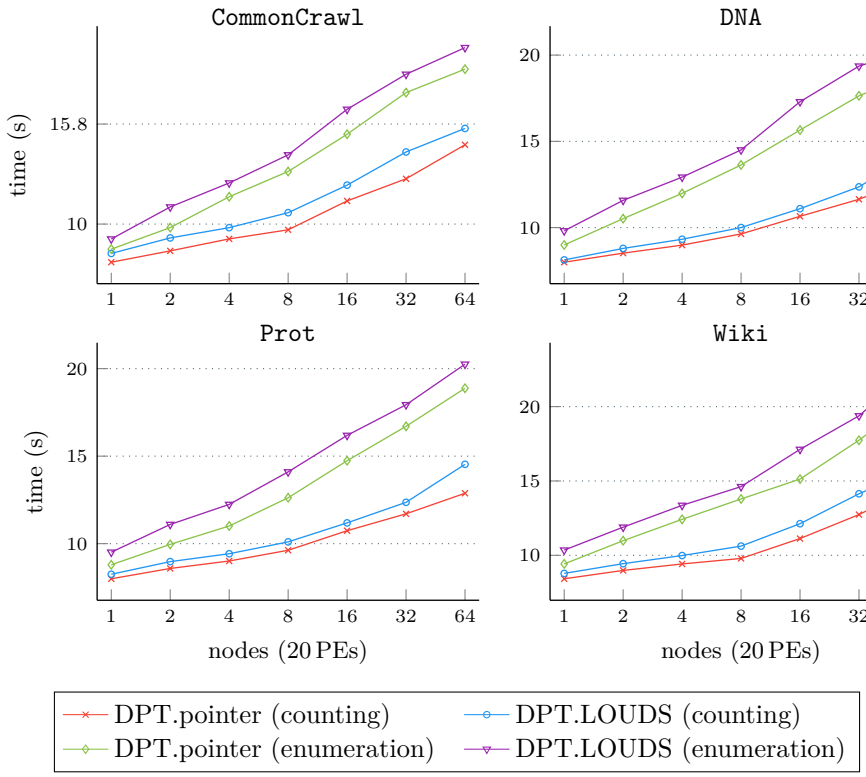


Figure 6.6. Running time for answering 10 million counting and enumeration queries per processing element.

In Figure 6.6, we see the same experiment for counting and enumeration queries. As expected, they require more time (because some queries are sent to two processing elements instead of one) but scale similarly to the existential queries (on the distributed Patricia trie). Enumeration queries are more time consuming than counting queries, as we need to aggregate the information instead of computing the result based on two positions.

6.5 CONCLUSION AND FUTURE WORK

We have presented a distributed Patricia trie that can answer existential, counting, and enumeration queries. Four years after the paper appeared, Flick and Aluru [FA19] presented a two-leveld index that is superior to our distributed Patricia trie in all our experiments (except that we can answer more types of queries). It is faster to construct and can answer (counting) queries faster and scales better.

Future Work. As we have seen in this chapter, building an index on top of the suffix array helps to answer queries very efficiently. However, if we want to reduce the required space further, we can look at indices based on the Burrows–Wheeler transform (BWT) [BW94]. One prominent examples is the FM-index [FM05]. While the idea of the FM-index is not directly applicable to distributed memory, we ask if building distributed indices based on the BWT is practical and results in better indices than using the suffix array?

APPENDIX A
DIVSUFSORT'S CODE

Here, we show the parts of DivSufSort's code that are described in detail in Section 4.2.

A.1 DIVSUFSORT.C

```
for(i=n-1, m=n, c0=T[n-1]; 0<=i;) {
  /* type A suffix. */
  do {++BUCKET_A(c1=c0);} while((0<=--i) && ((c0=T[i])>=c1));
  if(0 <= i) {
    /* type B* suffix. */
    ++BUCKET_BSTAR(c0, c1);
    SA[--m] = i;
    for(--i, c1=c0; 0<=i && (c0=T[i])<=c1; --i, c1=c0) {
      /* type B suffix. */
      ++BUCKET_B(c0, c1);
    }
  }
}
m = n - m;
/*...*/

for(c0 = 0, i = 0, j = 0; c0 < ALPHABET_SIZE; ++c0) {
  t = i + BUCKET_A(c0);
  BUCKET_A(c0) = i + j; /* start point */
  i = t + BUCKET_B(c0, c0);
  for(c1 = c0 + 1; c1 < ALPHABET_SIZE; ++c1) {
    j += BUCKET_BSTAR(c0, c1);
    BUCKET_BSTAR(c0, c1) = j; /* end point */
    i += BUCKET_B(c0, c1);
  }
}
/*...*/

PAb = SA + n - m; ISAb = SA + m;
for(i = m - 2; 0 <= i; --i) {
  t = PAb[i], c0 = T[t], c1 = T[t + 1];
  SA[--BUCKET_BSTAR(c0, c1)] = i;
}
t = PAb[m - 1], c0 = T[t], c1 = T[t + 1];
SA[--BUCKET_BSTAR(c0, c1)] = m - 1;
```

