Practical Wavelet Tree Construction

PATRICK DINKLAGE, JONAS ELLERT, JOHANNES FISCHER, FLORIAN KURPICZ, and MARVIN LÖBEL, TU Dortmund University, Germany

We present new sequential and parallel algorithms for wavelet tree construction based on a new *bottom-up* technique. This technique makes use of the structure of the wavelet trees—refining the characters represented in a node of the tree with increasing depth—in an opposite way, by first computing the leaves (most refined), and then propagating this information upwards to the root of the tree. We first describe new sequential algorithms, both in RAM and external memory. Based on these results, we adapt these algorithms to parallel computers, where we address both shared memory and distributed memory settings.

In practice, all our algorithms outperform previous ones in both time and memory efficiency, because we can compute all auxiliary information solely based on the information we obtained from computing the leaves. Most of our algorithms are also adapted to the wavelet *matrix*, a variant that is particularly suited for large alphabets.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Computing methodologies → Parallel algorithms**;

Additional Key Words and Phrases: Data structures, text indexing, shared memory, distributed memory, external memory

ACM Reference format:
Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. 2021. Practical Wavelet Tree Construction. *J. Exp. Algorithmics* 26, 1, Article 1.8 (July 2021), 67 pages.
https://doi.org/10.1145/3457197

1 INTRODUCTION

Starting with Ferragina and Manzini's presentation of a Burrows-Wheeler-Transform-based full-text index [15], it became evident that two basic queries on a string need to be supported efficiently: counting the number of occurrences of a given character α up to a given position i (operation "$rank_\alpha(i)$"), and finding the position of the ith occurrence of a character α (the inverse operation "$select_\alpha(i)$"). *Wavelet Trees* [21] have emerged as the ideal data structure for supporting these queries. On a high level, they can be viewed as a "restructuring" of the bits of the input string

Parts of this article were already published in References [12, 13, 16].
Patrick Dinklage and Florian Kurpicz were partially supported by the German Research Foundation DFG SPP 1736 "Algorithms for Big Data." Additionally, the authors are grateful for the provided computing time on the Linux HPC cluster at Technical University Dortmund (LiDO3), partially funded in the course of the Large-scale Equipment Initiative by the German Research Foundation (DFG) as Project No. 271512359.
Authors' address: P. Dinklage, J. Ellert, J. Fischer, F. Kurpicz, and M. Löbel, TU Dortmund University, Department of Computer Science, 44221, Dortmund, Germany; emails: patrick.dinklage@tu-dortmund.de, jonas.ellert@tu-dortmund.de, johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de, marvin.loebel@tu-dortund.de.

1084-6654/2021/07-ART1.8
https://doi.org/10.1145/3457197

such that the above queries can be reduced to a logarithmic number of such queries on binary strings. The "restructuring" needs to be done in advance (as part of the index construction); this is what we call "wavelet tree construction." Given the practical significance of wavelet trees (they are used, e.g., in the de-facto standard for DNA read aligners [32]), it is of high importance to have fast, memory-efficient, and well-scaling algorithms for their construction. This article addresses precisely this issue.

Wavelet trees (and their variant *wavelet matrices*) are also used for compression [22, 34], in computational geometry as an alternative to fractional cascading [4], for variable length gapped pattern matching [3], and as a tool to compute the Burrows-Wheeler Transform [28], to mention only a few examples. Additional information on further applications can be found in multiple surveys [14, 22, 34, 39].

We consider different models of computation that are highly relevant in practice. To be more precise, we present wavelet tree and wavelet matrix construction algorithms in shared, distributed, and external memory.

Our Contributions. Our new wavelet tree and matrix construction algorithms are based on a common novel technique that we call *bottom-up* computation. 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 first present three sequential (Section 5) and three shared memory parallel (Section 6) wavelet tree 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. The sequential construction algorithms are the most memory efficient and fastest (publicly available) implementations.[1] Our shared memory parallel construction algorithms are also the fastest and most memory efficient implementations.

Then, in Section 7, we adapt our algorithms to compute the wavelet tree and wavelet matrix efficiently in distributed memory. These algorithms are the first distributed wavelet tree and wavelet matrix construction algorithms. Our algorithms scale well; if we use more than four compute nodes, then our fastest distributed construction algorithm achieves a higher throughput than our fastest shared memory construction algorithm using all cores on one node. Additionally, we can compute the wavelet matrix for arbitrarily large alphabets.

Next, we present the first external memory wavelet tree and wavelet matrix construction algorithms in Section 8. Here, we also use the basic idea of the bottom-up computation to reduce the number of I/Os, because we can compute all information required for the computation in one scan of the text. This results in the first practical external memory wavelet tree and wavelet matrix construction algorithms.

Finally, in Section 9, 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 input text. To retain functionality, special Huffman codes are used.

The results in Sections 5–8 were already published in References [12, 13, 16]. However, compared with these publications, we significantly improved the performance of our shared memory implementations and used (as far as possible) the same experimental setup for all evaluations to obtain comparable results (even for different models of computation). Additionally, the Huffman-shaped wavelet tree and wavelet matrix construction algorithms in Section 9 have not been published before.

[1]The source code is available at https://github.com/kurpicz/pwm. All of our source code is linked at the beginning of corresponding experimental evaluations.

α	0	1	2	3	4	5	6	7	
$\text{bit}(\alpha)$	$(0$	$(0$	$(0$	$(0$	$(1$	$(1$	$(1$	$(1$	MSB
	0	0	1	1	0	0	1	1	
	$0)_2$	$1)_2$	$0)_2$	$1)_2$	$0)_2$	$1)_2$	$0)_2$	$1)_2$	LSB

Fig. 1. Binary representation of all characters in $\Sigma = [0, 8)$.

2 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 $\text{bits}(\alpha)$; see Figure 1. Whenever we write a binary representation of a value, we indicate it by a subscript two. The kth bit (from MSB to LSB) of a character α is denoted by $\text{bit}(k, \alpha)$ for all $0 \leq k < \lceil \lg \sigma \rceil$. The *bit prefix* of size k of $\alpha \in \Sigma$ are the k most significant bits, i.e., $\text{bit\_prefix}(k, \alpha) = (\text{bit}(0, \alpha) \dots \text{bit}(k-1, \alpha))_2$. We interpret sequences of bits as integer values.

Let BV be a bit vector of size n. The operation $rank_0(\text{BV}, i)$ returns the number of 0's in $\text{BV}[0, i)$, whereas $select_0(\text{BV}, i)$ returns the position of the ith 0 in BV. The operations $rank_1(\text{BV}, i)$ and $select_1(\text{BV}, i)$ are defined analogously. Both rank and select queries on a bit vector of size n can be answered in $O(1)$ time using succinct dictionary data structures that requires only $o(n)$ bits space [40].

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, then B is defined as $B[0] = A[0]$ and $B[i] = A[i-1] + B[i-1]$ for all $i \in [1, n)$.

2.1 Machine Models

When analyzing running times, our model of computation is the word **RAM (random access machine)** model [23] with (computer) word size $w = \Omega(\lg n)$ for inputs of size n. Here, we can access a computer word in $O(1)$ time. 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. 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. PE is an abstract notion that can mean different things, depending on the setting. Now, we describe different models and mention what a PE refers to.

2.1.1 Parallel Random Access Memory Model. The first parallel model we consider is the *shared memory* parallel model. Here, all PEs have a shared memory that they can use to communicate, by writing to designated memory addresses in the shared memory. We analyze parallel shared memory algorithms using JáJá's work-time model [25] 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.,

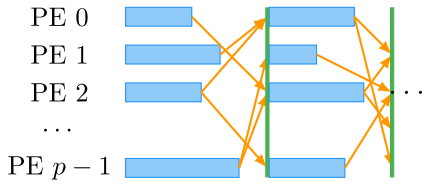


Fig. 2. Two supersteps in the BSP model. Local computation is depicted in blue, communication is highlighted in orange, and synchronization barriers are shown in green.

its sequential running time. The time, however, is the number of time units that are needed to execute the algorithm, when operations can be executed in parallel by different PEs of which we have unlimited many. In this article, we consider the CREW PRAM, where multiple PEs can read the text concurrently but each memory position can be written by a single PE at a time.

In practice, 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 PEs executing their operations. For example, let two PEs increase the variable *occ*, which initially is 0: the first PE increases it by 2 and second PE 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 PE 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 PEs write to different memory positions that are in the same cache line. Then, all other PEs that hold that cache line are forced to reload it, which results in additional and often unnecessary memory access (but not undefined behavior like race conditions) and should be avoided.

2.1.2 The Distributed Memory Model. In the distributed memory model, algorithms run on p distinct PEs 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. In practice, the number of PEs in this setting is often orders of magnitude greater than the number of PEs in shared memory, because we can use multiple shared memory machines that are connected by a network. Each PE has a unique *rank* in the range from 0 to $p-1$. The PEs 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 PEs communicate over the local memory. We do not consider this *hybrid* of distributed and shared memory explicitly. However, our implementations are based on MPI and therefore make use of this automatically.

We analyze our algorithms in the ***bulk-synchronous parallel (BSP)*** model [48]. Here, each algorithm is a sequence of *supersteps*, with each superstep being split into three phases: First, the PEs can perform any number of operations based on local data. We use t to denote the maximum time used by a PE. Second, the PEs can send data to other PEs (communication phase). Here, h is the maximum number of computer words communicated by any PE, and G is the running time required for the communication of one word. Last, all PEs wait until every PE has finished the first two phases. L is the time of this barrier synchronization. There is no synchronization between the first and second phase. PEs 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 2.

The *BSP cost* of an algorithm is the sum of all its supersteps, where the time of one superstep is $t + hG + L$. In addition to the BSP costs, we are also interested in the *communication volume* of

distributed memory algorithms, which we define as the total number of words communicated over all supersteps by all PEs.

2.1.3 COST of Parallelization. In addition to the models described above, we also consider the practical cost of parallelization by McSherry et al. [36]. Here, the **Configuration that Outperforms a Single Thread (*COST*)** of the parallel algorithm is the number of PEs 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 PEs needed for a parallel algorithm to be faster than any sequential algorithm on the same hardware.

The COST is similar to the concept of *speedup* as defined by Casanova et al. [6, page 10]: the smallest number of PEs that results in a speedup greater than 1 is the COST of the algorithm. However, we use the term *speedup* in its more common way by comparing the running time on more cores with the running time of the *same* algorithm on a single core.

2.1.4 External Memory Model. The *external memory model* [1] 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 computer words 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. We only use the *scan* operation: scanning N bits of data requires $\text{scan}(N) = \lceil N/(Bw) \rceil$ I/Os, where w is the size of a word in bits.

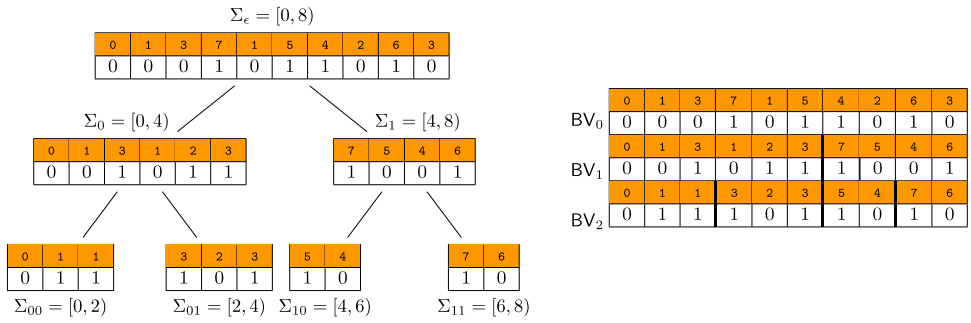
For *semi-external* algorithms (see, for example, References [27, 47]), we assume that we have random access to either the input or output—but not both, as then we would have an algorithm working in main memory. This relaxation allows for algorithms that cannot easily be expressed in the external memory 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)** [19] provides semi-external construction algorithms for many string data structures. Note that this model is not to be confused with the **semi-external memory (SEM)** model for graph algorithms, where all vertices are stored in main memory and all edges are stored in external memory [49].

2.2 The Wavelet Tree

Let T be a text of length n over an alphabet $[0, \sigma)$. The *wavelet tree* [21] 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 ith bit in BV_v is $\text{bit}(d(v), T_{[\ell,r)}[i])$, 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* [21] and the *level-wise* [33] wavelet tree. The pointer-based wavelet tree uses pointers to represent the tree structure; see Figure 3(a). Therefore, it requires space for $O(\sigma)$ pointers in addition to the bit vectors and succinct dictionary data structures for the binary rankand selectqueries.

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



(a) Pointer-based wavelet tree. (b) Level-wise wavelet tree.

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

for each level $\ell \in [0, \lceil \lg \sigma \rceil)$; see Figure 3(b). This retains the functionality from the pointer-based wavelet tree [4, 33], but it 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 1 (Fuentes-Sepúlveda et al. [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* $\mathrm{bit\_prefix}(\ell, T[i])$.

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 [7]. 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 selectfrom binary alphabets to alphabets of size σ. Answering these queries then requires $O(\lg \sigma)$ 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 References [8, 40]. Throughout this article, 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. [8]. It requires the same space as a wavelet tree and has the same asymptotic running time for access, rank, and selectqueries. But in practice it is often faster than a wavelet tree for rankand selectqueries [8], 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. [31], cannot simply be transformed to wavelet matrix construction algorithms.

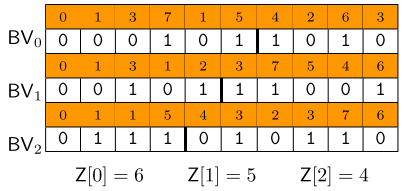
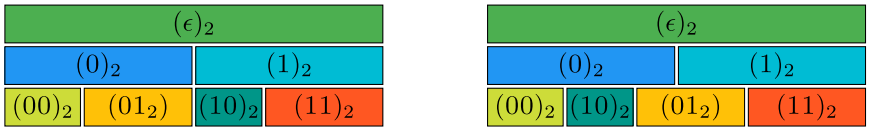


Fig. 4. The matrix wavelet of our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$. Again, the orange arrays contain the characters represented at the corresponding position in the bit vector and are not a part of the wavelet matrix. The thic lines are the visual representation of the number of zeros in each level.



(a) Intervals of a wavelet tree. (b) Intervals of a wavelet matrix.

Fig. 5. Positions of the intervals in a wavelet tree (a) and a wavelet matrix (b). The intervals are identified by the bit prefix that all characters represented by 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.

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 $\text{Z}[0, \lceil \lg \sigma \rceil)$ to store the number of zeros at each level ℓ in $\text{Z}[\ell]$. 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 $\text{BV}_{\ell-1}$. Then the position of its ℓth MSB in BV_ℓ depends on $\text{BV}_{\ell-1}[i]$ in the following way: if $\text{BV}_{\ell-1}[i] = 0$, then $\text{bit}(\ell, \alpha)$ is stored at position $rank_0(\text{BV}_{\ell-1}, i)$, otherwise $(\text{BV}_{\ell-1}[i] = 1)$, it is stored at position $\text{Z}[\ell - 1] + rank_1(\text{BV}_{\ell-1}, i)$; see Figure 4.

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 5. 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. *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*$(\text{bit\_prefix}(\ell, T[i]))$.

This leads to a naive wavelet matrix construction algorithm, where $\text{BV}_\ell[i] = \text{bit}(\ell, T'[i])$, where T' is T stably sorted using the reversed bit prefixes of length ℓ of the characters as key [8].

Table 1. Characteristics of the Texts Used in this Article: Name of the Text, Alphabet Size σ of the Whole Text, Alphabet Size σ_x for an x-character Prefix of the Text, Total Text Size n, and Empirical Entropy H_k for $k \in [0, 3]$

Name	σ	$\sigma_{2^{30}}$	$\sigma_{2^{31}}$	$\sigma_{2^{32}}$	n	H_0	H_1	H_2	H_3
CommonCrawl	243	242	242	242	196,885,192,752	6.19	4.49	2.52	2.08
DNA	4	4	4	4	218,281,833,486	1.99	1.97	1.96	1.95
Prot	26	26	26	26	50,143,206,617	4.21	4.20	4.19	4.17
Wiki	213	209	209	210	246,327,201,088	5.38	4.15	3.05	2.33
SA[2]	n	2^{30}	2^{31}	2^{32}	137,438,953,472	37 (= lg n)	0	0	0
RuWB[3]	29,263	26.682	27,302	28,402	9,232,978,762	10.93	—	—	—

2.4 Experimental Setup

To enable the best reproducibiliy of our results, we first describe the texts used in the experiments in Section 2.4.1, and then give a detailed description of our experimental environment in Section 2.4.2.