Listing A.1. Initialization of DivSufSort

```
#ifdef _OPENMP
    tmp = omp_get_max_threads();
    buf = SA + m, bufsize = (n - (2 * m)) / tmp;
    c0 = ALPHABET_SIZE - 2, c1 = ALPHABET_SIZE - 1, j = m;
#pragma omp parallel default(shared) \
  private(curbuf,k,l,d0,d1,tmp) {
      tmp = omp_get_thread_num();
      curbuf = buf + tmp * bufsize;
      k = 0;
      for(;;) {
        #pragma omp critical(sssort_lock)
        {
          if(0 < (l = j)) {
            d0 = c0, d1 = c1;
            do {
              k = BUCKET_BSTAR(d0, d1);
              if(--d1 <= d0) {
                d1 = ALPHABET_SIZE - 1;
                if(--d0 < 0) { break; }
              }
            } while(((l - k) <= 1) && (0 < (l = k)));
            c0 = d0, c1 = d1, j = k;
          }
        }
        if(l == 0) { break; }
        sssort(T, PAb, SA + k, SA + l,
               curbuf, bufsize, 2, n, *(SA + k) == (m - 1));
      }
    }
#else
  buf = SA + m, bufsize = n - (2 * m);
  for(c0 = ALPHABET_SIZE - 2, j = m; 0 < j; --c0) {
    for(c1 = ALPHABET_SIZE - 1; c0 < c1; j = i, --c1) {
      i = BUCKET_BSTAR(c0, c1); {
      if(1 < (j - i)) {
        sssort(T, PAb, SA + i, SA + j,
               buf, bufsize, 2, n, *(SA + i) == (m - 1));
      }
    }
  }
  /*...*/

  /* Compute ranks of type B* substrings. */
  for(i = m - 1; 0 <= i; --i) {
    if(0 <= SA[i]) {
      j = i;
      do {ISAb[SA[i]] = i;} while((0<=--i) && (0<= SA[i]));
      SA[i + 1] = i - j;
      if(i <= 0) { break; }
    }
    j = i;
    do {ISAb[SA[i] = ~SA[i]] = j;} while(SA[--i] < 0);
    ISAb[SA[i]] = j;
  }
```

```
158   /* Construct the ISA of type B* suffixes using trsort. */
159   trsort(ISAb, SA, m, 1);
160
161   /* Set the sorted order of tyoe B* suffixes. */
162   for(i = n - 1, j = m, c0 = T[n - 1]; 0 <= i;) {
163     for(--i, c1=c0; 0<=i && (c0=T[i])>=c1; --i, c1=c0) { }
164     if(0 <= i) {
165       t = i;
166       for(--i, c1=c0; 0<=i && (c0=T[i])<=c1; --i, c1=c0) { }
167       SA[ISAb[--j]] = ((t == 0) || (1 < (t - i))) ? t : ~t;
168     }
169   }
170
171   /* Calculate the index of start/end point of each bucket. */
172   BUCKET_B(ALPHABET_SIZE - 1, ALPHABET_SIZE - 1) = n;
173   for(c0 = ALPHABET_SIZE - 2, k = m - 1; 0 <= c0; --c0) {
174     i = BUCKET_A(c0 + 1) - 1;
175     for(c1 = ALPHABET_SIZE - 1; c0 < c1; --c1) {
176       t = i - BUCKET_B(c0, c1);
177       BUCKET_B(c0, c1) = i;
178
179       /* Move all type B* suffixes to the correct position. */
180       for(i=t, j=BUCKET_BSTAR(c0, c1);
181           j<=k;
182           --i, --k) { SA[i] = SA[k];
183       }
184     }
185     BUCKET_BSTAR(c0, c0 + 1) = i - BUCKET_B(c0, c0) + 1;
186     BUCKET_B(c0, c0) = i;
187   }
188 #endif
```

Listing A.2. Sorting B*-substrings

```
for(c1 = ALPHABET_SIZE - 2; 0 <= c1; --c1) {
  /* Scan the suffix array from right to left. */
  for(i = SA + BUCKET_BSTAR(c1, c1 + 1),
      j = SA + BUCKET_A(c1 + 1) - 1, k = NULL, c2 = -1;
      i <= j;
      --j) {
    if(0 < (s = *j)) {
      assert(T[s] == c1);
      assert(((s + 1) < n) && (T[s] <= T[s + 1]));
      assert(T[s - 1] <= T[s]);
      *j = ~s;
      c0 = T[--s];
      if((0 < s) && (T[s - 1] > c0)) { s = ~s; }
      if(c0 != c2) {
        if(0 <= c2) { BUCKET_B(c2, c1) = k - SA; }
        k = SA + BUCKET_B(c2 = c0, c1);
      }
      assert(k < j);
      *k-- = s;
    } else {
      assert(((s == 0) && (T[s] == c1)) || (s < 0));
      *j = ~s;
    }
  }
}

/* Construct the suffix array by using
   the sorted order of type B suffixes. */
k = SA + BUCKET_A(c2 = T[n - 1]);
*k++ = (T[n - 2] < c2) ? ~(n - 1) : (n - 1);
/* Scan the suffix array from left to right. */
for(i = SA, j = SA + n; i < j; ++i) {
  if(0 < (s = *i)) {
    assert(T[s - 1] >= T[s]);
    c0 = T[--s];
    if((s == 0) || (T[s - 1] < c0)) { s = ~s; }
    if(c0 != c2) {
      BUCKET_A(c2) = k - SA;
      k = SA + BUCKET_A(c2 = c0);
    }
    assert(i < k);
    *k++ = s;
  } else {
    assert(s < 0);
    *i = ~s;
  }
}
```

Listing A.3. Inducing phases (first inducing B- then A-suffixes)

A.2 SSSORT.C

```
746 void
747 sssort(const sauchar_t *T, const saidx_t *PA,
748        saidx_t *first, saidx_t *last,
749        saidx_t *buf, saidx_t bufsize,
750        saidx_t depth, saidx_t n, saint_t lastsuffix) {
751   saidx_t *a;
753   saidx_t *b, *middle, *curbuf;
754   saidx_t j, k, curbufsize, limit;
756   saidx_t i;
758   if(lastsuffix != 0) { ++first; }
759   /*..*/
763   if((bufsize < SS_BLOCKSIZE) &&
764      (bufsize < (last - first)) &&
765      (bufsize < (limit = ss_isqrt(last - first)))) {
766     if(SS_BLOCKSIZE < limit) { limit = SS_BLOCKSIZE; }
767     buf = middle = last - limit, bufsize = limit;
768   } else {
769     middle = last, limit = 0;
770   }
771   for(a = first, i = 0; SS_BLOCKSIZE < (middle - a);
771     a += SS_BLOCKSIZE, ++i) {
772     ss_mintrosort(T, PA, a, a + SS_BLOCKSIZE, depth);
773     /*...*/
777     curbufsize = last - (a + SS_BLOCKSIZE);
778     curbuf = a + SS_BLOCKSIZE;
779     if(curbufsize<=bufsize) {curbufsize=bufsize, curbuf=buf;}
780     for(b=a, k=SS_BLOCKSIZE, j=i; j&1; b-=k, k<<=1, j>>=1) {
781       ss_swapmerge(T,PA,b-k,b,b+k,curbuf,curbufsize,depth);
782     }
783   }
785   ss_mintrosort(T, PA, a, middle, depth);
786   /*...*/
789   for(k = SS_BLOCKSIZE; i != 0; k <<= 1, i >>= 1) {
790     if(i & 1) {
791       ss_swapmerge(T,PA,a-k,a,middle,buf,bufsize,depth);
792       a -= k;
793     }
794   }
795   if(limit != 0) {
797     ss_mintrosort(T, PA, middle, last, depth);
798     /*...*/
801     ss_inplacemerge(T, PA, first, middle, last, depth);
802   }
803   /*...*/
```

Listing A.4. sssort (first part)

```
804   /*...*/
805   if(lastsuffix != 0) {
806     /* Insert last type B* suffix. */
807     saidx_t PAi[2]; PAi[0] = PA[*(first - 1)], PAi[1] = n - 2;
808     for(a = first, i = *(first - 1);
809         (a<last) &&
809         ((*a<0) || (0<ss_compare(T,&(PAi[0]),PA+*a,depth)));
810         ++a) {
811       *(a - 1) = *a;
812     }
813     *(a - 1) = i;
814   }
815 }
```

Listing A.5. sssort (second part)

```
310 ss_mintrosort(const sauchar_t *T, const saidx_t *PA,
311                 saidx_t *first, saidx_t *last,
312                 saidx_t depth) {
313 #define STACK_SIZE SS_MISORT_STACKSIZE
314   struct { saidx_t *a, *b, c; saint_t d; } stack[STACK_SIZE];
315   const sauchar_t *Td;
316   saidx_t *a, *b, *c, *d, *e, *f;
317   saidx_t s, t;
318   saint_t ssize;
319   saint_t limit;
320   saint_t v, x = 0;

322   for(ssize = 0, limit = ss_ilg(last - first);;) {

324     if((last - first) <= SS_INSERTIONSORT_THRESHOLD) {
325   #if 1 < SS_INSERTIONSORT_THRESHOLD // = 8
326       if(1<(last-first)){ss_insertionsort(T,PA,first,last,depth);}
327   #endif
328       STACK_POP(first, last, depth, limit);
329       continue;
330     }

332     Td = T + depth;
333     if(limit-- == 0){ss_heapsort(Td, PA, first, {last-first);}
334     if(limit < 0) {
335       for(a = first + 1, v = Td[PA[*first]]; a < last; ++a) {
336         if((x = Td[PA[*a]]) != v) {
337           if(1 < (a - first)) { break; }
338           v = x;
339           first = a;
340         }
341       }
342       if(Td[PA[*first] - 1] < v) {
343         first = ss_partition(PA, first, a, depth);
344       }
345       if((a - first) <= (last - a)) {
346         if(1 < (a - first)) {
```

```
347             STACK_PUSH(a, last, depth, -1);
348             last = a, depth += 1, limit = ss_ilg(a - first);
349           } else {
350             first = a, limit = -1;
351           }
352         } else {
353           if(1 < (last - a)) {
354             STACK_PUSH(first, a, depth + 1, ss_ilg(a - first));
355             first = a, limit = -1;
356           } else {
357             last = a, depth += 1, limit = ss_ilg(a - first);
358           }
359         }
360         continue;
361       }

363       /*...*/
```

Listing A.6. ss_mintrosort (first part)

lp

```
363 /* choose pivot */
364 a = ss_pivot(Td, PA, first, last);
365 v = Td[PA[*a]];
366 SWAP(*first, *a);

368 /* partition */
369 for(b = first; (++b < last) && ((x = Td[PA[*b]]) == v);) { }
370 if(((a = b) < last) && (x < v)) {
371   for(; (++b < last) && ((x = Td[PA[*b]]) <= v);) {
372     if(x == v) { SWAP(*b, *a); ++a; }
373   }
374 }
375 for(c = last; (b < --c) && ((x = Td[PA[*c]]) == v);) { }
376 if((b < (d = c)) && (x > v)) {
377   for(; (b < --c) && ((x = Td[PA[*c]]) >= v);) {
378     if(x == v) { SWAP(*c, *d); --d; }
379   }
380 }
381 for(; b < c;) {
382   SWAP(*b, *c);
383   for(; (++b < c) && ((x = Td[PA[*b]]) <= v);) {
384     if(x == v) { SWAP(*b, *a); ++a; }
385   }
386   for(; (b < --c) && ((x = Td[PA[*c]]) >= v);) {
387     if(x == v) { SWAP(*c, *d); --d; }
388   }
389 }
```

Listing A.7. ss_mintrosort (second part)

```
391 if(a <= d) {
392   c = b - 1;

394   if((s = a - first) > (t = b - a)) { s = t; }
395   for(e=first, f=b-s; 0<s; --s, ++e, ++f) {SWAP(*e, *f);}
396   if((s = d - c) > (t = last - d - 1)) { s = t; }
397   for(e=b, f=last-s; 0<s; --s, ++e, ++f) {SWAP(*e, *f);}

399   a = first + (b - a), c = last - (d - c);
400   b = (v<=Td[PA[*a]-1]) ? a : ss_partition(PA, a, c, depth);

402   if((a - first) <= (last - c)) {
403     if((last - c) <= (c - b)) {
404       STACK_PUSH(b, c, depth + 1, ss_ilg(c - b));
405       STACK_PUSH(c, last, depth, limit);
406       last = a;
407     } else if((a - first) <= (c - b)) {
408       STACK_PUSH(c, last, depth, limit);
409       STACK_PUSH(b, c, depth + 1, ss_ilg(c - b));
410       last = a;
411     } else {
412       STACK_PUSH(c, last, depth, limit);
413       STACK_PUSH(first, a, depth, limit);
414       first = b, last = c, depth += 1, limit = ss_ilg(c - b);
415     }
416   } else {
417     if((a - first) <= (c - b)) {
418       STACK_PUSH(b, c, depth + 1, ss_ilg(c - b));
419       STACK_PUSH(first, a, depth, limit);
420       first = c;
421     } else if((last - c) <= (c - b)) {
422       STACK_PUSH(first, a, depth, limit);
423       STACK_PUSH(b, c, depth + 1, ss_ilg(c - b));
424       first = c;
425     } else {
426       STACK_PUSH(first, a, depth, limit);
427       STACK_PUSH(c, last, depth, limit);
428       first = b, last = c, depth += 1, limit = ss_ilg(c - b);
429     }
430   }
431 } else {
432   limit += 1;
433   if(Td[PA[*first] - 1] < v) {
434     first = ss_partition(PA, first, last, depth);
435     limit = ss_ilg(last - first);
436   }
437   depth += 1;
438 }
```