2.4.1 Text Corpora. One of the most used text corpora is the *Pizza & Chili* corpus, which is available at 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 article, we obtained different real world texts. We summarize important properties, including their empirical entropy [30], in Table 1. We will build wavelet trees using an *effective alphabet*, i.e., using characters from $[0, \sigma - 1]$ for alphabet size σ. Next, we give some details on the used texts and where to obtain them.

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 00600. 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 *1,000 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 `#ID`s are not continuous; not all `#ID`s are assigned, and some `#ID`s are separated in two parts, which is denoted by an _1 and _2 suffix. Throughout this article, 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

[2]Note that the entropy is not a good measure for large alphabets [18].
[3]Due to the size of the alphabet, we only computed H_0 for `RuWB`.

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 that were 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 article, we denote the Wikipedia dump by **`Wiki`**.

Russian Common Crawl (word-based). We transform the Common Crawl of Russian websites, available from http://web-language-models.s3-website-us-east-1.amazonaws.com/wmt16/deduped/ru.xz, in a way where we consider every distinct word a distinct single character in a larger alphabet. We denote the resulting file by **`RuWB`**.

Suffix Array of Common Crawl. We construct the suffix array [35] of `CommonCrawl`, which is a permutation of the positive integers smaller than the length of `CommonCrawl`. We store it as an array with 40 bits per integer and consider it a text where every character is distinct, which serves as a stress test for our distributed memory construction algorithm for the wavelet matrix. Throughout this article, we denote the suffix array of Common Crawl dump by **`SA`**.

2.4.2 Hardware and Software. 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:

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.

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.

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 (Small) or 48 (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.

For external memory our experiments (which we present in Section 8), 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 [11] development snapshot (26-09-2017). In our external memory experiments, we consider two different settings:

Ext.hdd four 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.

All 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

Table 2. Sequential Wavelet Tree and Wavelet Matrix Construction Algorithms in the Word RAM Model [23]

Reference	Time Complexity	Additional Space (bits)
naive	$O(n \lg \sigma)$	$n\lceil \lg \sigma \rceil$
seq.serial [44]	$O(n \lg \sigma)$	—
[9]	$O(n \lg \sigma)$	$O(\lg n \lg \sigma)$
[2]	$O(n \lg \sigma / \sqrt{\lg n})$	—
[38]	$O(n \lg \sigma / \sqrt{\lg n})$	—
[10]	$O(n \lg \sigma)$ (online)	$\alpha n \lceil \lg \sigma \rceil + O(1)$
[26]	$O(n \lg \sigma / \sqrt{\lg n})$	— (in practice: $\approx 4n \log \sigma$)
seq.pc, seq.pc.ss & seq.ps (**Section 5**)	$\boldsymbol{O(n \lg \sigma)}$	$\boldsymbol{\sigma \lceil \lg n \rceil}$

We use dashes ("—") to mark algorithms for which no analysis of the required additional space is conducted by the authors.

with in Section 6.5 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.

To be consistent with previous experimental evaluations, 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.

When measuring running times, we use the median of five executions. Memory usage is measured in an additional run in order not to influence the measured running times. We always set a time limit of 2 hour for these six executions such that in rare cases some slow algorithms did not finish for very large inputs.

3 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 $O(n \lg \sigma)$ 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 [8] and the algorithms we present in this article. Still, some wavelet tree construction algorithms can easily be modified to compute the wavelet matrix instead of the wavelet tree.

3.1 Sequential Wavelet Tree Construction Algorithms

An overview of the sequential wavelet tree construction algorithms presented in this section is given in Table 2, 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. For this reason, we do not consider theoretical algorithms such as Tischler [46] and Claude et al. [9].

Computing the wavelet tree for a text of size n over an alphabet of size σ *naively* in time $O(n \lg \sigma)$ is simple [21]. To compute 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*, then we do not require an additional copy of the text. To obtain a total running time of $O(n \lg \sigma)$, we have to sort the text

Table 3. Shared Memory Parallel Wavelet Tree and Wavelet Matrix Construction Algorithms

Algorithm	Work (left) and Time (right) Complexity		Additional Space (Bit)
par.level [44]	$O(n \lg \sigma)$	$O(\lg n \lg \sigma)$	$O(n \lg n)$
par.sort [44][4]	$O(n \lg \lg n \lg \sigma)$	$O(\lg n \lg \sigma)$	$O(n \lg n \lg \sigma)$
par.rec [31]	$O(n \lg \sigma)$	$O(\lg n \lg \sigma)$	$O(p \lg n \lg \sigma)$
par.dd [17]	$O(\sigma n / \lg n + n \lg \sigma)$	$O(\lg n \lg \sigma)$	$O(n \lg \sigma)$
par.pc & par.pc.ss (**Section 6.1**)[5]	$O(n \lg \sigma)$	$O(\frac{n}{p} \lg \sigma)$	$\sigma \lceil \lg n \rceil$
par.ps (**Section 6.2**)	$O(n \lg \sigma + p\sigma)$	$O(\frac{n}{p} \lg \sigma + \sigma + \lg p)$	$n \lceil \lg \sigma \rceil + p\sigma \lceil \lg n \rceil$
par.dd.pc & par.dd.ps (**Section 6.3**)	$\boldsymbol{O(n \lg \sigma + p\sigma)}$	$\boldsymbol{O(\frac{n}{p} \lg \sigma + \sigma + \lg p)}$	$\boldsymbol{n \lceil \lg \sigma \rceil + p\sigma \lceil \lg n \rceil}$

We use dashes ("—") to mark algorithms for which no analysis of the required additional space is conducted by the authors.

in linear time. To this end, we use *stable* counting sort, which requires additional $n\lceil \lg \sigma \rceil$ bits of space [43, p. 300].

Shun [44] 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.

Babenko et al. [2], Munro et al. [38] independently improved the running time for wavelet tree construction algorithms to $O(n \lg \sigma / \sqrt{\lg n})$ by using word packing and broadword programming techniques, i.e., applying operations to multiple integers that all fit into one computer word at the same time.

In the context of these two theoretically fastest algorithms, we also mention Kaneta's [26] recent algorithms that bridge the gap between theory and practice and are based on the ones by Babenko et al. [2], Munro et al. [38]. Their main contribution is a practical implementation of the broadword programming techniques using new CPU instructions. Unfortunately, their implementation is not publicly available.

3.2 Parallel Wavelet Tree Construction Algorithms

Now, we have a look at parallel wavelet tree construction algorithms. We give an overview of the algorithms in Table 3. We do not describe three primarily theoretical algorithms by Shun [45], as we focus on practical results in this article.

Shun [44] 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 4.

(1) Shun's [44] first parallel wavelet tree construction algorithm (par.level) computes the histogram of each level in parallel. To this end, each PE 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 PE are represented in the level. Then, the text is scanned once

[4]By computing the wavelet tree level-by-level, instead of all levels in parallel, the additional space can be reduced to $O(n \lg n)$ bits in in par.sort [44]. However, this increases the time to $O(\lg n \lg \sigma)$. The work remains the same in both cases.

[5]We cannot use more than $\lceil \lg n \rceil$ processing elements to parallelize the algorithm that we present in Section 6.1.

more—again in parallel—and the bit vector is created. Note that this approach requires two scans of the text.

(2) Shun's [44] 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.

Fuentes-Sepúlveda et al. [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 PE computes a partial wavelet tree for a slice of the text. The slices are non-overlapping and consecutive, 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 6.3. 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 [17], par.dd uses a slightly adopted version of Shun's [44] wavelet tree construction algorithm seq.serial; see Section 3.1. Labeit et al. [31], too, present a domain decomposition wavelet tree construction algorithm that uses a different merge function compared with Fuentes-Sepúlveda et al.'s [17] one.

In addition to their domain decomposition wavelet tree construction algorithm, Labeit et al. [31] 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 $s : \Sigma \to \texttt{bool}$, generates two texts $T_{\texttt{true}}$ and $T_{\texttt{false}}$ such that for all $i \in [0..|\{k \in [0..n) : s(k) = \alpha\}|)$ and $\alpha \in \{\texttt{true}, \texttt{false}\}$ we have $T_\alpha[i] = T[j]$, where j is the unique position that fulfills (1) $s(T[j]) = \alpha$ and (2) $|\{k \in [0..j) : s(T[k]) = \alpha\}| = i$. 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

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

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 PEs as follows: At the first level, we use all p PEs to split the text and compute the bit vector; then, on the second level, we use a number of PEs proportional to the sizes of the two results of the previous split operations (but at least one PE) to compute the bit vectors of the intervals *and* split the considered text into two. Hence, all PEs are used throughout the computation.

4 BOTTOM-UP COMPUTATION

The new 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.

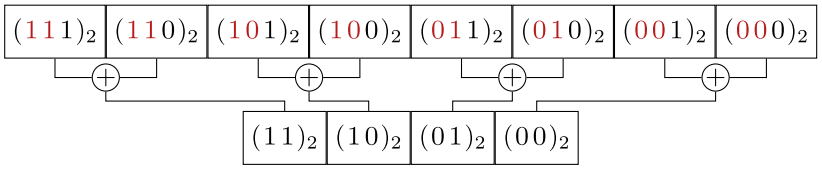


Fig. 6. Computing histogram of bit prefixes of length two from histogram of length three bit prefixes. Common prefixes are highlighted in dark red.

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 6.

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.

In the sequential setting, the computation of all histograms and starting positions of the intervals requires $O(\sigma\lg\sigma)$ time, which is dominated by the $O(n\lg\sigma)$ 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., [17, 44].

5 SEQUENTIAL CONSTRUCTION

Now, we employ the bottom-up construction in two different wavelet tree construction algorithms. First, in Section 5.1, 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 6, where random access on bits is problematic, we also describe a second variant in Section 5.2, where the access pattern to the bit vectors is a scan from left to right. All wavelet tree construction algorithms that we present in this section can easily be adopted to compute the wavelet matrix instead (see Section 5.3).

5.1 Prefix Counting

Our first wavelet tree construction algorithm (*seq.pc*; see Algorithm 1) starts with the computation of the initial histogram $\text{Hist}[0,\sigma)$ of the text (line 2). In addition, the first level of the wavelet tree is computed, as it contains the most significant 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 $O(n)$ 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.

Algorithm 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 (basis for all histograms).
3  |  BV_0[i] = bit(0, T[i])      // Fill first level's bit vector (MSBs 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.
```

We compute these histograms as described in Section 4 by ignoring the last bit of the considered bit prefixes. As described before, we can do so in $O(\sigma)$ 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 Borders in line 8). Again, this requires $O(\sigma)$ 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, 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 $O(n)$ 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 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 5.1. *Algorithm seq.pc computes the wavelet tree of a text of length n over an alphabet of size σ in $O(n \lg \sigma)$ 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.*

5.2 Prefix Sorting

Our next wavelet tree construction algorithm (*seq.ps*, Algorithm 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 $O(n)$ 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 [43, p. 300]. This leads to the following running time and memory requirements:

Lemma 5.2. *Algorithm seq.ps computes the wavelet tree of a text of length n over an alphabet of size σ in $O(n \lg \sigma)$ 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 PE. We describe the parallel version of seq.ps in Section 6.2.

Algorithm 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 (basis for all histograms).
3  |  BV_0[i] = bit(0, T[i])      // Fill first level's bit vector (MSBs 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.
```

5.3 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 ℓth 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 5.

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. However, 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 Borders , see line 8 in Algorithm 1 (seq.pc and seq.pc.ss) and line 8 in Algorithm 2 (seq.ps), since we store the starting positions of the intervals in 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 Borders$[\rho_\ell[i]]$ = Borders$[\rho_\ell[i-1]]$ + 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.

5.4 Experimental Evaluation

We implemented all our sequential wavelet tree and wavelet matrix construction algorithms. The code is available at www.github.com/kurpicz/pwm. We use the hardware setup described in Section 2.4.2 and conducted the experiments on the Big nodes. Our inputs are prefixes of the texts described in Section 2.4.1.

We also include the following wavelet tree construction algorithms (see Section 3.1):

(1) seq.naive is the naive wavelet tree construction algorithm based on sorting implemented by us,
(2) seq.serial[7] was the previously fastest sequential wavelet tree construction algorithm [44], and
(3) seq.sdsl[8] is part of the frequently used **Succinct Data Structure Library (SDSL)** [19].

One wavelet tree (and wavelet matrix) construction algorithm that we could not include here is that of Kaneta [26], 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.

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. Since both types of construction algorithms mostly differ in the computation of the borders, which contributes insignificantly to the running time and memory peak of the algorithm, we moved the plots for the wavelet matrix construction algorithms to Section A. Only when there are larger differences or algorithms only working for the wavelet matrix (cf., Section 7), we show plots for both types directly in the evaluation.

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 7 and 33. Missing data is either due to exceeding the time limit or exhausting the available main memory.

First, we mention that the throughput does not differ a lot for the different inputs, as we show the number of bits computed per second, which is independent of the alphabet size and hence 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

[7]Downloaded from https://people.csail.mit.edu/jshun/wavelet.tar.
[8]Downloaded from https://github.com/simongog/sdsl-lite/.

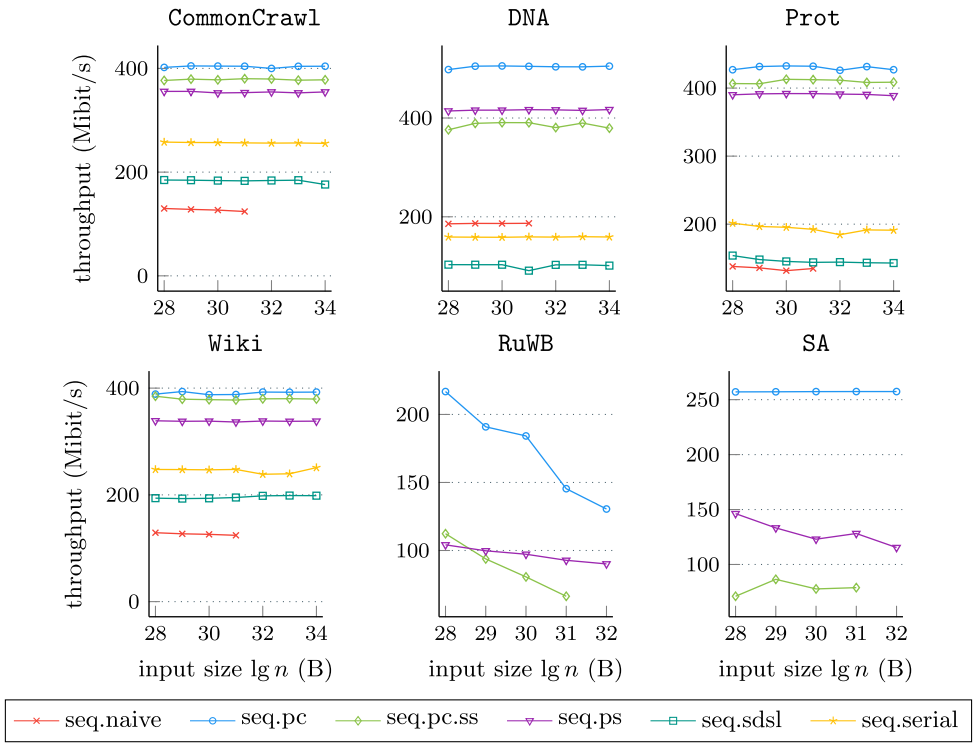


Fig. 7. Throughput of the sequential wavelet *tree* construction algorithms.

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.

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. It is notable that the only other existing algorithm for matrices (seq.sdsl.wm) is almost by an order of magnitude slower than its tree counterpart, whereas our algorithms achieve almost exactly the same throughput for both structures.

We tested only our algorithms on the inputs with large alphabets, because only our implementations are able to handle these texts and all other implementations cannot handle 40-bit integers as input. Here, we see that seq.pc is still the fastest construction algorithm. However, on RuWB its throughput decreases when the text size is increased. We cannot explain this behavior, which comes even more surprising, because it does not happen for SA where the alphabet size increases even more (for longer texts) as $\sigma = n$. Next, seq.ps is surprisingly faster than seq.pc.ss, which is not the case when using text over smaller alphabets. Overall, the throughput of our algorithms is close to 50% smaller. This is because the histogram (as implemented) does not fit into the cache anymore, making the construction more expensive overall.

Memory Peak. The memory peaks of the above algorithms are presented in Figure 8 is normalized by the input size. Whenever we measure the memory, we also measure everything that has to be in main memory for the computation. For example, in external memory, input and output is not measured, as it remains on disk. Since we compute wavelet trees, the output can be smaller than

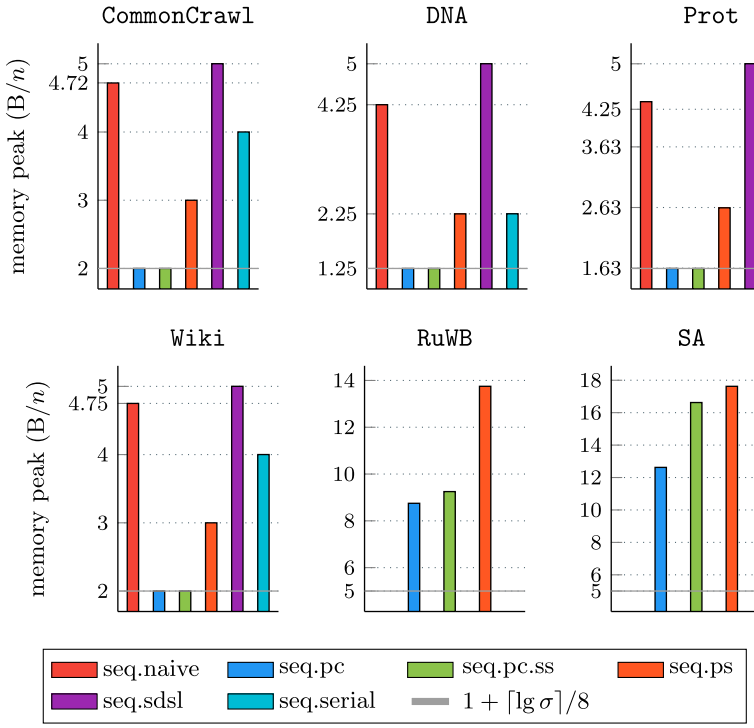


Fig. 8. Memory peaks of sequential wavelet *tree* construction algorithms for $n = 2^{31}$. We also depict is the memory required to store the text and the wavelet tree ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character).

the input (on texts with smaller alphabet). We only give the memory peaks for inputs of size 2 GiB, because

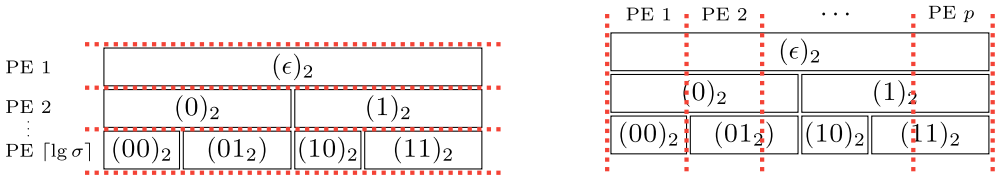
(1) since we normalize the memory peaks, they are independent of the input size and
(2) this is the maximum input size that all algorithms can process given the time and memory constraints described earlier.

We do not consider texts with large alphabets as none of the algorithms can handle those as implemented. Throughout this article, we measure the required memory by overwriting `malloc`. To this end, we use *malloc_count*,[9] which keeps track of *all* heap memory allocations.

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.

[9]Source code available at https://github.com/bingmann/malloc_count/.



(a) Each PE computes one level.

(b) Each PE computes the same slice of the bit vector on all levels.

Fig. 9. Our two simple methods to parallelize the wavelet tree and wavelet matrix construction.

As mentioned before, when computing the wavelet trees for RuWB and SA, the histogram does not fit into the main memory anymore. We can also see this in the memory peaks of the construction algorithms. It comes with no surprise that all algorithms require more memory when computing the wavelet tree or wavelet matrix for these texts than for texts over a small alphabet. Here, we can also see how an increasing alphabet size influences the memory peak, as the peaks are higher for SA than for RuWB.

The memory peak of the wavelet matrix construction algorithms have exactly the same memory peaks as the corresponding ones for trees.

6 SHARED MEMORY CONSTRUCTION

Now, we look at shared memory parallel wavelet tree construction algorithms. First, in Section 6.1, we describe a naive parallelization of seq.pc that only scales up to $\lceil \lg \sigma \rceil$ PEs. Therefore, we also parallelize seq.ps in Section 6.2, which scales better. In Section 6.3, we look at the meta-approach *domain decomposition* [17, 31], 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 6.4). Finally, in Section 6.5, we give an experimental evaluation of these algorithms.

Practical Parallel Histogram Computation. Before we look at 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.

Let T be a text of length n over an alphabet of size σ. We first split the text into p slices of equal size; let $\mathsf{T}_i[0..n/p]$ denote the slice for PE $i \in [0, p)$. Every processor i then computes the *local* histogram $\mathrm{Hist}_i[0, \sigma)$ for its slice in time $O(n/p + \sigma)$ and total work $O(n + p\sigma)$. Computing the *global* histogram can then be done by summing over the local counts; using parallel prefix sums [25], this takes $O(\sigma + \lg p)$ time and $O(p\sigma)$ work. This results in a total time $O(n/p + \sigma + \log p)$ and work $O(n + p\sigma)$, which is work efficient only for $p \leq n/\sigma$. Nonetheless, this approach works well for most practical scenarios with small alphabets; in theory, faster approaches for histogram computation are conceivable, e.g., using parallel sorting.

6.1 Parallel Prefix Counting

It is embarrassingly easy to parallelize seq.pc (Algorithm 1) such that each PE computes one level of the wavelet tree; see Figure 9(a). 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 ℓth 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*.

Lemma 6.1. *Algorithm par.pc computes the wavelet tree in $O(n + \sigma + \lg p)$ time with $O(n \lg \sigma + p\sigma)$ 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 $\lceil \lg \sigma \rceil$ PEs. To use more PEs, instead of parallelizing level-wise, we could do the following. Each of the p PEs 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 PE 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 PEs 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.

Similar to the sequential version of this algorithm (seq.pc), we can compute the wavelet tree in a single scan of the text, given the histograms. In the parallel setting, the main difference is that now all PEs are used to compute the histograms in together, whereas before each PE computed only the histogram required for its level.

6.2 Parallel Prefix Sorting

Instead of having each PE write randomly to each bit vector, we want each PE to be responsible for the same slice on each level of the wavelet tree; see Figure 9(b). Those slices have size $\Theta(n/p)$. To this end, we parallelize seq.ps (Algorithm 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 PE 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 [17, 31], which we discuss in Section 6.3. 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) works as follows: First, each of the p PEs 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 $O(n/p + \sigma + \lg p)$ time, $O(n + p\sigma)$ 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 $O(\sigma/p)$.

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 PE can fill BV_ℓ accordingly (line 13). This leads to the following lemma.

Lemma 6.2. *Algorithm par.ps computes the wavelet tree of a text of length n over an alphabet of size σ in $O((n/p) \lg \sigma + \sigma + \lg p)$ time and $O(n \lg \sigma + p\sigma))$ work requiring $n\lceil \lg \sigma \rceil + p\sigma\lceil \lg n \rceil$ bits of space in addition to the input and output.*

This algorithm can efficiently use up to $p \leq n \lg \sigma / \sigma$ PEs. Using that many PEs yields $O(n \lg \sigma)$ work with $O(\lg \sigma\ (\sigma + \lg n))$ time. Employing more PEs 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 References [2, 26, 38].

Using sorting for the parallel construction of wavelet trees has already been considered by Shun [44] (*par.sort*). We gave a short description of their approach in Section 3.2. The main difference between their algorithm and ours is that we construct the wavelet tree bottom-up whereas they construct the wavelet 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 6.5.

Algorithm 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.

parfor $c = 0$ **to** $p - 1$ **do**
  for $i = c\frac{n}{p}$ **to** $(c+1)\frac{n}{p} - 1$ **do**
    $\mathrm{Hist}_c[T[i]]$++ // Compute histogram of the local part of the text.
    $\mathrm{BV}_0[i] = \mathrm{bit}(0, T[i])$ // Fill first level's bit vector in parallel.
$\mathrm{Borders}_c$ = Parallel zero-based prefix sum on Hist_c // Global starting positions.
for $\ell = \lceil \lg \sigma \rceil - 1$ **to** 1 **do** // For each level (from the last to the second).
  parfor $c = 0$ **to** $p - 1$ **do** // Get histogram for local part and current level.
    for $i = 0$ **to** $2^\ell - 1$ **do**
      $\mathrm{Borders}'_c[i] = \mathrm{Borders}_c[2^{\lceil \lg \sigma \rceil - \ell} i]$
  T_{sorted} = parallel CountingSort(T, $\mathrm{Borders}'_c$, ℓ) // Use starting positions to sort text.
  parfor $c = 0$ **to** $p - 1$ **do**
    for $i = c\frac{n}{p}$ **to** $(c+1)\frac{n}{p}$ **do** // Fill bit vector from using the sorted text.
      $\mathrm{BV}_\ell[i] = \mathrm{bit}(\ell, T_{\mathrm{sorted}}[i])$

6.3 Domain Decomposition

The *domain decomposition* [17, 31] is a popular meta-approach for parallel wavelet tree construction. Here, each PE 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 5), to compute the partial wavelet trees. The final wavelet tree is then computed by merging all partial wavelet trees in parallel, which is more like a concatenation of intervals in the bit vectors. 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. The main difference between the domain decomposition and our parallel prefix sorting approach is that during the former we decompose the text and during the latter we *decompose* the bit vectors. To be more precise, in the parallel prefix sorting approach, each PE fills one interval of *each* bit vector, which is the same on all levels. Whereas during the domain decomposition, each PE computes all bit vectors for the same (interval) of the text.

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 10. 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 PE writes its intervals at the corresponding positions.

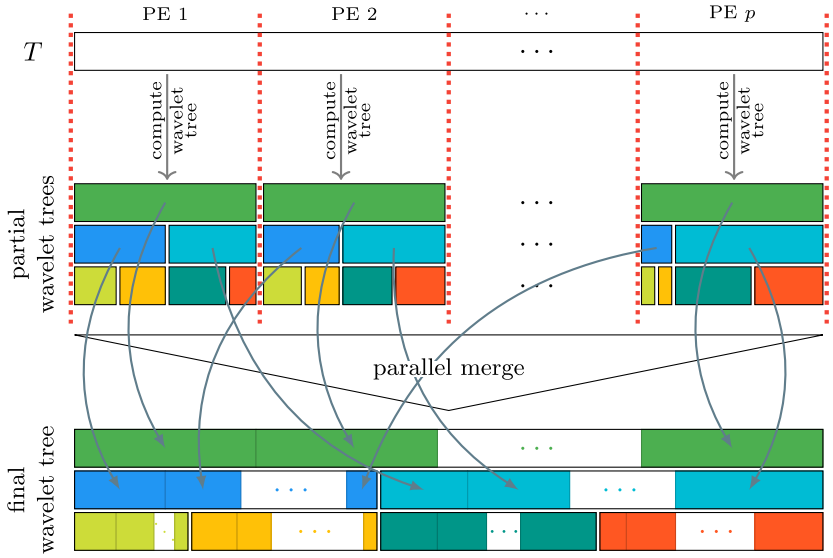


Fig. 10. Each PE considers a part of the text and computes a *partial* wavelet tree for that tree, in parallel. Then, all partial wavelet trees are merged—again, in parallel. To merge the partial wavelet trees, we concatenate the intervals representing the same characters on each level. To highlight 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. Note that the running time of the parallel merging is dominated by the computation of the partial wavelet trees, as we only read each bit once and write each bit once. This leads to the following lemma.

Lemma 6.3. *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 $O((n/p)\lg\sigma + \sigma + \lg p)$ time and $O(n\lg\sigma + p\sigma)$ work requiring $n\lceil\lg\sigma\rceil + p\sigma\lceil\lg n\rceil$ bits of space in addition to the input and output.*

6.4 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 6.1) in the same fashion we adapted its sequential counterpart in Section 5.3—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 PE.

To adapt par.ps (Section 6.2), we have to adjust the computation of the starting positions, because we lose the tree structure when we compute the wavelet matrix; see Algorithm 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 PE 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, which is the main difference between the wavelet tree and wavelet matrix construction algorithms, we get:

Lemma 6.4. *The wavelet-matrix-constructing counterpart of par.ps, par.ps.wm, computes the wavelet matrix in $O(\lg\sigma\,(n/p+\sigma+\lg p))$ time and $O(\lg\sigma(n+p\sigma))$ work to compute the wavelet matrix for a text of size n over an alphabet of size σ. The space requirements do not change compared to par.ps.*

Note that the limitations on the number of PEs that can efficiently be used remains the same as in the wavelet tree construction algorithms. The parallel wavelet tree construction algorithms using domain decomposition that we described in Section 6.3 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 10, 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.