Listing A.8. ss_mintrosort (third part)

```
139 static INLINE
140 saint_t
141 ss_compare(const sauchar_t *T,
142             const saidx_t *p1, const saidx_t *p2,
143             saidx_t depth) {
144   const sauchar_t *U1, *U2, *U1n, *U2n;

146   for(U1 = T + depth + *p1,
147       U2 = T + depth + *p2,
148       U1n = T + *(p1 + 1) + 2,
149       U2n = T + *(p2 + 1) + 2;
150       (U1 < U1n) && (U2 < U2n) && (*U1 == *U2);
151       ++U1, ++U2) {
152   }

154   return U1 < U1n ?
155         (U2 < U2n ? *U1 - *U2 : 1) :
156         (U2 < U2n ? -1 : 0);
157 }
```

Listing A.9. ss_compare

```
165 static
166 void
167 ss_insertionsort(const sauchar_t *T, const saidx_t *PA,
168     saidx_t *first, saidx_t *last, saidx_t depth) {
169   saidx_t *i, *j;
170   saidx_t t;
171   saint_t r;

173   for(i = last - 2; first <= i; --i) {
174     for(t=*i, j=i+1; 0<(r=ss_compare(T,PA+t,PA+*j, depth));) {
175       do { *(j - 1) = *j; } while((++j < last) && (*j < 0));
176       if(last <= j) { break; }
177     }
178     if(r == 0) { *j = ~*j; }
179     *(j - 1) = t;
180   }
181 }
```

Listing A.10. ss_insertionsort

A.3 TRSORT.C

```
for(ISAd = ISA + depth; -n < *SA; ISAd += ISAd - ISA) {
  first = SA;
  skip = 0;
  unsorted = 0;
  do {
    if((t = *first) < 0) { first -= t; skip += t; }
    else {
      if(skip != 0) { *(first + skip) = skip; skip = 0; }
      last = SA + ISA[t] + 1;
      if(1 < (last - first)) {
        budget.count = 0;
        tr_introsort(ISA, ISAd, SA, first, last, &budget);
        if(budget.count != 0) { unsorted += budget.count; }
        else { skip = first - last; }
      } else if((last - first) == 1) {
        skip = -1;
      }
      first = last;
    }
  } while(first < (SA + n));
  if(skip != 0) { *(first + skip) = skip; }
  if(unsorted == 0) { break; }
}
```

Listing A.11. trsort

```
static
void
tr_copy(saidx_t *ISA, const saidx_t *SA,
        saidx_t *first, saidx_t *a, saidx_t *b, saidx_t *last,
        saidx_t depth) {
  /* sort suffixes of middle partition by using sorted
     order of suffixes of left and right partition. */
  saidx_t *c, *d, *e;
  saidx_t s, v;

  v = b - SA - 1;
  for(c = first, d = a - 1; c <= d; ++c) {
    if((0 <= (s = *c - depth)) && (ISA[s] == v)) {
      *++d = s;
      ISA[s] = d - SA;
    }
  }
  for(c = last - 1, e = d + 1, d = b; e < d; --c) {
    if((0 <= (s = *c - depth)) && (ISA[s] == v)) {
      *--d = s;
      ISA[s] = d - SA;
    }
  }
}
```

Listing A.12. tr_copy

```
325 static
326 void
327 tr_introsort(saidx_t *ISA, const saidx_t *ISAd,
328              saidx_t *SA, saidx_t *first, saidx_t *last,
329              trbudget_t *budget) {
330   /*...*/

319   for(ssize = 0, limit = tr_ilg(last - first);;) {

321     if(limit < 0) {
322       if(limit == -1) {
323         /* tandem repeat partition */
324         tr_partition(ISAd-incr,first,first,last,&a,&b,last-SA-1);

326         /* update ranks */
327         if(a < last) {
328           for(c=first,v=a-SA-1; c<a; ++c) {ISA[*c]=v;}
329         }
330         if(b < last) {
331           for(c=a,v=b-SA-1; c<b; ++c) {ISA[*c]=v;}
332         }

334         /* push */
335         if(1 < (b - a)) {
336           STACK_PUSH5(NULL, a, b, 0, 0);
337           STACK_PUSH5(ISAd - incr, first, last, -2, trlink);
338           trlink = ssize - 2;
339         }
340         if((a - first) <= (last - b)) {
341           if(1 < (a - first)) {
342             STACK_PUSH5(ISAd,b,last,tr_ilg(last-b),trlink);
343             last = a, limit = tr_ilg(a - first);
344           } else if(1 < (last - b)) {
345             first = b, limit = tr_ilg(last - b);
346           } else {
347             STACK_POP5(ISAd, first, last, limit, trlink);
348           }
349         } else {
350           if(1 < (last - b)) {
351             STACK_PUSH5(ISAd,first,a,tr_ilg(a-first),trlink);
352             first = b, limit = tr_ilg(last - b);
353           } else if(1 < (a - first)) {
354             last = a, limit = tr_ilg(a - first);
355           } else {
356             STACK_POP5(ISAd, first, last, limit, trlink);
357           }
358         }
359       } else if(limit == -2) {
360         /* tandem repeat copy */
361         a = stack[--ssize].b, b = stack[ssize].c;
362         if(stack[ssize].d == 0) {
363           tr_copy(ISA, SA, first, a, b, last, ISAd - ISA);
364         } else {
365           if(0 <= trlink) { stack[trlink].d = -1; }
```

```
366             tr_partialcopy(ISA,SA,first,a,b,last,ISAd-ISA);
367           }
368           STACK_POP5(ISAd, first, last, limit, trlink);
369         } else {
370           /* sorted partition */
371           /*...*/

424         }
425         continue;
426       }
427       /*...*
```