Algorithm 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 − 1 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.
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]
11     T_sorted = CountingSort(T, Borders) parfor c = 0 to p − 1 do
12         for i = c n/p to (c + 1) n/p do
13             BV_ℓ[i] = bit(ℓ, T_sorted[i])
```

6.5 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.github.com/kurpicz/pwm. We used the hardware setup described in Section 2.4.2, conducted the experiments on a single Big node, and use (prefixes of) the text described in Section 2.4.1 as inputs.

We compare our new algorithms with all algorithms described in prior work for which an implementation (that we are aware of) exists: par.dd [45], par.level [44], par.sort [44], and par.rec [31].[10] A brief description of these algorithms can be found in Section 3.2.

[10] All downloaded from https://github.com/jlabeit/wavelet-suffix-fm-index.

As our algorithms are highly optimized for byte alphabets (especially the computation of borders and histograms), we do not consider inputs with large alphabets in this evaluation. Note that none of the other algorithms can handle 40-bit integers as input either. However, some of our distributed memory construction algorithms can handle this type of input. Therefore, we refer to Section 7 for parallel construction algorithms for inputs with large alphabet.

Construction Time. First, we report the construction times of the wavelet tree and wavelet matrix construction algorithms in a weak scaling experiment. In our description of the results, we focus on the results of our wavelet tree construction algorithms, which are depicted in Figure 11, as running times and scalability of our wavelet matrix construction algorithms are nearly identical; see Figure 35. 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, 512, or 1,024 MiB per PE. Therefore, in the following, we only discuss data from experiments where we use inputs of size 1,024 MiB.

We next look at our slowest parallel algorithms par.pc, par.pc.ss. While they do not scale up to 48 PEs, they do scale up to $\lceil \lg \sigma \rceil$ PEs (two for `DNA`, four for `Prot`, and eight for `CommonCrawl` and `Wiki`), which is as expected and matches the theoretical analysis. Our last algorithm that does not scale well is par.ps, which is due to the overhead of stable parallel sorting.

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 almost all inputs. Only on `Wiki` par.dd.pc.ss is slightly 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 5.4). There, seq.pc is faster on all inputs, and we cannot give a clear reason why seq.pc.ss is faster when used as part of a domain decomposition.

Now, let us focus on the throughput using one and 48 PEs on 1,024 MiB input per PE. We report detailed results in Table 4. 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 PEs. This is consistent with the results previously reported [16]. 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 (see Section 2.1.3 for a description) of *all* our parallel wavelet tree and wavelet matrix construction algorithms is 2, as when using two PEs, all parallel versions are faster than the fastest sequential one (which is the same as executing our parallel algorithm using only one PE). The previously fastest algorithm par.rec has COST of 4 on `CommonCrawl` and `Wiki`, and a COST of 8 on `DNA`.

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 12. There, we only give the results for algorithms using 48 PEs and processing 256 MiB input per PE. This is because

(1) the memory requirements increase when increasing the number of PEs, making this the most interesting case, and

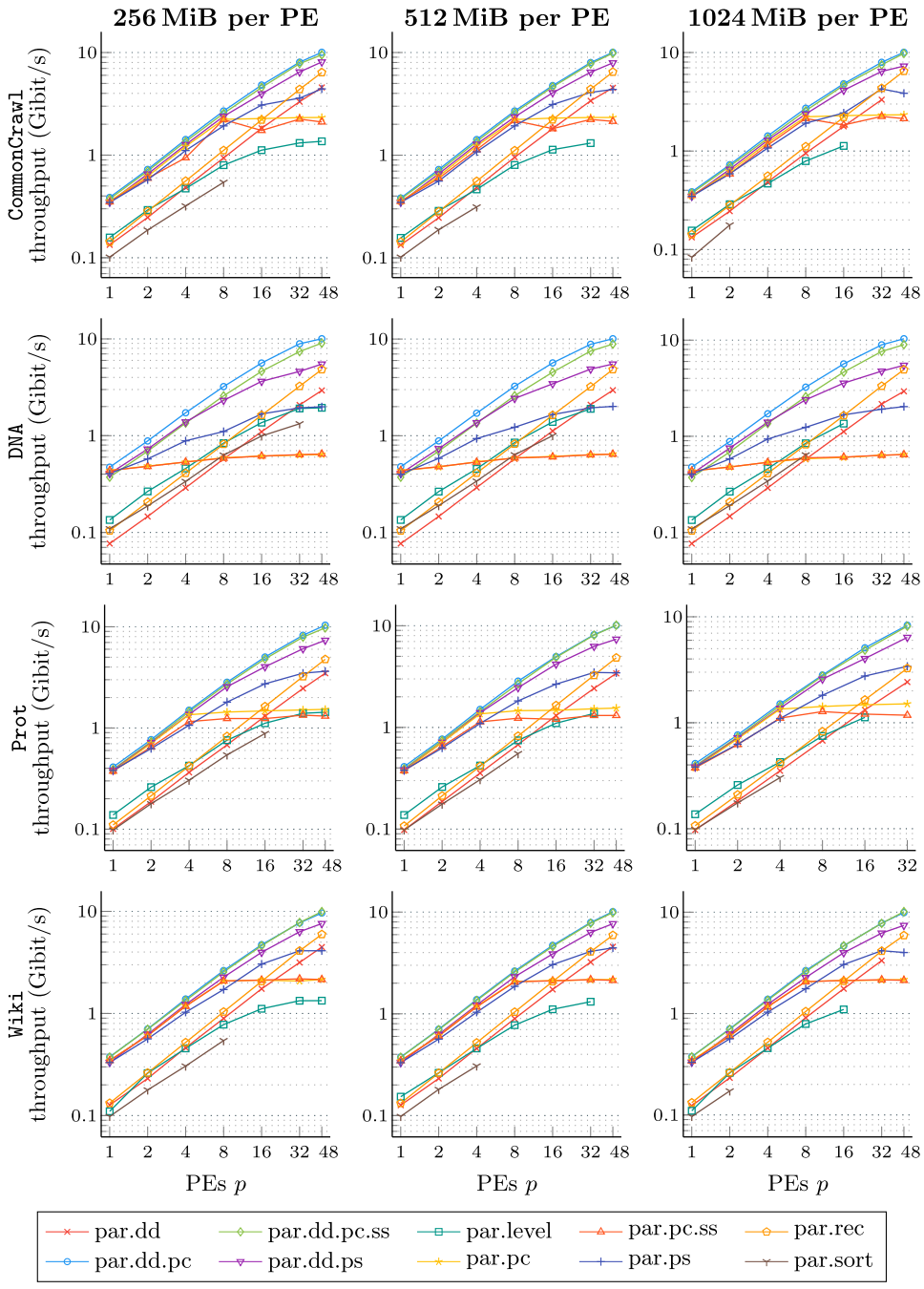


Fig. 11. Weak scaling parallel wavelet *tree* construction experiments.

Table 4. Throughput (Gibits/s) of the Wavelet Tree and Wavelet Matrix Construction Algorithms in Our Weak Scaling Experiment when Using 1 (t_1) or 48 (t_{48}) PEs and 1,024 MiB Input per PE

	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

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.

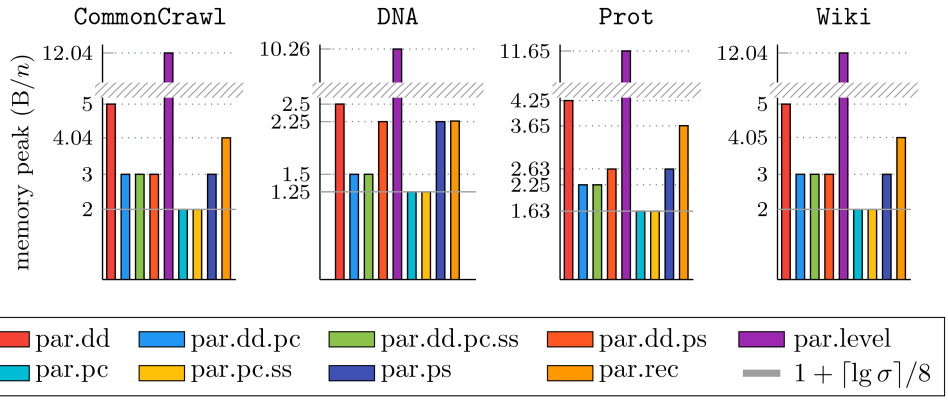


Fig. 12. Memory peaks of the parallel wavelet *tree* construction algorithms using 48 PEs and 256 MiB input per PE. We use small inputs to have results for par.dd, still we could not include par.sort. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

(2) due to their memory consumption, not all algorithms (par.dd, par.sort, and par.level) finish the experiment when using 48 PEs and larger input sizes. Using only 256 MiB input per PE 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

Table 5. Overview of the BSP Costs and Communication Volumes of the Algorithms Described in Section 7, Listing the Wavelet Tree (WT) and Matrix (WM) Versions of the Bucket Sort Algorithm Separately

Algorithms	BSP costs in $O(\cdot)$	Comm. Volume in $O(\cdot)$ words
par.dd.pc(.wm)	$W_{localWT}(n/p) + \sigma \lg p + G(\sigma \lg p + (n/p) \lg \sigma / \lg n) + L(\lg p + \lg \sigma)$	$n \lg \sigma / \lg n + \sigma p$
dist.bsort.wt	$(n/p) \lg \sigma + \sigma \lg p + G(\sigma \lg p + (n/p) \lg \sigma) + L \lg \sigma \lg p$	$(n \lg^2 \sigma) / \lg n + \sigma p$
dist.bsort.wm	$(n/p) \lg \sigma + \lg \sigma \lg p + G(\lg \sigma \lg p + (n/p) \lg \sigma) + L \lg \sigma \lg p$	$(n \lg^2 \sigma) / \lg n + p \lg \sigma$
dist.split.wt(.wm)	$(n/p) \lg \sigma + \sigma \lg p + G(\sigma \lg p + (n/p) \lg \sigma) + L(\lg \sigma + \lg p + \lg(\sigma) \lg(p))$	$(n \lg^2 \sigma) / \lg n + \sigma p$

The asymptotic amount of required local memory is $O((n/p) \lg \sigma + \sigma \lg n)$ for all algorithms except for Bucket Sort (WM), which does not require the $O(\sigma)$ factor.

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 `Prot`. 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× 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×.

7 DISTRIBUTED MEMORY CONSTRUCTION

Now, we present three distributed memory wavelet tree construction algorithms. Our algorithms are based on the ideas presented in the previous section. In the distributed setting, we assume that we have an input text T that is distributed among all PEs such that PE i initially holds part $T_i = T[i\lceil n/p \rceil .. (i+1)\lceil n/p \rceil - 1]$ of length $\lceil n/p \rceil$ and, after computation, holds the ith part of each level's bit vector of length $\lceil n/p \rceil$. In the case that n is not a multiple of p, PE $p-1$ may have a shorter slice of the text, which we consider an implementation detail. Furthermore, we show how to alter each algorithm to compute instead the wavelet matrix. Table 5 gives an overview of the BSP costs and communication volumes of our algorithms.

Prefix and All-to-All Sums. A common problem in distributed computing is finding the *prefix sum* of a vector $(x_0, x_1, \ldots, x_{p-1})$, which is distributed so that PE i knows only x_i. The task of prefix summing is to compute the vector of sums $X_i = \sum_{j=0}^{i-1} x_j$ so that after the operation, PE i knows the sum X_i of values held by PEs with lower rank. Similar to a PRAM-optimal text book algorithm [25, section 2.1.1], in the distributed setting, we can use a binary merge tree to compute the prefix sum in time $O(\lg p)$ and $O(\lg p)$ supersteps with any PE sending $O(1)$ words in each superstep, resulting in a communication volume of $O(p)$ words. Assuming that each x_i comes from a universe of size n, the local memory required is $O(\lg n)$ bits.

A related problem is the computation of the sum $\sum_{j=0}^{p-1} x_j$ of all local values and broadcasting it back to all PEs. The general case is known as an ***all-to-all reduction (AllReduce)*** and supports any associative reduction operation. In our case, where the reduction of two elements is their sum, we henceforth use the term *all-to-all sum*, which can also be computed using a binary merge tree [42] and thus has the same asymptotic BSP costs as prefix summing.

Lemma 7.1. *Using p PEs, we can compute the zero-based prefix sums and the all-to-all sum of a distributed vector of p words from a universe of n words with BSP costs of $O(\lg p + G \lg p + L \lg p)$, a communication volume of $O(p)$ words and using $O(\lg n)$ bits of local memory.*

Input partitioning and consequences. Because of the partitioning, some characters of the input alphabet Σ may never occur in some parts of T known to some *PEs*. For this reason, it is non-trivial to find the histogram and the effective alphabet $[0 \,..\, \sigma - 1]$ in the distributed model. We give a description of how we can do this in the following section, but we note that when the input alphabet is larger than the effective alphabet ($|\Sigma| > \sigma$), this may asymptotically dominate the wavelet tree and matrix construction algorithms that we describe in the sections following that. However, this step can also be considered optional in that a larger alphabet would merely result in more levels being constructed. Therefore, when describing the construction algorithms later, we will assume that the input is given already in its effective transformation, and we use σ to denote the size of the effective alphabet.

7.1 Histogram, Effective Alphabet, and Borders

We first describe how we compute the histogram and effective alphabet of an input text T in distributed memory. Each PE first computes a *local* histogram by scanning its local text part once. This takes $O(n/p + |\Sigma|)$ time (accounting also for initialization of an empty histogram) and requires $|\Sigma| \lceil \lg n \rceil$ bits of local memory. In a second step, we compute the all-to-all sum of the $|\Sigma|$ words containing the local occurrence counts for each symbol in Σ. In $O(|\Sigma|)$ subsequent time, we can reduce Σ to the effective alphabet $[0 \,..\, \sigma - 1]$ and store the effective alphabet ranks of all symbols using $|\Sigma| \lceil \lg \sigma \rceil$ bits on all PEs. We can then override T by its effective transformation T' in one more scan taking $O(n/p)$ time.

Lemma 7.2. *Using p PEs, we can compute the histogram, effective alphabet Σ' and the effective transformation of a text $T \in \Sigma^n$ with BSP costs of $O(n/p + |\Sigma| + \lg p + G|\Sigma| \lg p + L \lg p)$, a communication volume of $O(|\Sigma| p)$ words and using $|\Sigma|(\lceil \lg n \rceil + \lceil \lg \sigma \rceil)$ bits of local memory.*

Once the histogram of the entire T is known to each PE, they can compute the Borders array (Section 5.1) for any level of the wavelet tree of T in time $O(\sigma)$ and store it in $\sigma \lceil \lg n \rceil$ bits of memory.

7.2 Domain Decomposition

The domain decomposition can be adapted from shared (Section 6.3) to distributed memory in a very straightforward manner. As in the shared memory setting, in a first step, each PE i first constructs the entire wavelet tree for its input part T_i using a sequential algorithm. Let $t_{localWT}(n/p)$ denote the time and $m_{localWT}(n/p)$ denote the number of bits of memory that the local construction requires.

We then need to merge the local wavelet trees by concatenating the bit vectors corresponding to the same nodes in the global wavelet tree. In distributed memory, this requires communication. For this, we need to determine the receiving PE for every bit in BV_ℓ on every level ℓ. Since every PE will receive $\lceil n/p \rceil$ bits of BV_ℓ, the receiving PE for a bit at position j has rank $j/\lceil n/p \rceil$. To find the position j of a bit in the global bit vector BV_ℓ locally on PE i, we can use the Borders array to determine the starting position of a node's bit vector within BV_ℓ, and zero-based prefix summing to find how many bits the PEs with rank less than i have computed. The zero-based prefix sum takes $O(\sigma \lg p)$ time in $O(\lg p)$ BSP supersteps. Because the local wavelet tree has been constructed for a local text only, there are at most two different receiving PEs for the local bits of any wavelet tree node (on all PEs). This allows us to group them for efficient communication in practice. On any PE, the total number of words sent is bounded by the number $\lceil n/p \rceil \lceil \lg \sigma \rceil$ of bits in the local wavelet

tree. Since we can pack $O(\lg n)$ bits into one computer word, we send $O((n \lg \sigma) / (p \lg n))$ words. However, in case $n/(p \lg n)$ is not integer, each PE may need to send an additional incomplete word for every node, resulting in a total of $O(p\sigma)$ incomplete words in the worst case. We perform the merge level by level, so we need a barrier synchronization on each level, amounting to $O(\lg \sigma)$ total BSP barriers for the merge. We require $\lceil n/p \rceil \lceil \lg \sigma \rceil$ bits of local memory to store the wavelet tree and less than $2\sigma \lceil \lg n \rceil$ bits for storing the Borders arrays for all levels. Furthermore, we require a send/receive buffer of $\lceil n/p \rceil$ bits. In summary, we get the following lemma.

Lemma 7.3. *Using domain decomposition with p PEs, we can construct the wavelet tree for $T \in \Sigma^n$ with BSP costs of $t_{localWT}(n/p) + O(\sigma \lg p + G(\sigma \lg p + (n \lg \sigma)/(p \lg n)) + L(\lg p + \lg \sigma))$, a communication volume of $O((n \lg \sigma) / \lg n + \sigma p)$ words and using at most $\max\{m_{localWT}(n/p), \lceil n/p \rceil (\lceil \lg \sigma \rceil + 1) + 2\sigma \lceil \lg n \rceil\}$ bits of local memory.*

7.3 Prefix Sorting

Our parallel shared memory prefix sorting algorithm from Section 6.2 can be used in distributed memory simply by applying any distributed stable sorting algorithm to sort the text's characters according to their bit prefixes. The most natural sorting algorithm for this scenario is *bucket sorting*. In the sort operation on level ℓ, there are $2^{\ell+1}$ distinct sort keys: the $(\ell + 1)$-bit prefixes of the symbols. This equals the number of nodes on level $\ell + 1$. Thus, we allocate $2^{\ell+1}$ buckets on every PE and append each character to the corresponding local bucket in a distributed left-to-right scan of the text, during which we can also compute the bit vector for the respective level of wavelet tree. We concatenate the buckets in the same fashion as the nodes of one level in the domain decomposition's merge, which produces the stably sorted distributed representation of the text's characters for the next level. We repeat these steps until all levels have been processed.

The scans to construct the bit vectors and to fill the buckets require $O((n/p) \lg \sigma)$ time. To allocate the buckets with their proper sizes in advance, we can previously scan the text once and use counters, which temporarily requires at most $\sigma \lceil \lg(n/p) \rceil$ bits of space. Furthermore, we need to compute prefix sums of the bucket sizes on each level for their distributed concatenation. Since there are $O(\sigma)$ buckets per PE during the wavelet tree construction, this requires $O(\sigma \lg p)$ words to be sent and takes $O(\sigma \lg p)$ time. Furthermore, the prefix summing induces $O(\lg p)$ BSP barriers per level, i.e., $O(\lg p \lg \sigma)$ barriers for all levels. Additionally to prefix sum communication, during the concatenations, we send a total of $O(n \lg \sigma)$ characters, corresponding to the number of bits in the wavelet tree. We can use word packing to send $O(\lg n / \lg \sigma)$ characters per word, but may need an additional incomplete word per bucket and per PE. The number of total words sent is then $O((n \lg^2 \sigma)/ \lg n + p\sigma)$ in the worst case, dominating the number of words sent for prefix summing. We require at most $\lceil n/p \rceil \lceil \lg \sigma \rceil + 2\sigma \lceil \lg n \rceil$ bits of local memory to store the wavelet tree and precomputed node sizes and an additional send/receive buffer of $\lceil n/p \rceil \lceil \lg \sigma \rceil$ bits for the sort buckets. As opposed to domain decomposition, we directly construct the wavelet tree in its distributed levelwise representation and no subsequent merge operation is necessary. This leads to the following lemma.

Lemma 7.4. *Using distributed stable bucket sorting with p PEs, we can construct the wavelet tree for $T \in \Sigma^n$ with BSP costs of $O((n/p) \lg \sigma + \sigma \lg p + G(\sigma \lg p + (n/p) \lg \sigma) + L \lg \sigma \lg p)$, a communication volume of $O((n \lg^2 \sigma)/ \lg n + p\sigma)$ words and using at most $2(\lceil n/p \rceil \lceil \lg \sigma \rceil + \sigma \lceil \lg n \rceil) + \sigma \lceil \lg(n/p) \rceil$ bits of local memory.*

7.4 Splitting

Our third and final distributed memory algorithm follows an idea similar to the parallel algorithm par.rec presented in Section 3.2: given text T of length n for some wavelet tree node v, we first compute the node's bit vector BV_v in parallel using the p available PEs and count the number z of

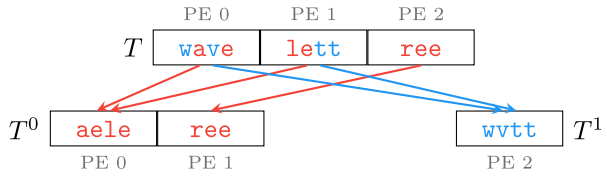


Fig. 13. Split of the first level for example text T = wavelettree using three PEs. Initially, each PE i has part T_i and decides which symbols go to T^0 and which go to T^1. The symbols are communicated accordingly.

0-bits in the process. Then, we split up T into T^0 and T^1 according to the bits in BV_v: If $\mathrm{BV}_v[k] = 0$ (for some $k < n$), then the character $T[k]$ is appended to T^0, and analogously, if $\mathrm{BV}_v[k] = 1$, then $T[k]$ is appended to T^1. We recurse on T^0 for the left child of v using pz/n PEs (but at least one) and on T^1 for the right child using $p(n-z)/n$ PEs (also at least one). By making the number of used PEs depend on the relative number of 0- and 1-bits, we achieve load balancing in that we assign more PEs to larger nodes. Figure 13 visualizes the splitting operation. In case only one PE remains to process a node, we use a sequential construction algorithm to construct the remaining wavelet subtree. Although the distribution of node bit vectors is different, the scenario after the construction using splitting is equivalent to that after the local construction phase in domain decomposition, since the bit vectors remain in PE rank order. Therefore, we can use the same merge operation to retrieve the levelwise representation.

The BSP cost analysis for this algorithm is rather complex, because whenever only one PE remains to compute a wavelet subtree of height greater than one, we switch to sequential construction. From that point on, the PE no longer communicates and the local computation time and memory requirements depend on the chosen sequential algorithm. Whether and how often this case occurs during construction depends on the distribution of the characters in the input and thus, so do the BSP costs. For this reason, we refrain from a detailed analysis and consider only a favorable case.

We call a text *uniform* if all the characters in it occur equally often. We now assume that the input T is uniform, that σ is a power of two and p a multiple of σ. Then, T^0 and T^1 are always of the same length $n/2$ in every recursion, and we recurse on both using $p/2$ PEs each.

The lengths of T^0 and T^1 equal the sizes of the left and right child of v, respectively. Therefore, the splits are very similar to filling sort buckets in Section 7.3: they take $O((n/p)\lg\sigma + \sigma\lg p)$ time and require $O(n\lg\sigma)$ characters to be sent. This requires up to $O((n\lg^2\sigma)/\lg n + p\sigma)$ words (using word packing) and dominates the communication volume. To store the partial wavelet tree and Borders arrays for all levels, we require at most $\lceil n/p\rceil\lceil\lg\sigma\rceil + 2\sigma\lceil\lg n\rceil$ bits of memory. The two prefix sums (for T^0 and T^1) and z can be stored in negligible $\lceil\lg n\rceil$ additional bits and finally, we need to store T^0 and T^1 as well as send/receive buffers in additional $2\lceil n/p\rceil\lceil\lg\sigma\rceil$ bits of local memory. As mentioned previously, we merge the partial wavelet trees using the same operation as for the domain decomposition.

Proposition 7.5. *For a uniform text T of length n over an alphabet of size $\sigma = 2^k$ for some integer $k > 0$ and p a multiple of σ, we can construct the wavelet tree using splitting with with BSP costs of $O((n/p)\lg\sigma + \sigma\lg p + G((n/p)\lg\sigma + \sigma\lg p) + L(\lg\sigma + \lg p + \lg\sigma\lg p))$, a communication volume of $O((n\lg^2\sigma)/\lg n + p\sigma)$ words and using at most $3\lceil n/p\rceil\lceil\lg\sigma\rceil + 2\sigma\lceil\lg n\rceil$ bits of local memory.*

7.5 Adaption to the Wavelet Matrix

In the following, we describe adaptations of our distributed memory wavelet tree construction algorithms to construct the wavelet matrix.

Domain Decomposition. As described in Section 2.3, the bit vector layout of the wavelet matrix only differs from that of the wavelet tree in that the nodes are re-ordered on each level according to the bit-reversal permutation. Therefore, we can easily adapt the domain decomposition for wavelet tree construction to construct instead the wavelet matrix. Locally, on each PE, we first construct the wavelet *tree* for the local part of the input. We now only modify the merge operation to take into account the bit-reversal permutation when concatenating node bit vectors on each level. Since we can compute the bit-reversal of a word in constant time, Theorem 7.3 also applies to the distributed construction of the wavelet matrix.

It remains to compute the z_ℓ values, i.e., the number of 0-bits on each level ℓ. We can use the Borders array to accumulate the sizes of left children in the wavelet tree on level $\ell + 1$, which equals the number of 0-bits on level ℓ. Since this requires $O(\sigma)$ time and a subsequent all-to-all sum, it does not worsen the asymptotic BSP costs.

Stable Sorting. The main idea behind the wavelet matrix is to be able to handle large input alphabets. To this end, we can simplify the bucket sorting approach to construct the wavelet matrix using only a constant number of buckets.

We can describe the reordering of nodes in the wavelet matrix compared to the wavelet tree in an intuitive way: On level $\ell + 1$, all left children of the wavelet tree nodes on level ℓ are moved to the left and all right children are moved to the right. Following this idea, to bring the symbols of the text in the correct order to compute the following level in the wavelet matrix, on level ℓ, we stably sort them according to their ℓth bit only as opposed to the $(\ell + 1)$-bit prefix for the wavelet tree. Then, we have only two distinct sort keys (0 and 1), and we can use the bucket sort approach with merely two sort buckets. These buckets can be filled on the fly as we scan the text to compute the bit vector of level ℓ, which contains precisely the ℓth bits of each symbol. The fact that we only have to deal with two buckets allows us to allocate a single buffer of length $\lceil n/p \rceil$ (where both buckets fit in precisely) and insert the symbols with sort key 0 from left to right and those with sort key 1 from right to left in reverse while keeping track of the number of entries in the 0-bucket, which marks the bucket boundaries. Subsequently, we reverse the contents of the 1-bucket in-place, requiring only a computation time linear in the size of that bucket. This way, we do not need a preliminary scan to find the bucket sizes. We can furthermore use the tracked number of 0-bits to compute z_ℓ in an all-to-all sum operation.

Using this approach, we lose the linear dependency of σ in our costs: We no longer need to precompute the Borders arrays for each level and store them, saving us $O(\sigma)$ time and up to $2\sigma\lceil \lg n \rceil$ bits of local memory, and we only require sizes and prefix sums for two instead of $O(\sigma)$ buckets on each level. However, we now need to account for $O(\lg p)$ time and $O(\lg p)$ words sent for the prefix sums *per level*, i.e., $O(\lg \sigma \lg p)$ time and $O(\lg \sigma \lg p)$ words in total. The all-to-all summing operation on each level for computing z_ℓ has the same asymptotic costs as the prefix sum computation.