Listing A.13. tr_introsort

```
429 if((last - first) <= TR_INSERTIONSORT_THRESHOLD) {
430       tr_insertionsort(ISAd, first, last);
431       limit = -3;
432       continue;
433     }

435     if(limit-- == 0) {
436       tr_heapsort(ISAd, first, last - first);
437       for(a = last - 1; first < a; a = b) {
438         for(x=ISAd[*a],b=a-1;(first<=b)&&(ISAd[*b]==x);--b){*b=~*b;}
439       }
440       limit = -3;
441       continue;
442     }

444     /* choose pivot */
445     a = tr_pivot(ISAd, first, last);
446     SWAP(*first, *a);
447     v = ISAd[*first];

449     /* partition */
450     tr_partition(ISAd, first, first + 1, last, &a, &b, v);
451     if((last - first) != (b - a)) {
452       next = (ISA[*a] != v) ? tr_ilg(b - a) : -1;

454       /* update ranks */
455       for(c=first,v=a-SA-1; c<a; ++c) {ISA[*c]= v;}
456       if(b<last) {for(c=a,v=b-SA-1; c<b; ++c) {ISA[*c]=v;}}

458       /* push */
459       if((1 < (b - a)) && (trbudget_check(budget, b - a))) {
460         /*...*/

512       }
513     } else {
514       if(trbudget_check(budget, last - first)) {
515         limit = tr_ilg(last - first), ISAd += incr;
516       } else {
517         if(0 <= trlink) { stack[trlink].d = -1; }
518         STACK_POP5(ISAd, first, last, limit, trlink);
519       }
520     }
521   }
522 #undef STACK_SIZE
523 }
```

Listing A.14. trsort

BIBLIOGRAPHY

[Abd+14] Ahmed Abdelhadi, AH Kandil, and Mohamed Abouelhoda. "Cloud-Based Parallel Suffix Array Construction Based on MPI". In: *Middle East Conference on Biomedical Engineering (MECBME)*. IEEE Computer Society. 2014, pages 334–337.

[AN08] Donald A. Adjeroh and Fei Nan. "Suffix Sorting via Shannon-Fano-Elias Codes". In: *2008 Data Compression Conference (DCC)*. IEEE Computer Society, 2008, page 502.

[Arr+14] Diego Arroyuelo, Carolina Bonacic, Veronica Gil Costa, Mauricio Marín, and Gonzalo Navarro. "Distributed Text Search Using Suffix Arrays". In: *Parallel Comput.* 40.9 (2014), pages 471–495.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pages 1116–1127.

[Bab+15] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana A. Starikovskaya. "Wavelet Trees Meet Suffix Trees". In: *26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 2015, pages 572–591.

[Bad+16] Johannes Bader, Simon Gog, and Matthias Petri. "Practical Variable Length Gap Pattern Matching". In: *15th International Symposium on Experimental Algorithms (SEA)*. Volume 9685. Lecture Notes in Computer Science. Springer, 2016, pages 1–16.

[Bah+19a] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Hermann Foot, Florian Grieskamp, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. *Abschlussbericht der Projektgruppe 616*. 2019.

[Bah+19b] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. "SACABench: Benchmarking Suffix Array Construction". In: *26th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 11811. Lecture Notes in Computer Science. Springer, 2019, pages 392–406.

[Bai16] Uwe Baier. "Linear-time Suffix Sorting - A New Approach for Suffix Array Construction". In: *27th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 54. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz Center for Informatics, 2016, 23:1–23:12.

[Ben+05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. "Representing Trees of Higher Degree". In: *Algorithmica* 43.4 (2005), pages 275–292.

[Bha+91] P. C. P. Bhatt, Krzysztof Diks, Torben Hagerup, V. C. Prasad, Tomasz Radzik, and Sanjeev Saxena. "Improved Deterministic Parallel Integer Sorting". In: *Inf. Comput.* 94.1 (1991), pages 29–47.

[Bin+16a] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. "Inducing Suffix and LCP Arrays in External Memory". In: *ACM J. Exp. Algorithmics* 21.1 (2016), 2.3:1–2.3:27.

[Bin+16b] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-performance algorithmic distributed batch data processing with C++". In: *2016 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, 2016, pages 172–183.

[Bin+17] Timo Bingmann, Andreas Eberle, and Peter Sanders. "Engineering Parallel String Sorting". In: *Algorithmica* 77.1 (2017), pages 235–286.

[Bin+18] Timo Bingmann, Simon Gog, and Florian Kurpicz. "Scalable Construction of Text Indexes with Thrill". In: *2018 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, 2018, pages 634–643.

[Bin+20] Timo Bingmann, Peter Sanders, and Matthias Schimek. "Communication-Efficient String Sorting". In: *Computing Research Repository (CoRR)* arXiv:2001.08516 (2020). Full version to appear at IPDPS 2020.

[Bin12] Timo Bingmann. *pDCX,* `https://github.com/bingmann/pDCX`. 2012.

[Bin18] Timo Bingmann. "Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools". PhD thesis. Karlsruhe Institute of Technology, Germany, 2018.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. "Fast Lightweight Suffix Array Construction and Checking". In: *14th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 2676. LNCS. Springer, 2003, pages 55–69.

[Ble+96] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. "A Comparison of Sorting Algorithms for the Connection Machine CM-2". In: *Commun. ACM* 39.12es (1996), pages 273–297.

[BS97] Jon Louis Bentley and Robert Sedgewick. "Fast Algorithms for Sorting and Searching Strings". In: *8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 1997, pages 360–369.

[BW94] Michael Burrows and David John Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Technical report. Digital Equipment Corporation, 1994.

[Cas+08] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. CRC Press, 2008.

[CF02] Andreas Crauser and Paolo Ferragina. "A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory". In: *Algorithmica* 32.1 (2002), pages 1–35.

[Cla+11] Francisco Claude, Patrick K. Nicholson, and Diego Seco. "Space Efficient Wavelet Tree Construction". In: *18th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 7024. Lecture Notes in Computer Science. Springer, 2011, pages 185–196.

[Cla+15] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. "The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets". In: *Inf. Syst.* 47 (2015), pages 15–32.

[Cla97] David Clark. "Compact Pat Trees". PhD thesis. 1997.

[Cli05] Raphaël Clifford. "Distributed Suffix Trees". In: *J. Discrete Algorithms* 3.2-4 (2005), pages 176–197.

[CN08] Francisco Claude and Gonzalo Navarro. "Practical Rank/Select Queries over Arbitrary Sequences". In: *15th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 5280. Lecture Notes in Computer Science. Springer, 2008, pages 176–187.

[Dem+08a] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. "Better External Memory Suffix Array Construction". In: *ACM J. Exp. Algorithmics* 12 (2008), 3.4:1–3.4:24.

[Dem+08b] Roman Dementiev, Lutz Kettner, and Peter Sanders. "STXXL: Standard Template Library for XXL Data Sets". In: *Software: Practice and Experience* 38.6 (2008), pages 589–637.

[Din+20] Patrick Dinklage, Johannes Fischer, and Florian Kurpicz. "Distributed Wavelet Tree Construction". In: *22nd Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2020, pages 214–228.