Lemma 7.6. *Using distributed stable bucket sorting with p PEs, we can construct the wavelet matrix for $T \in \Sigma^n$ with BSP costs of $O((n/p + \lg p) \lg \sigma + G(\lg \sigma \lg p + (n/p) \lg \sigma) + L \lg \sigma \lg p)$, a communication volume of $O((n \lg^2 \sigma)/\lg n + p \lg \sigma)$ words and using at most $2\lceil n/p \rceil \lceil \lg \sigma \rceil$ bits of local memory.*

Splitting. Since the scenario after the construction using splitting is the same as that after local construction in the domain decomposition, we can again simply alter the communication pattern of the merge operation with respect to the bit-reversal permutation.

7.6 Experimental Evaluation

We implemented the presented distributed memory construction algorithms for the wavelet tree and wavelet matrix in C++ and provide them in a public repository at https://github.com/pdinklag/distwt.

To realize communication between PEs, we use MPI [37], more specifically the point-to-point send (non-blocking, e.g., `MPI_Isend`) and probe/receive (blocking, e.g., `MPI_Probe`/`MPI_Recv`) operations. Where applicable, we make use of MPI's implementations of collective operations like synchronization (`MPI_Barrier`), exclusive prefix summing (`MPI_ExScan`) and all-to-all reduction (`MPI_Allreduce`). Furthermore, we use MPI's concept of communicators to group nodes constructing the same wavelet subtree in the recursive split algorithm.

We compile using the GNU `g++` compiler version 7.3.0 and link against the Intel MPI Library version 2018.3.

Implementations. We provide the implementations *dist.dd.wt* and *dist.dd.wm* (domain decomposition for wavelet tree and wavelet matrix), *dist.split.wt* and *dist.split.wm* (recursive split for wavelet tree and wavelet matrix), and finally *dist.bsort.wt* and *dist.bsort.wm* (bucket sorting with $O(\sigma)$ buckets for the wavelet tree or constantly two buckets for the wavelet matrix). In the additional variant *dist.dynbsort.wt* of dist.bsort.wt, we grow buckets dynamically while constructing the level's bit vector instead of precomputing their sizes in a preliminary scan. Because of the capacity doubling strategy of `std::vector`, this causes more memory allocations and also allocates excess memory, but, as we will see, skipping that extra scan results in notably faster running times.

For local computations in the domain decomposition (for local wavelet tree construction) and recursive split (in case only one PE remains to construct a wavelet subtree), we pick seq.pc (Section 5.1) as the fastest practical sequential wavelet tree construction.

Performance Measurements. Throughout the experiments, we measure three major performance figures: *running time*, *network traffic*, and *memory usage*. Time and memory usage are local figures that we measure using wall clock times or by counting memory allocations, respectively. For measuring traffic, we build a façade for all MPI operations that we use and estimate their inter-node traffic, i.e., we do not count communication between PEs located on the same node. The estimation is straightforward for the primitive send and receive operations, where we can simply count the size of each passed message. For the collectives (prefix summing and all-to-all sums), since we do not know details about the MPI implementations, we assume a binary merge tree communication pattern. We believe that this estimation is somewhat pessimistic in the worst case and probably exceeds the actual traffic caused by these operations. However, since they only make up for a small percentage of the total traffic (compared to the larger payloads such as (bit-)strings), we consider this potential excess negligible.

Experiments. We conduct two main types of scaling experiments in the Small cluster, which offers 20 PEs per node: a *weak scaling* experiment, where we increase the size of the input as we increase the number of PEs, and a *breakdown* experiment where we increase the size of the input while keeping the number of PEs fixed. Furtermore, we identify the COST (Section 2.1.3) of our distributed algorithms against seq.pc and also extend the idea to find the number of nodes required to outperform the best shared memory algorithm, par.dd.pc, on a single node using all available cores.

7.6.1 Weak Scaling Results. We conduct weak scaling experiments where for N nodes, i.e., $p = 20N$, we process a prefix of size $n := N \cdot \text{GiB}$ (one gibibyte per node) of each input. It is crucial to note that, for the large alphabet inputs `RuWB` and `SA`, taking a longer prefix also results in the

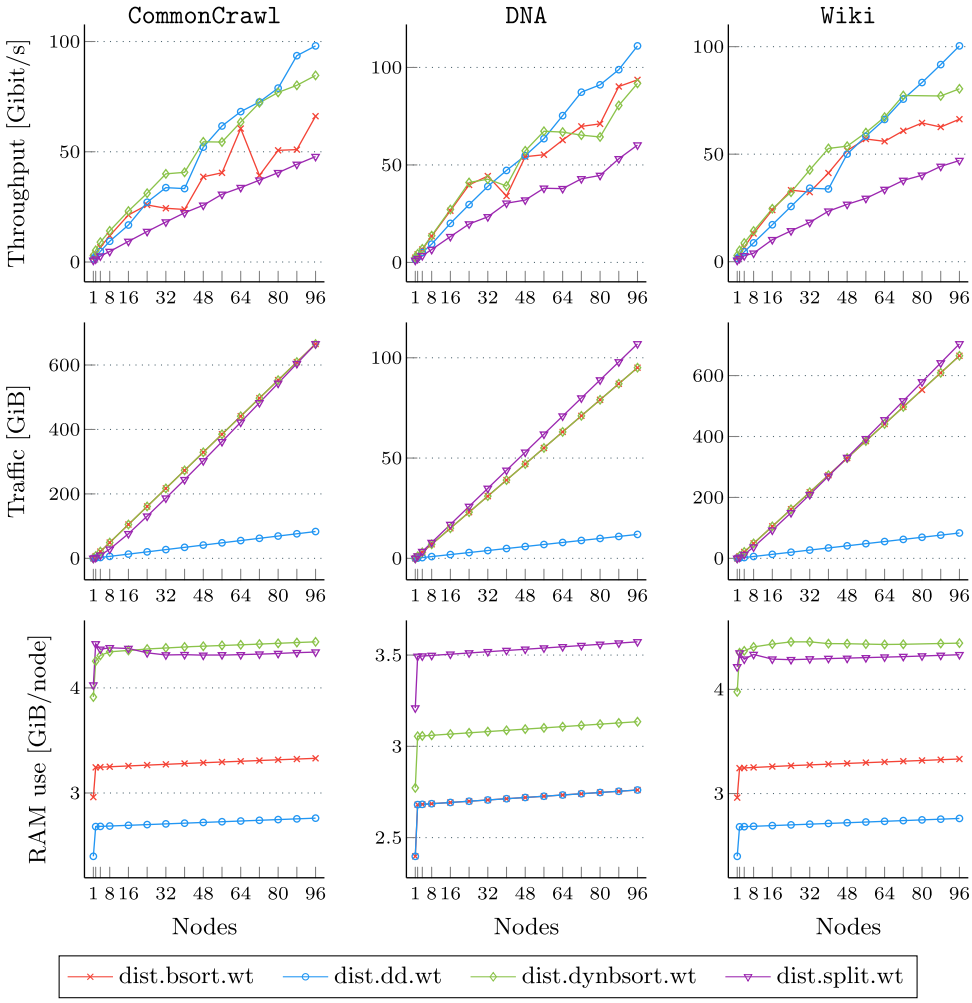


Fig. 14. Weak scaling distributed wavelet *tree* construction experiments.

input alphabet becoming larger. The results for wavelet trees are given in Figure 14 and those for wavelet matrices in Figure 15. In the following, we first consider the results for the small alphabet inputs (`CommonCrawl`, `DNA`, and `Wiki`) and then look at the large alphabet inputs (`RuWB` and `SA`).

Throughput. We see that all implementations scale well with an increasing number of nodes. To exemplify this, on `CommonCrawl`, dist.dd.wt achieves a median throughput of 1.47 Gibit/s using one node on a 1 GiB prefix and 98.01 Gibit/s using 96 nodes on a 96 GiB prefix (speedup of about 67).

The highest throughputs are achieved by dist.dd.wt (up to 98.01 Gibit/s on `CommonCrawl`), while dist.dynbsort.wt seems to be very close or even faster in some smaller instances (e.g., up to 48 nodes for `CommonCrawl`). Their overall similarity in throughputs is notable and reflected by the fact that dist.dynbsort.wt can be seen as just the merge operation of dist.dd.wt with some slight differences:

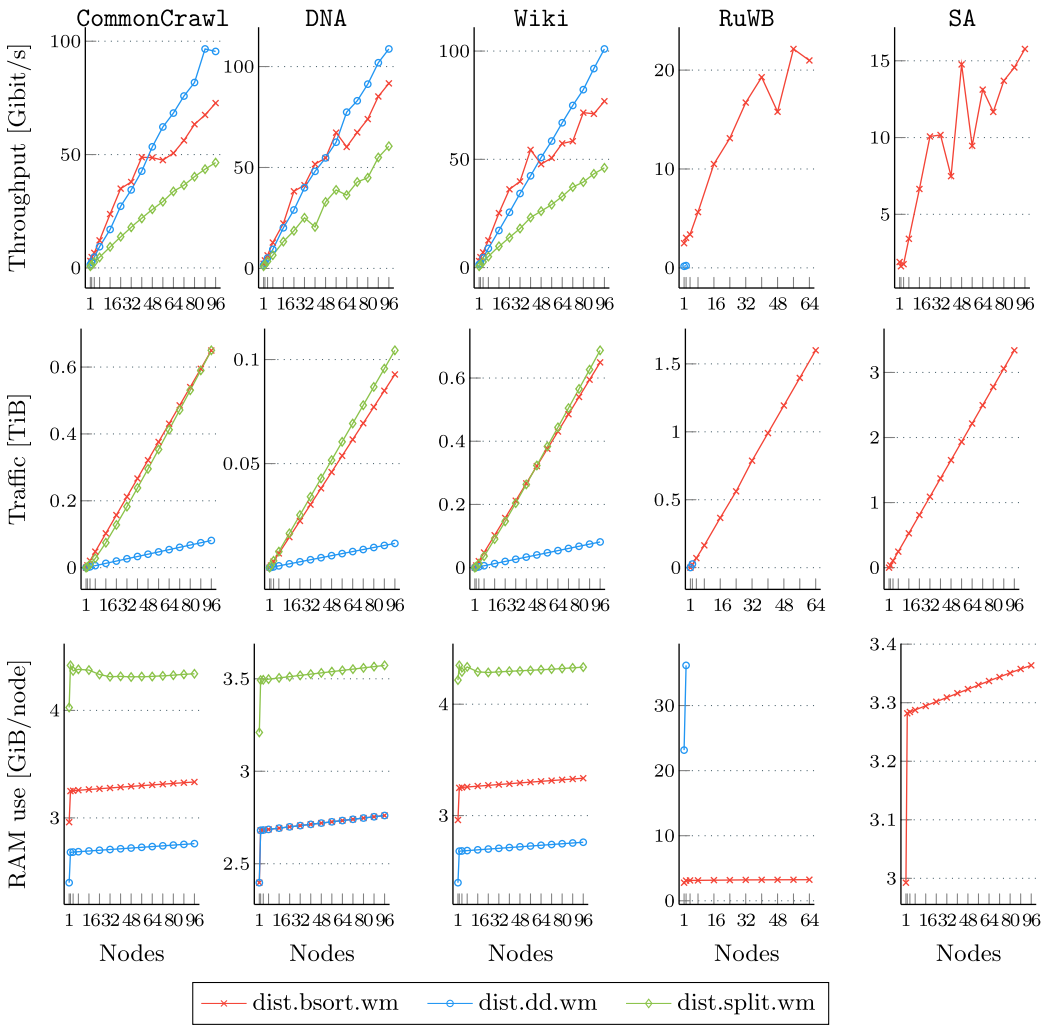


Fig. 15. Weak scaling distributed wavelet *matrix* construction experiments.

(1) dist.dynbsort.wt requires no preliminary local construction of the wavelet tree and
(2) communication is more local in dist.dynbsort.wt: when sorting by an $(\ell + 1)$-bit prefix, the order only changes within blocks of symbols whose ℓ-bit prefix was the same, i.e., buckets get refined in each sort operation and the distance between PEs that any symbol is sent over becomes smaller on each level. This results in increasing fast shared memory communication of PEs on the same node, which is an advantage over dist.dd.wt. However,
(3) in dist.dd.wt's merge, we communicate bit vectors and pack eight bits into one byte, while we need to communicate substrings to concatenate buckets in dist.dynbsort.wt. These occupy one byte per symbol (for the small alphabets) and thus increase the amount of handled data by a factor of eight (or a multiple of eight for larger alphabets).

The analysis is also valid for dist.bsort.wt. Here, however, the additional scan to determine the bucket sizes in advance causes the throughput to plummet (half as low as dist.dynbsort.wt on

CommonCrawl at 96 nodes). On DNA, however, we only do a single concatenation of two buckets after constructing the first level and the throughputs of dist.dynbsort.wt and dist.bsort.wt become nearly equal.

The fact that dist.split.wt fares worst (around 50% of the throughput of dist.dynbsort.wt on most CommonCrawl instances) can be explained as follows: the split texts T^0 and T^1, for a whole level, directly correspond to sorted buckets and after construction, we need to perform a merge operation. Thus, it combines the two expensive operations of dist.dynbsort.wt and dist.dd.wt.

We observe similar throughputs for our the wavelet matrix constructors. However, it is notable that even though dist.bsort.wm is closely related to the wavelet tree constructor dist.dynbsort.wt, it achieves only lower throughputs in comparison. The reason for this lies in the fact that we only concatenate two buckets for each level of the wavelet matrix. Other than in the wavelet tree scenario, these buckets are arbitrarily scattered over the PEs, so that no locality advantage comes into play at lower levels. This issue is again less severe for DNA, where we only construct two levels.

Traffic. Comparing the traffic of dist.dynbsort.wt and dist.dd.wt, the factor of eight in the amount of communicated data (symbols instead of bits) becomes very visible: on CommonCrawl at 96 nodes, we have a traffic of approximately 83 GiB for dist.dd.wt and 665 GiB for dist.dynbsort.wt (factor 8.01). We can also make out the equivalence of the split texts communicated in dist.split.wt to the buckets of dist.dynbsort.wt, causing very similar traffic footprints. Slight differences occur as dist.split.wt requires a merge operation after construction and dist.dynbsort.wt does not. However, it should be noted that dist.split.wt has even better data locality on deeper levels than dist.dynbsort.wt, because T^0 and T^1 are guaranteed to stay within the same node range. This ultimately results in more shared memory traffic, which we do not measure. The same observations apply to the wavelet matrix constructors, however, with the extra issues concerning dist.bsort.wm described above. The traffic footprints of dist.dynbsort.wt and dist.bsort.wt are exactly equal: they only differ in local memory consumption; the communicated data is the same.

RAM usage. Due to the low memory requirements of seq.pc, dist.dd.wt requires the lowest amount of RAM overall. For the merge operation after construction, we only need *one* buffer for sending and receiving wavelet tree levels (bit vectors) as we merge level by level. We see that the difference in memory usage of dist.bsort.wt and dist.split.wt is precisely the memory required by the subsequent merge operation in dist.split.wt, i.e., that of dist.dd.wt, matching our theoretical analysis. The difference between dist.dynbsort.wt and dist.bsort.wt is the excess memory allocated by dist.dynbsort.wt when doubling bucket capacities as they are filled (around 33% of excess memory on CommonCrawl instances).

Similar observations apply for the wavelet matrix constructors. Since dist.bsort.wm only needs two sort buckets, which we manage in the same buffer (filling one from the left and the other from the right), no dynamic re-allocation is needed and thus the memory footprint is better than that of dist.split.wm (about 25% less memory on CommonCrawl at two nodes). Still, the fact that dist.dd.wm only requires one buffer for merging bit vectors causes it to achieve the lowest RAM usage for small alphabets.

Large Alphabet Inputs. We note that we only have roughly 69 GiB of RuWB available and therefore did not do any experiments for this input beyond 64 nodes.

The algorithms with a linear dependency on the input alphabet size, i.e., all except dist.bsort.wm, fail to process most prefixes of RuWB and SA due to RAM limitations. The only algorithm able to process all instances is dist.bsort.wm with a near-constant memory footprint (2.49 GiB on average per node on SA at 96 nodes, i.e., 2.49× the local input size) thanks to the constant number of required buckets. No wavelet tree constructor processed any prefix of SA successfully and only dist.split.wt could process small instances of RuWB (omitted in Figure 14).

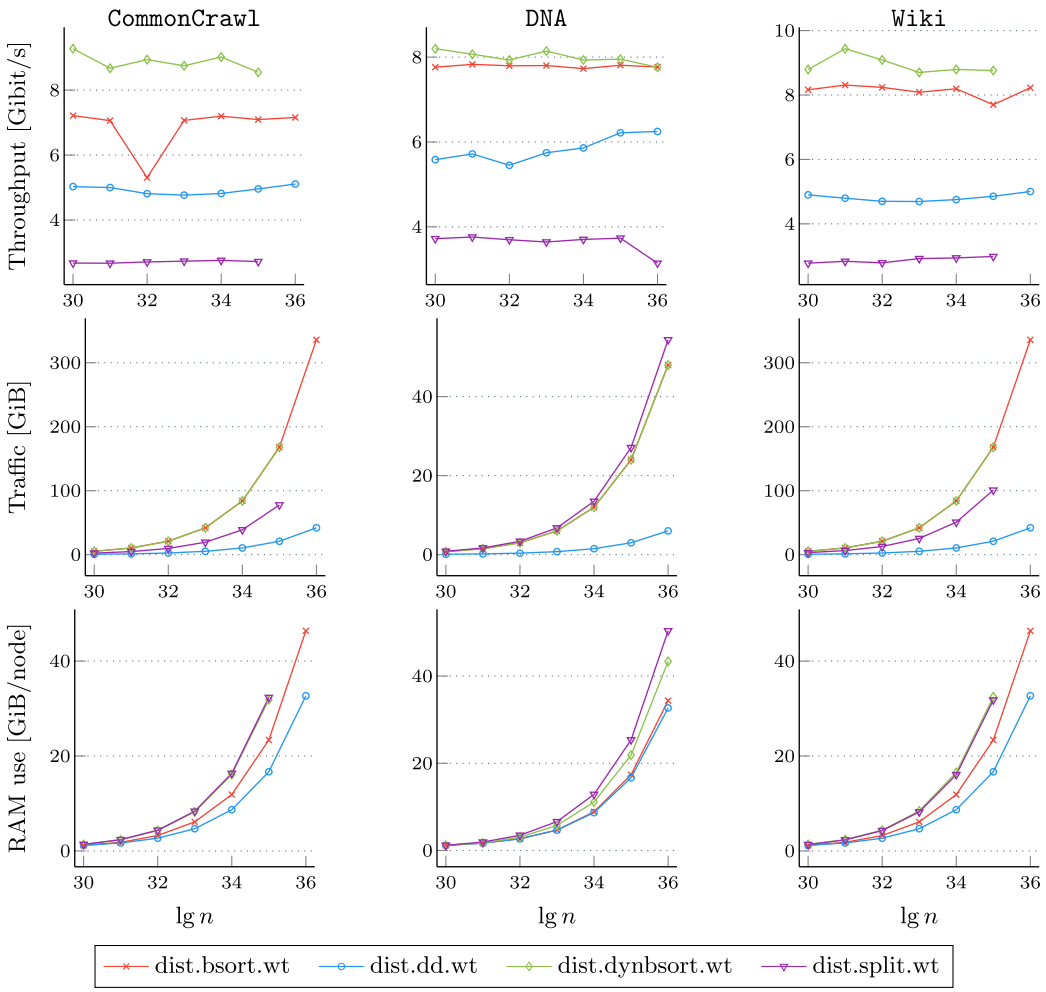


Fig. 16. Distributed wavelet *tree* construction breakdown experiments.

7.6.2 Breakdown Test Results. In our breakdown test, we fix the number of nodes to four ($p = 80$) and double the length n of the processed prefix in each iteration, starting with 1 GiB and stopping where none of our implementations succeeds any longer. We measure the same values as in the weak scaling experiments and give the results in Figures 16 and 17.