[Din19] Patrick Dinklage. "Translating Between Wavelet Tree and Wavelet Matrix Construction". In: *Prague Stringology Conference (PSC)*. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2019, pages 126–135.

[DK13] Mrinal Deo and Sean Keely. "Parallel Suffix Array and Least Common Prefix for the GPU". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Association for Computing Machinery, 2013, pages 197–206.

[EK19] Jonas Ellert and Florian Kurpicz. "Parallel External Memory Wavelet Tree and Wavelet Matrix Construction". In: *26th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 11811. Lecture Notes in Computer Science. Springer, 2019, pages 407–416.

[FA15] Patrick Flick and Srinivas Aluru. "Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Association for Computing Machinery, 2015, 16:1–16:10.

[FA19] Patrick Flick and Srinivas Aluru. "Distributed Enhanced Suffix Arrays: Efficient Algorithms for Construction and Querying". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*. Association for Computing Machinery, 2019, 72:1–72:17.

[Far97] Martin Farach. "Optimal Suffix Tree Construction with Large Alphabets". In: *38th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1997, pages 137–143.

[Fer+09] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. "The Myriad Virtues of Wavelet Trees". In: *Inf. Comput.* 207.8 (2009), pages 849–866.

[FG15] Johannes Fischer and Pawel Gawrychowski. "Alphabet-Dependent String Searching with Wexponential Search Trees". In: *26th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 9133. Lecture Notes in Computer Science. Springer, 2015, pages 160–171.

[FG99] Paolo Ferragina and Roberto Grossi. "The String B-tree: A New Data Structure for String Search in External Memory and Its Applications". In: *J. ACM* 46.2 (1999), pages 236–280.

[FH11] Johannes Fischer and Volker Heun. "Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays". In: *SIAM J. Comput.* 40.2 (2011), pages 465–492.

[Fis+17] Johannes Fischer, Florian Kurpicz, and Peter Sanders. "Engineering a Distributed Full-Text Index". In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2017, pages 120–134.

[Fis+18] Johannes Fischer, Florian Kurpicz, and Marvin Löbel. "Simple, Fast and Lightweight Parallel Wavelet Tree Construction". In: *20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2018, pages 9–20.

[Fis11] Johannes Fischer. "Inducing the LCP-Array". In: *12th International Symposium on Algorithms and Data Structures (WADS)*. Volume 6844. Lecture Notes in Computer Science. Springer, 2011, pages 374–385.

[FK17] Johannes Fischer and Florian Kurpicz. "Dismantling DivSufSort". In: *Prague Stringology Conference (PSC)*. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017, pages 62–76.

[FK19] Johannes Fischer and Florian Kurpicz. "Lightweight Distributed Suffix Array Construction". In: *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2019, pages 27–38.

[FL99] Paolo Ferragina and Fabrizio Luccio. "String Search in Coarse-Grained Parallel Computers". In: *Algorithmica* 24.3-4 (1999), pages 177–194.

[FM05] Paolo Ferragina and Giovanni Manzini. "Indexing Compressed Text". In: *J. ACM* 52.4 (2005), pages 552–581.

[FN05] Paolo Ferragina and Gonzalo Navarro. *Pizza & Chili Corpus, `http://pizzachili.dcc.uchile.cl/index.html`*. 2005.

[FS+17] José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. "Parallel Construction of Wavelet Trees on Multicore Architectures". In: *Knowl. Inf. Syst.* 51.3 (2017), pages 1043–1066.

[FS17] Paulo G. S. da Fonseca and Israel B. F. da Silva. "Online Construction of Wavelet Trees". In: *16th International Symposium on Experimental Algorithms (SEA)*. Volume 75. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 16:1–16:14.

[Fut+01] Natsuhiko Futamura, Srinivas Aluru, and Stefan Kurtz. "Parallel Suffix Sorting". In: *Electrical Engineering and Computer Science* 64 (2001).

[GO11] Simon Gog and Enno Ohlebusch. "Fast and Lightweight LCP-Array Construction Algorithms". In: *13th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2011, pages 25–34.

[Gog+14a] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *13th International Symposium on Experimental Algorithms (SEA)*. Volume 8504. Lecture Notes in Computer Science. Springer, 2014, pages 326–337.

[Gog+14b] Simon Gog, Alistair Moffat, J. Shane Culpepper, Andrew Turpin, and Anthony Wirth. "Large-Scale Pattern Search Using Reduced-Space On-Disk Suffix Arrays". In: *IEEE Trans. Knowl. Data Eng.* 26.8 (2014), pages 1918–1931.

[Gog+19] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. "Fixed Block Compression Boosting in FM-Indexes: Theory and Practice". In: *Algorithmica* 81.4 (2019), pages 1370–1391.

[Gon+92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. "New Indices for Text: Pat Trees and Pat Arrays". In: *Information Retrieval: Data Structures & Algorithms*. 1992, pages 66–82.

[Got17] Keisuke Goto. "Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets". In: *Computing Research Repository (CoRR)* arXiv:1703.01009 (2017).

[Gro+03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. "High-Order Entropy-Compressed Text Indexes". In: *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics. 2003, pages 841–850.

[Gro+11] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. "Wavelet Trees: From Theory to Practice". In: *1st International Conference on Data Compression, Communications and Processing (CCP)*. IEEE Computer Society, 2011, pages 210–221.

[Hag98] Torben Hagerup. "Sorting and Searching on the Word RAM". In: *15th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. Volume 1373. Lecture Notes in Computer Science. Springer, 1998, pages 366–398.

[Hon+09] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. "Breaking a Time-and-Space Barrier in Constructing Full-Text Indices". In: *SIAM J. Comput.* 38.6 (2009), pages 2162–2178.

[Huf52] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pages 1098–1101.

[IT99] Hideo Itoh and Hozumi Tanaka. "An Efficient Method for in Memory Construction of Suffix Arrays". In: *6th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. IEEE Computer Society, 1999, pages 81–88.

[Jac89] Guy Jacobson. "Space-efficient Static Trees and Graphs". In: *30th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1989, pages 549–554.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KA05] Pang Ko and Srinivas Aluru. "Space Efficient Linear Time Construction of Suffix Arrays". In: *J. Discrete Algorithms* 3.2-4 (2005), pages 143–156.

[Kan18] Yusaku Kaneta. "Fast Wavelet Tree Construction in Practice". In: *25th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 11147. Lecture Notes in Computer Science. Springer, 2018, pages 218–232.

[Kas+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications". In: *12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 2089. Lecture Notes in Computer Science. Springer, 2001, pages 181–192.

[Kim+04] Dong Kyue Kim, Junha Jo, and Heejin Park. "A Fast Algorithm for Constructing Suffix Arrays for Fixed-Size Alphabets". In: *3rd International Workshop on Experimental and Efficient Algorithms (WEA)*. Volume 3059. Lecture Notes in Computer Science. Springer, 2004, pages 301–314.

[Kim+05] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. "Constructing Suffix Arrays in Linear Time". In: *J. Discrete Algorithms* 3.2-4 (2005), pages 126–142.

[KK17] Juha Kärkkäinen and Dominik Kempa. "Engineering a Lightweight External Memory Suffix Array Construction Algorithm". In: *Math. Comput. Sci.* 11.2 (2017), pages 137–149.

[KK19] Dominik Kempa and Tomasz Kociumaka. "String Synchronizing Sets: Sublinear-Time BWT Construction and Optimal LCE Data Structure". In: *51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. Association for Computing Machinery, 2019, pages 756–767.

[KM99] S. Rao Kosaraju and Giovanni Manzini. "Compression of Low Entropy Strings with Lempel-Ziv Algorithms". In: *SIAM J. Comput.* 29.3 (1999), pages 893–911.

[Knu14] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. 5th. Addison-Wesley Professional, 2014.

[KR08] Juha Kärkkäinen and Tommi Rantala. "Engineering Radix Sort for Strings". In: *15th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 5280. Lecture Notes in Computer Science. Springer, 2008, pages 3–14.

[KS03] Juha Kärkkäinen and Peter Sanders. "Simple Linear Work Suffix Array Construction". In: *30th International Colloquium on Automata, Languages, and Programming (ICALP)*. Volume 2719. Lecture Notes in Computer Science. Springer, 2003, pages 943–955.

[KS07] Fabian Kulla and Peter Sanders. "Scalable Parallel Suffix Array Construction". In: *Parallel Comput.* 33.9 (2007), pages 605–612.

[Kär+06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. "Linear Work Suffix Array Construction". In: *J. ACM* 53.6 (2006), pages 918–936.

[Kär+09] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. "Permuted Longest-Common-Prefix Array". In: *20th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 5577. Lecture Notes in Computer Science. Springer, 2009, pages 181–192.

[Kär+15] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. "Parallel External Memory Suffix Sorting". In: *16th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 9133. Lecture Notes in Computer Science. Springer, 2015, pages 329–342.

[Kär+16] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. "Lazy Lempel-Ziv Factorization Algorithms". In: *ACM J. Exp. Algorithmics* 21.1 (2016), 2.4:1–2.4:19.

[Kär+17] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. "Engineering External Memory Induced Suffix Sorting". In: *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2017, pages 98–108.

[Lab+17] Julian Labeit, Julian Shun, and Guy E. Blelloch. "Parallel Lightweight Wavelet Tree, Suffix Array and FM-Index Construction". In: *J. Discrete Algorithms* 43 (2017), pages 2–17.

[Lao+18a] Bin Lao, Ge Nong, Wai Hong Chan, and Jing Yi Xie. "Fast In-Place Suffix Sorting on a Multicore Computer". In: *IEEE Trans. Comput* 67.12 (2018), pages 1737–1749.

[Lao+18b] Bin Lao, Ge Nong, Wai Hong Chan, and Yi Pan. "Fast Induced Sorting Suffixes on a Multicore Machine". In: *J. Supercomput.* 74.7 (2018), pages 3468–3485.

[Li+18] Zhize Li, Jian Li, and Hongwei Huo. "Optimal In-Place Suffix Sorting". In: *2018 Data Compression Conference (DCC)*. IEEE Computer Society, 2018, page 422.

[LS07] N. Jesper Larsson and Kunihiko Sadakane. "Faster Suffix Sorting". In: *Theor. Comput. Sci.* 387.3 (2007), pages 258–272.

[Mak12] Christos Makris. "Wavelet trees: A survey". In: *Comput. Sci. Inf. Syst.* 9.2 (2012), pages 585–625.

[Man04] Giovanni Manzini. "Two Space Saving Tricks for Linear Time LCP Array Computation". In: *9th Scandinavian Workshop on Algorithm Theory (SWAT)*. Volume 3111. Lecture Notes in Computer Science. Springer, 2004, pages 372–383.

[McS+15] Frank McSherry, Michael Isard, and Derek Gordon Murray. "Scalability! But at what COST?" In: *HotOS15*. USENIX Association, 2015.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Volume 1. EATCS Monographs on Theoretical Computer Science. Springer, 1984.

[Met+16] Ahmed A Metwally, Ahmed H Kandil, and Mohamed Abouelhoda. "Distributed Suffix Array Construction Algorithms: Comparison of Two Algorithms". In: *Cairo International Biomedical Engineering Conference (CIBEC)*. IEEE Computer Society. 2016, pages 27–30.

[MF04] Giovanni Manzini and Paolo Ferragina. "Engineering a Lightweight Suffix Array Construction Algorithm". In: *Algorithmica* 40.1 (2004), pages 33–50.

[MM93] Udi Manber and Eugene W. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM J. Comput.* 22.5 (1993), pages 935–948.

[MN06] Veli Mäkinen and Gonzalo Navarro. "Position-Restricted Substring Searching". In: *7th Latin American Symposium on Theoretical Informatics (LATIN)*. Volume 3887. Lecture Notes in Computer Science. Springer, 2006, pages 703–714.

[MN07] Veli Mäkinen and Gonzalo Navarro. "Rank and Select Revisited and Extended". In: *Theor. Comput. Sci.* 387.3 (2007), pages 332–347.

[Mor06] Yuta Mori. *divsufsort,* `https://github.com/y-256/libdivsufsort`. 2006.

[Mor08] Yuta Mori. *sais,* `https://sites.google.com/site/yuta256/sais`. 2008.

[Mor68] Donald R. Morrison. "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4 (1968), pages 514–534.

[MP07] Michael A. Maniscalco and Simon J. Puglisi. "An Efficient, Versatile Approach to Suffix Sorting". In: *ACM J. Exp. Algorithmics* 12 (2007), 1.2:1–1.2:23.