The measured traffic and RAM are as expected considering the weak scaling results. Correspondingly, dist.dd.wt and dist.bsort.wt for the wavelet tree and dist.dd.wm for the matrix have the lowest memory footprints and can therefore handle the largest inputs up to $2^{36} = 64$ GiB for the small alphabets, i.e., up to 16 GiB per node. Regarding the large alphabets, dist.bsort.wm is again the only implementation that can successfully handle similarly sized instances. The throughputs remain roughly constant throughout all breakdown experiments (decreasing by at most 8% for dist.dynbsort.wt on `CommonCrawl`), so the overhead caused by additional distribution is small.

7.6.3 Breakdown Test Results. In our breakdown test, we fix the number of nodes to four ($p = 80$) and double the length n of the processed prefix in each iteration, starting with 1 GiB and

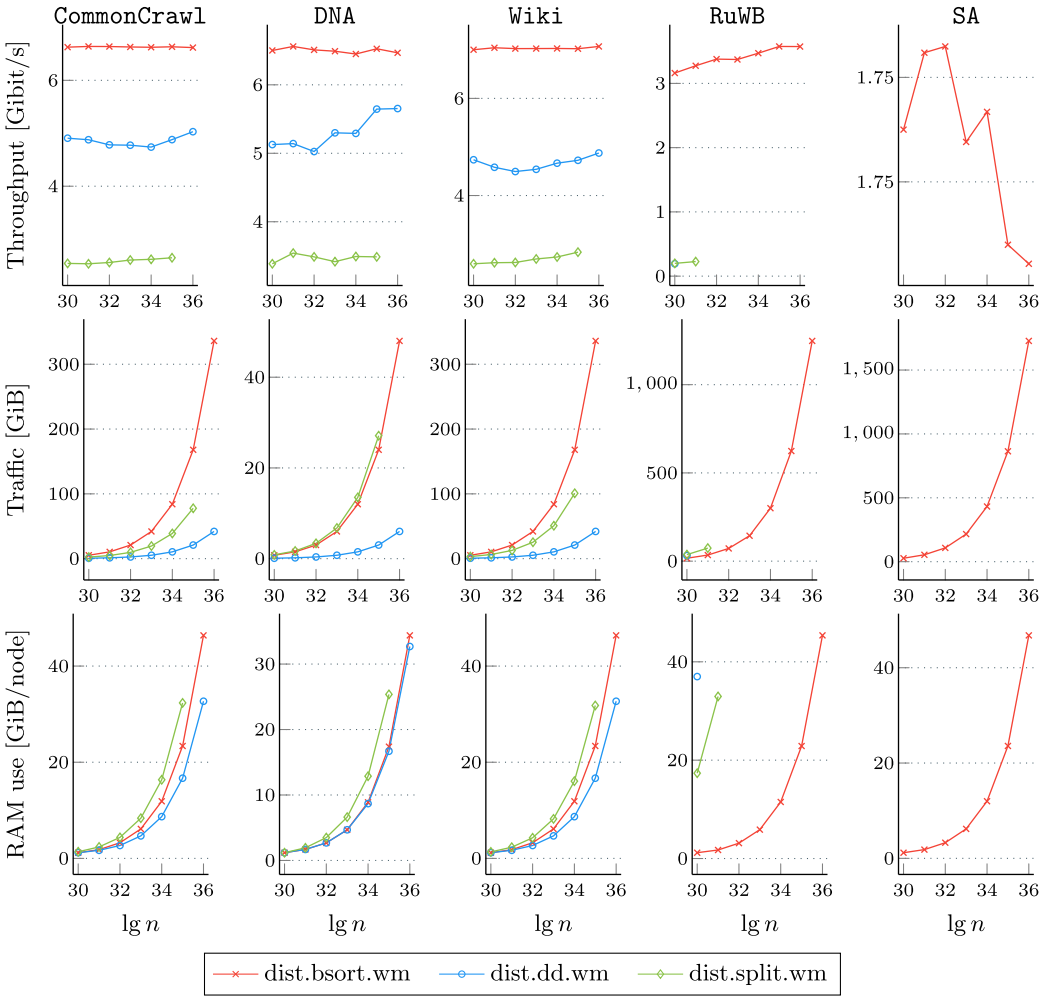


Fig. 17. Distributed wavelet *matrix* construction breakdown experiments.

stopping where none of our implementations succeeds any longer. We measure the same values as in the weak scaling experiments and give the results in Figures 16 and 17.

The measured traffic and RAM are as expected considering the weak scaling results. Correspondingly, dist.dd.wt and dist.bsort.wt for the wavelet tree and dist.dd.wm for the matrix have the lowest memory footprints and can therefore handle the largest inputs up to 2^{36} = 64 GiB for the small alphabets, i.e., up to 16 GiB per node. Regarding the large alphabets, dist.bsort.wm is again the only implementation that can successfully handle similarly sized instances. The throughputs remain roughly constant throughout all breakdown experiments (decreasing by at most 8% for dist.dynbsort.wt on `CommonCrawl`), so the overhead caused by additional distribution is small.

7.6.4 Outperforming Single Threads and Nodes. We conclude the experiments by finding the COST (see Section 2.1.3) of our distributed memory algorithms. Furthermore, extend this notion and also look for the configuration that outperforms the best known parallel shared memory algorithm using PEs on a single node. To this end, we compare to seq.pc (Section 5.1) as the single-threaded implementation and to par.dd.pc (Section 6.3) as the parallel shared memory

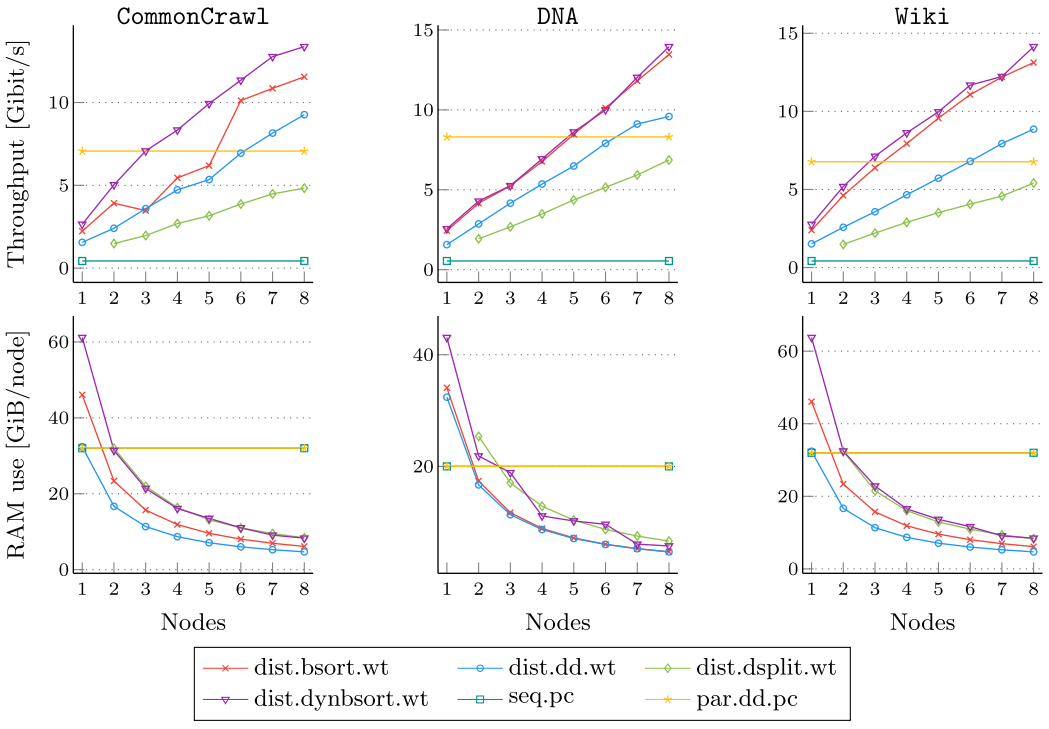


Fig. 18. Distributed wavelet *tree* construction COST experiments.

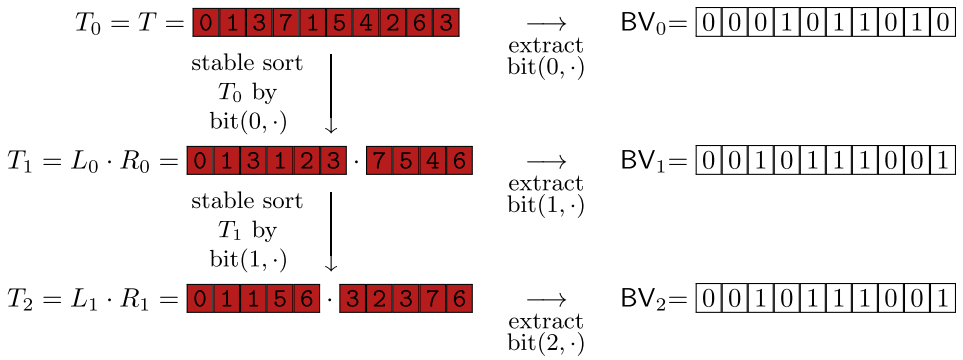


Fig. 19. Construction of the wavelet matrix for our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$ by partitioning the text, which is highlighted in dark red on the left-hand side, and the extraction of bits on the right-hand side.

implementation, the two that achieved the highest throughputs in their respective models. We fix the input prefix length to 16 GiB, which is the largest that seq.pc and par.dd.pc can successfully process in our setup. For our distributed implementations, we increase the number of nodes until we outperform both seq.pc and par.dd.pc; see Figure 18.

We note that dist.split.wt failed for all inputs on a single node due to RAM limitations. Apart from this, all of our other implementations outperform the single-threaded seq.pc on only one node. To that regard, the COST of our distributed algorithms is very low. Outperforming the

single-node parallel shared memory implementations requires between three (wavelet tree for Wiki) and five nodes (tree and matrix for DNA).

The per-node memory usages for this experiment are, in fact, the same for seq.pc and par.dd.pc for all inputs. As one would expect, dist.dd.wt requires the least memory and has the same footprint as seq.pc, as it uses the same algorithm for local construction. However, in the case of DNA, dist.dd.wt uses excess memory to store each input symbol in a byte each rather than compressing it to two bits. The bucket sorters, dist.bsort.wt and dist.dynbsort.wt, require additional memory for the sort buckets, which have a similar memory footprint to the split texts in dist.split.wt. Our implementations require less per-node memory than the sequential and shared memory implementations starting with two nodes (CommonCrawl and Wiki) to three nodes (DNA), which does not impact our low COST given above.

Due to high similarity of the results for the same texts, we omit the COST analysis for wavelet matrix constructors. Much like in the previous scaling experiments, dist.bsort.wm is the only implementation that can successfully construct the wavelet matrices for prefixes of RuWB and SA.

8 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 total 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 4), 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.

8.1 Semi-External Sequential Construction

In this section, we start with a brief discussion of how to adapt the sequential wavelet tree and wavelet matrix construction algorithms from Section 5 to the semi-external model, which we defined in Section 2.1.4. Remember that in the semi-external 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 5.2. 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 $O(n)$ 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 [43, 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 $\text{scan}(n\lceil \lg \sigma \rceil)$. We call this algorithm *se.ps*.

To overcome the space requirements for sorting, we now describe a new almost in-place algorithm that re-arranges the text as required by the wavelet tree in $O(n)$ time, using only $O(\sqrt{n} \cdot (\lg \sigma + \lg n))$ bits of additional space. Let T_ℓ be the input text sorted by the length-ℓ bit prefixes, then our goal is to obtain $T_{\ell+1}$ from T_ℓ. We achieve this by considering one node of level ℓ at a time, starting with the leftmost node and finishing with the rightmost one. Separating a node into its children means that we have to stably partition the corresponding text segment by the ℓth bit of each symbol (where the starting position and length of the text segment are known due to the precomputed histograms and borders). Let $T_\ell[i..j]$ be the text segment of any node, for which we assume that the numbers of zeros and ones in the text segment are multiples of $\lceil \sqrt{n} \rceil$. We briefly sketch how to overcome this rather strict assumption at the end of this paragraph. To partition

$T_\ell[i..j]$, we use two buffers that can hold $\lceil\sqrt{n}\rceil$ symbols each. First, we scan $T_\ell[i..j]$ from left to right and investigate the ℓth bit of each symbol. Whenever it is a 0-bit, we append the symbol to the first buffer, and otherwise to the second buffer. Whenever one of the buffers is full, we write its content to an already processed part of $T_\ell[i..j]$. Once we are done with the scan, $T_\ell[i..j]$ consists of length-$\lceil\sqrt{n}\rceil$ blocks such that every block contains either only zeros or only ones. Now, we only have to stably reorder the blocks such that the zeros are left and the ones are right. This can be done trivially by first precomputing the permutation that we have to apply to the blocks, and then swapping the blocks according to the permutation (resolving each cycle of the permutation separately). The additional space required for this step is $O(\sqrt{n}\lg n)$ bits for storing the permutation, and $O(\sqrt{n}\lg\sigma)$ bits as a buffer for swapping blocks. We assumed that the numbers of zeros and ones of the text segment are multiples of $\lceil\sqrt{n}\rceil$. Without this assumption, the two buffers that we use may be partially filled after we are done with the scan, i.e., their content has not been written back to the text yet. When reordering the text such that the 0-blocks are left and the 1-blocks are right, we have to insert the symbols of the 0-buffer (first buffer) between the rightmost full 0-block and the leftmost full 1-block. The symbols of the 1-buffer (second buffer) have to be inserted after the rightmost 1-block. This can easily be realized with an additional buffer of size $\lceil\sqrt{n}\rceil$. We omit further technical details. Clearly, processing a node as described above takes linear time in its length, and re-arranging the entire text takes $O(n)$ time.

We denote the new variant by *se.ps.ip*. This variant requires less space than se.ps, but due to the almost in-place re-arranging, it is one of the slowest algorithms (see our evaluation in Section 8.4 for details).

Lemma 8.1. *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 $O(n\lg\sigma)$ time using $O(\mathit{scan}(n\lceil\lg\sigma\rceil))$ I/Os, and $n\lceil\lg\sigma\rceil + \sigma\lceil\lg n\rceil + O(\sqrt{n}\cdot(\lg n + \lg\sigma))$ (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 5.1. 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 $O(n)$ time, $\mathrm{scan}(n\lceil\lg\sigma\rceil)$ I/Os, and $\sigma\lceil\lg n\rceil$ bits space in main memory, as described in Section 4.

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 $O(n\lg\sigma)$ 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 wavelet tree construction algorithm. Hence, we only read the text from the secondary memory. The number of I/Os is $2\,\mathrm{scan}(n\lceil\lg\sigma\rceil)$. 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 $\lceil\lg\sigma\rceil$ PEs. We denote this algorithm by *se.par.pc*.

Lemma 8.2. *The semi-external algorithm se.pc computes the wavelet tree of a text of length n over an alphabet of size σ in $O(n\lg\sigma)$ time using $O(\mathit{scan}(n\lceil\lg\sigma\rceil))$ I/Os, and $n\lceil\lg\sigma\rceil + \sigma\lceil\lg n\rceil$ bits of main memory including input and output, respectively.*

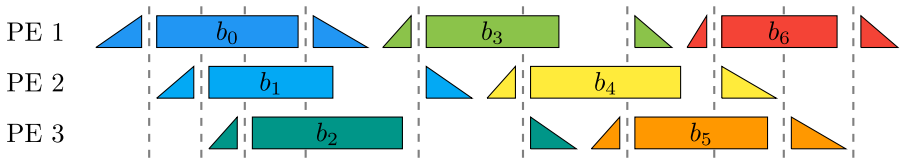


Fig. 20. Domain decomposition for a text $b_0|b_1|b_2|b_3|b_4|b_5|b_6$ split into seven segments. Here, we use ◺ to denote that segment b_i is loaded from external memory, ▭ represents the computation of the partial wavelet tree for segment b_i, and ◺ means writing the partial wavelet tree of segment 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.

Adaption to the Wavelet Matrix. Our semi-external 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.

8.2 Sequential Construction in External Memory

While the semi-external 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 2.1.4 for a definition of the model) algorithms that dispose of these limitations.

If we replace the sorting in se.ps with any external memory sorting algorithm, then 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.

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* [8]; see Figure 19. If we know T_ℓ, then we can easily build BV_ℓ by taking the ℓth 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 20. 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 $2 \cdot \text{scan}(n\lceil \lg \sigma \rceil)$ 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 $\text{scan}(n\lceil \lg \sigma \rceil) + 1$ 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 bound by $\mathcal{O}(n \lg^2 \sigma/(wB)) = \mathcal{O}(\lg \sigma) \cdot \text{scan}(n\lceil \lg \sigma \rceil)$. 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. In practice, if we keep these four strings as well as input and output on separate disks, then we have no random I/Os as all data on (separate) disks can be accessed concurrently in sequential order.

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 8.3. *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 4. 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 4. The wavelet tree construction needs additional $\mathcal{O}(\sigma)$ time to compute the histograms of all levels, resulting in the following lemma.

Lemma 8.4. *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.*

8.3 Parallel Construction in External Memory