[MR01] J. Ian Munro and Venkatesh Raman. "Succinct Representation of Balanced Parentheses and Static Trees". In: *SIAM J. Comput.* 31.3 (2001), pages 762–776.

[Mun+16] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. "Fast Construction of Wavelet Trees". In: *Theor. Comput. Sci.* 638 (2016), pages 91–97.

[Mus97] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Software: Practice and Experience* 27.8 (1997), pages 983–993.

[Mäk+04] Veli Mäkinen, Gonzalo Navarro, and Kunihiko Sadakane. "Advantages of Backward Searching - Efficient Secondary Memory and Distributed Implementation of Compressed Suffix Arrays". In: *15th International Symposium on Algorithms and Computation (ISAAC)*. Volume 3341. Lecture Notes in Computer Science. Springer, 2004, pages 681–692.

[Mäk+15] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

[Na05] Joong Chae Na. "Linear-Time Construction of Compressed Suffix Arrays Using o(n log n)-Bit Working Space for Large Alphabets". In: *16th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 3537. Lecture Notes in Computer Science. Springer, 2005, pages 57–67.

[Nav+97] Gonzalo Navarro, Joao Paulo Kitajima, Berthier A. Ribeiro-Neto, and Nivio Ziviani. "Distributed Generation of Suffix Arrays". In: *8th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 1264. Lecture Notes in Computer Science. Springer, 1997, pages 102–115.

[Nav14] Gonzalo Navarro. "Wavelet Trees for All". In: *J. Discrete Algorithms* 25 (2014), pages 2–20.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[Non+11] Ge Nong, Sen Zhang, and Wai Hong Chan. "Two Efficient Algorithms for Linear Time Suffix Array Construction". In: *IEEE Trans. Comput* 60.10 (2011), pages 1471–1484.

[Non+15] Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. "Induced Sorting Suffixes in External Memory". In: *ACM Trans. Inf. Syst.* 33.3 (2015), 12:1–12:15.

[Non13] Ge Nong. "Practical Linear-Time O(1)-Workspace Suffix Sorting for Constant Alphabets". In: *ACM Trans. Inf. Syst.* 31.3 (2013), page 15.

[NP13] Gonzalo Navarro and Alberto Ordóñez Pereira. "Compressing Huffman Models on Large Alphabets". In: *2013 Data Compression Conference (DCC)*. IEEE Computer Society, 2013, pages 381–390.

[NS14] Gonzalo Navarro and Kunihiko Sadakane. "Fully Functional Static and Dynamic Succinct Trees". In: *ACM Trans. Algorithms* 10.3 (2014), 16:1–16:39.

[NZ07] Ge Nong and Sen Zhang. "Optimal Lightweight Construction of Suffix Arrays for Constant Alphabets". In: *10th International Symposium on Algorithms and Data Structures (WADS)*. Volume 4619. Lecture Notes in Computer Science. Springer, 2007, pages 613–624.

[Oes16] Benedikt Oesing. *Effiziente Erstellung von Waveletmatrizen (B. Sc. Thesis in German)*. 2016. URL: `https://ls11-www.cs.tu-dortmund.de/fischer/abschlussarbeiten/wavelet`.

[Osi12] Vitaly Osipov. "Parallel Suffix Array Construction for Shared Memory Architectures". In: *19th International Symposium on String Processing and Inforation Retrieval (SPIRE)*. Volume 7608. Lecture Notes in Computer Science. Springer, 2012, pages 379–384.

[PS15] Jacopo Pantaleoni and Nuno Subtil. *nvbio, `https://github.com/NVlabs/nvbio`*. 2015.

[Pug+07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. "A Taxonomy of Suffix Array Construction Algorithms". In: *ACM Comput. Surv.* 39.2 (2007), article no. 4.

[Ram90] Rajeev Raman. "The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting". In: *10th Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Volume 472. Lecture Notes in Computer Science. Springer, 1990, pages 161–175.

[Rus+10] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. "Parallel and Distributed Compressed Indexes". In: *10th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 6129. Lecture Notes in Computer Science. Springer, 2010, pages 348–360.

[Sch19] Matthias Schimek. *Distributed String Sorting Algorithms*. Master's Thesis. 2019. URL: `https://publikationen.bibliothek.kit.edu/1000098432`.

[SD10] Jared T. Simpson and Richard Durbin. "Efficient Construction of an Assembly String Graph Using the FM-Index". In: *Bioinformatics [ISMB]* 26.12 (2010), pages 367–373.

[Sed98] Robert Sedgewick. *Algorithms in C - Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3. Ed.)* Addison-Wesley-Longman, 1998.

[Sew00] Julian Seward. "On the Performance of BWT Sorting Algorithms". In: *2000 Data Compression Conference (DCC)*. IEEE Computer Society, 2000, pages 173–182.

[Shu15] Julian Shun. "Parallel Wavelet Tree Construction". In: *2015 Data Compression Conference (DCC)*. IEEE Computer Society, 2015, pages 63–72.

[Shu17] Julian Shun. "Improved Parallel Construction of Wavelet Trees and Rank/Select Structures". In: *2017 Data Compression Conference (DCC)*. IEEE Computer Society, 2017, pages 92–101.

[Sin38] James Singer. "A Theorem in Finite Projective Geometry and Some Applications to Number Theory". In: *Trans. Am. Math. Soc.* 43.3 (1938), pages 377–385. ISSN: 0002-9947.

[SM09] Weidong Sun and Zongmin Ma. "Parallel Lexicographic Names Construction with CUDA". In: *15th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, 2009, pages 913–918.

[SS07] Klaus-Bernd Schürmann and Jens Stoye. "An Incomplex Algorithm for Fast Suffix Array Construction". In: *Software: Practice and Experience* 37.3 (2007), pages 309–329.

[Tis11] German Tischler. "On Wavelet Tree Construction". In: *22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*. Volume 6661. Lecture Notes in Computer Science. Springer, 2011, pages 208–218.

[Val90] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (1990), pages 103–111.

[Vis10] Uzi Vishkin. *Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques*. 2010.

[Wan+16] Leyuan Wang, Sean Baxter, and John D. Owens. "Fast Parallel Skew and Prefix-Doubling Suffix Array Construction on the GPU". In: *Concurrency and Computation: Practice and Experience* 28.12 (2016), pages 3466–3484.