For a more generic approach, we present a meta-algorithm based on the internal memory domain decomposition, as in Section 6.3. Let p be the number of available PEs, then in the internal memory setting, we split the text into $\Theta(p)$ segments and compute the wavelet tree of each segment on a different PE, using a sequential construction algorithm of our choice. After that, the so called *partial trees* can be merged into one *global tree*; see Figure 10 for an example.

In the external memory setting, we proceed similarly: Each PE still uses a sequential internal memory algorithm to compute the partial tree of a text segment. However, due to the limited

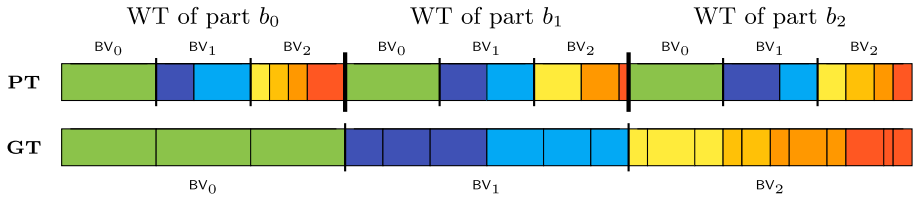


Fig. 21. External memory layout of partial (PT) and global (GT) WTs for $T = b_0 b_1 b_2$. Best viewed in color, as colors indicate parts of partial trees that are zipped together.

amount of main memory, we may have to decrease the segment length. Assume that the sequential construction algorithm needs $s(n, \sigma)$ bits of memory for a text of length n over the alphabet $[0, \sigma)$. If the segment length k satisfies $s(k, \sigma) \leq M/p$, then all PEs can work simultaneously without violating the main memory bound.

The smaller segment length implies that each PE may have to process multiple segments. We distribute the load by running a simple loop on each PE: 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 PE is allowed to read/write at a time (see Figure 20). 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 after the partial tree of the first segment, and so on. Here, each partial tree is the concatenation of its levels (see PT in Figure 21).

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 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 5. Since during the merge phase we are only copying bit vectors, in practice, we are limited by the speed of external memory. This holds even when using modern SSDs and only a single processor, which is why we are not using parallelism during the merge phase.

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 bounded 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 PEs are computing partial trees. The time during which all PEs are computing partial trees is at most $\lceil n/pk \rceil \cdot t(k, \sigma)$. As explained earlier, we are practically limited by the external memory speed during the merge phase and thus do not use parallelism. However, for the sake of theoretical analysis, we may apply the merging technique

from Section 6.3 to achieve a time bound of $O(n/p \lg \sigma + \lg(n\sigma/k))$ for the merge phase (apart from the I/O time).

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, $O(\lceil n/k \rceil \sigma \lg n)$ 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 increasing the number of I/Os.

Lemma 8.5. *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 available PEs, M the size of the main memory, and $k \in \mathbb{N}^+$ such that $s(k, \sigma) \leq M/p$. The external memory algorithm ext.dd computes the WT of a text of length n over an alphabet of size σ using $4\,scan(n\lceil \lg \sigma \rceil) + \sigma\lceil n/k \rceil$ I/Os. It takes $O(n \cdot (t(k, \sigma)/k + \lg \sigma)/p + \lg(n\sigma/k))$ time (apart from I/O time) and $O(\lceil n/k \rceil \sigma \lg n) + p \cdot s(k, \sigma)$ bits of internal memory including input and output.*

Adaptation to the Wavelet Matrix. Adapting 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, then the time, memory, and I/O bounds described in Theorem 8.5 hold for the corresponding wavelet matrix construction algorithm given that we use an algorithm with the same time and memory bounds.

8.4 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.github.com/kurpicz/pwm. We conduct the experiments on our external memory system that we described in Section 2.4.2. 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 [11] development snapshot (26-09-2017).

8.4.1 Evaluation of Semi-External Algorithms. We compare the following semi-external wavelet tree construction algorithms: (1) se.pc, (2) se.par.pc, (3) se.ps, and (4) se.ps.ip and their wavelet-matrix-constructing counterparts described in Section 8.1, (5) seq.sdsl, the semi-external algorithm contained in the SDSL, (6) seq.pc, the fastest sequential *main memory* wavelet tree construction algorithm (see Section 5.4), and (7) par.dd.pc, the fastest *shared memory* wavelet tree construction algorithm.

The last two algorithms are used to get a baseline and are expected to be faster. We show the results of our experiments in Figure 22. However, we focus on the wavelet tree construction, as the conclusions also hold for the wavelet matrix construction. Also, since 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. Since the operating system of our test machine uses around 512 MiB of the 16 GiB of memory, it will *swap* some data from memory to disk whenever an algorithm requires more than 15.5 GiB of memory.

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 the low throughput of our parallel semi-external algorithm se.par.pc on inputs with small alphabet. This shows on `DNA` and to some extent on `Prot`. We described the reasons for this in Section 6.1.

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

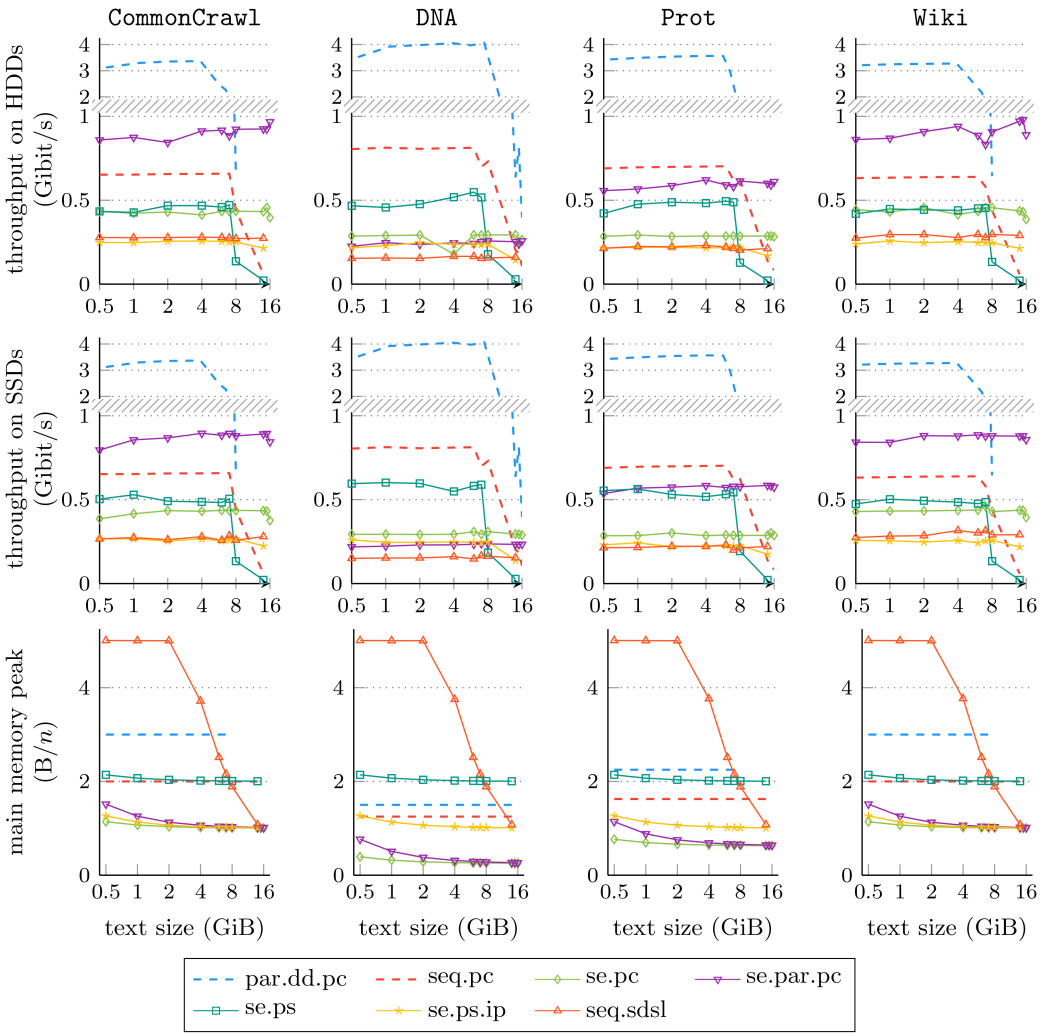


Fig. 22. Throughput and main memory peaks of the semi-external wavelet *tree* construction algorithms when using the HDDs (first row) and when using the SSDs (second row). The parallel algorithms are using all six PEs. In the last row, we give the main memory peak, which is independent of the used drive. 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.

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 algorithm provided by the SDSL is the slowest one we tested on `DNA` and of similar speed as se.ps.ip 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.

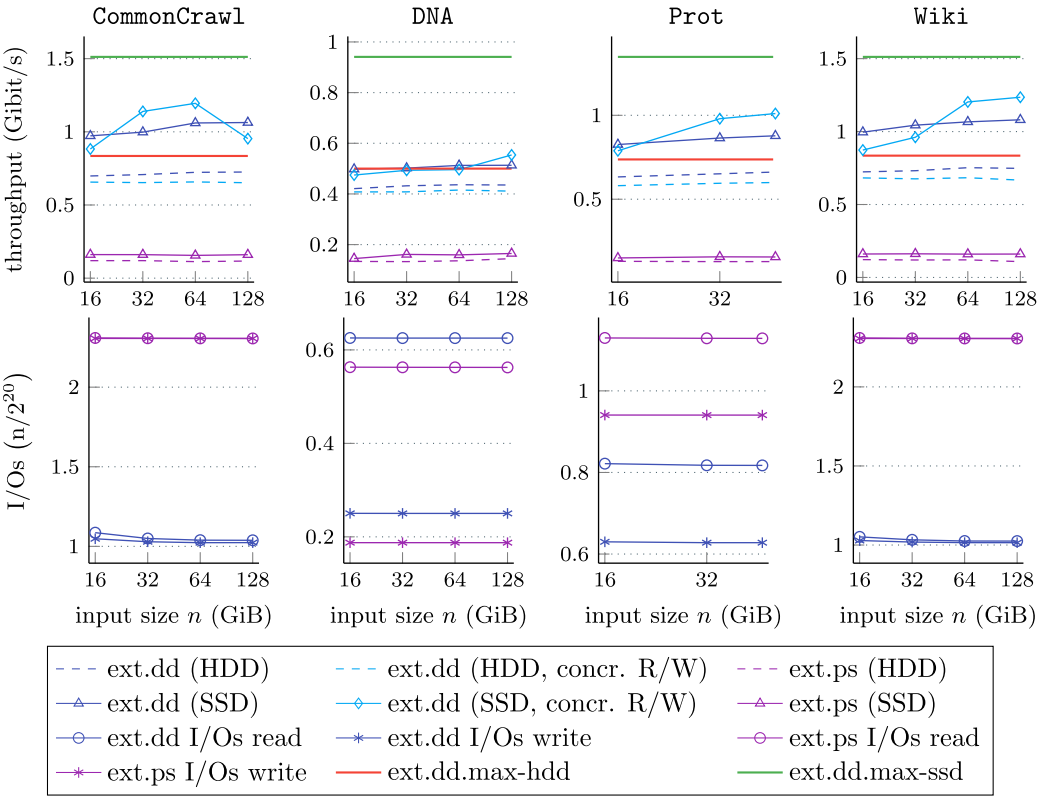


Fig. 23. Throughput and I/Os of the external memory wavelet *tree* construction algorithms in our strong scaling experiment. Here, our parallel external memory algorithm ext.dd uses six PEs.

In general, se.pc has the smallest memory peak. Its parallel variant se.par.pc requires only slightly more memory, as each PE 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. Unfortunately, our fastest semi-external wavelet tree algorithm se.ps requires the most memory of all our new algorithms; even more than seq.pc on all inputs. Only seq.sdsl requires more memory (while also being significantly slower).

8.4.2 Evaluation of External Memory Algorithms. Our external (and parallel external) 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 PEs the same, i.e., one PE for our sequential algorithms ext.ps and ext.ps.wm and six PEs 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 PEs together with the input sizes.

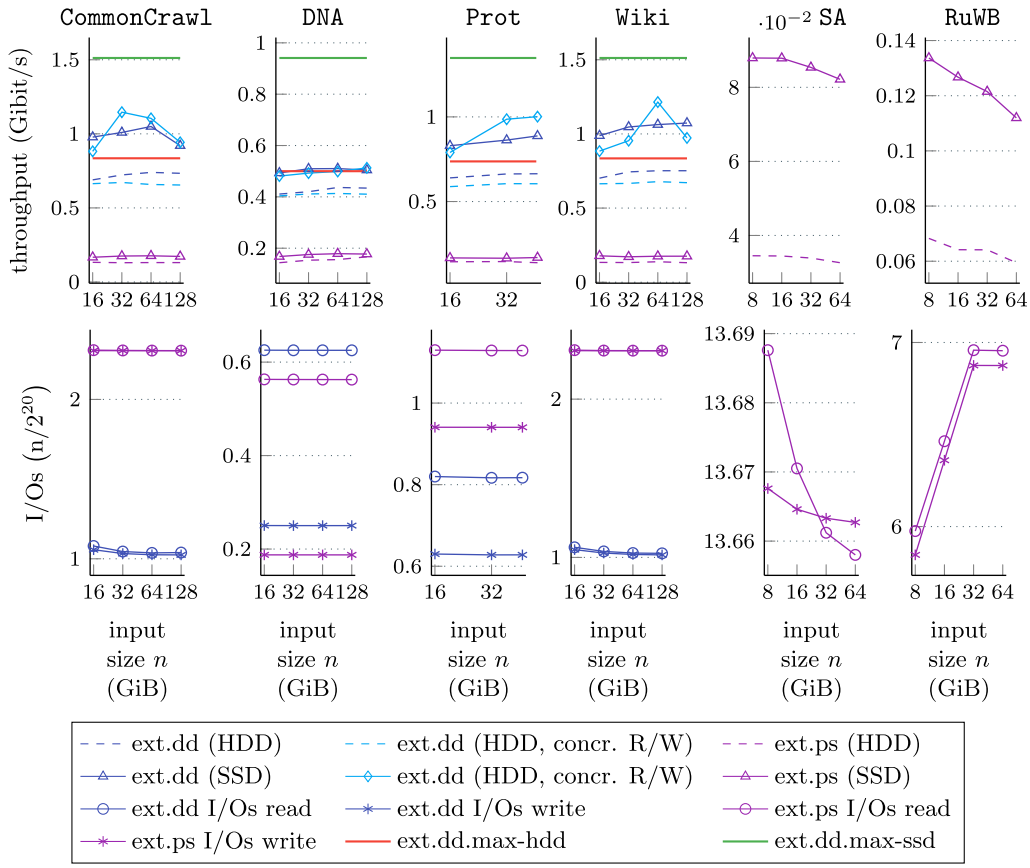


Fig. 24. Throughput and I/Os of the external memory wavelet *matrix* construction algorithms in our strong scaling experiment. Here, our parallel external memory algorithm ext.dd uses six PEs.

We first look at the maximum throughput that we can achieve with our parallel algorithms using Ext.ssd and Ext.hdd and denote those by *ext.dd.max-ssd* and *ext.dd.max-hdd*. 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 ext.dd.max-ssd and ext.dd.max-hdd are strict upper bounds for the throughput of our external memory domain decomposition algorithms. Note that as in the previous sections, we measure the troughput in computed output bits per second. Consequently, we get different bounds for different input texts, as the size of the resulting wavelet tree or matrix decreases for smaller alphabets. For example, for every input byte of `CommonCrawl`, we write one byte to the resulting wavelet tree, while for every input byte of `DNA`, we only write two *bits* to the resulting wavelet tree. Thus, the maximum possible throughput is significantly lower for `DNA`.

In addition, we have two different versions of ext.dd and ext.dd.wm that we now briefly describe.

(1) In the *traditional* variant only one PE is allowed to read or write at the same time. Thus, PEs may have to wait for other PEs 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 allows for concurrent read and write access by different PEs. Hence, no PE has to wait for its I/Os. In this case the operating system has to schedule the read and write access to external memory. We denote this version by *(concr. R/W)*.

In the following, similarly to previous evaluations, we only discuss 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. Note that ext.ps is the only external memory algorithm that is suitable for RuWB and SA. All other algorithm need to keep the histograms in main memory, which is not feasible for large alphabets. For the same reason, ext.ps can only build the wavelet *matrix* for these texts, but not the wavelet *tree*. In Figures 23 and 24, 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 PEs.

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 of the external memory device. The exception is the wavelet *matrix* construction for RuWB and SA, where Ext.ssd is around 2 times faster than Ext.hdd. The reason for this is that due to the large alphabets, we have to perform significantly more I/O operations, which also explains why the throughput for these texts is significantly lower. For our parallel algorithms, we benefit from faster external memory on all texts, 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 than ext.ps on all inputs but DNA. 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.ps requires already 1.79 times as many reads and 1.49 times as many writes as ext.dd. On CommonCrawl and Wiki it requires 2.25 times as many writes and 2.27 times as many read operations. For larger alphabets, the number of I/O operations required by ext.ps increases even further. On SA, the I/O operations remain almost constant independent from the input size, while larger inputs require more I/O operations on RuWB. This is due to the fact that the effective alphabet depends on the input size for RuWB (for $n = 8, 16, 32, 64$ GiB, we have $\lg \sigma = 25, 26, 27, 27$). In contrast, we do *not* reduce SA to its effective alphabet and always assume $\sigma = 2^{40}$, because the reduction cannot be easily performed in external memory.

Overall, ext.dd is the better external memory wavelet tree construction algorithm (compared to ext.ps). Additionally, in our weak scaling experiments (see Figure 25), we report that ext.dd has a higher throughput than ext.ps even on one PE. Thus, ext.ps should only be used for large alphabets.

Weak Scaling. We give the results of our weak scaling experiments in Figure 25, which are used to show that increasing the number of PEs 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 PEs. Again, we also show

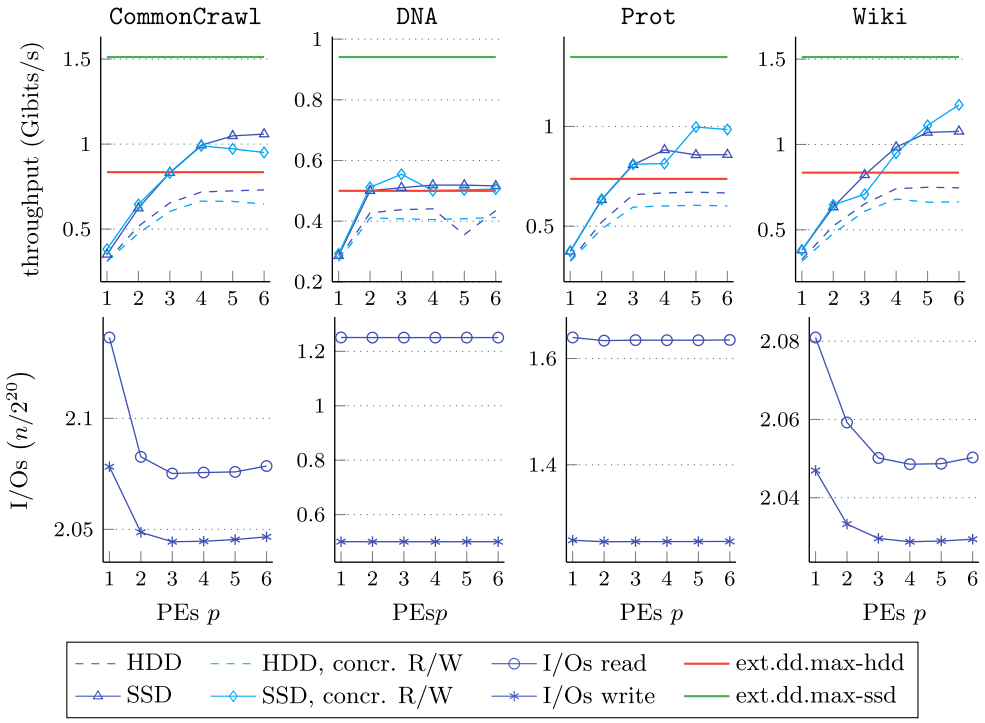


Fig. 25. Throughput of ext.dd, our parallel wavelet *tree* construction algorithm, for inputs of size 20 GiB per PE in our weak scaling experiment.

the maximum throughput that ext.dd and ext.dd.wm could achieve using both 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 PEs. However, on Ext.hdd, throughput does so only up to two (`DNA`), three (`Prot`), or four (`CommonCrawl` and `Wiki`) PEs. Using Ext.ssd, we achieve the linear increase up to two (`DNA`), four (`CommonCrawl`), five (`Prot`), or six (`Wiki`) PEs. Here, it should be noted that the best speedup on Ext.ssd is achieved when we allow concurrent 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 that we achieve with six PEs is only 3.23 (on `Wiki` in the Ext.ssd setting, allowing concurrent read and write operations). Still, these results are very good, because for sufficiently large alphabets, we achieve up to 81% of the maximal possible throughput (compared to ext.dd.max-ssd) on SSDs and up to 89% for HDDs (compared to ext.dd.max-hdd).

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 PEs. On `Prot`, we see a slight decrease when we use more than one PE (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.

9 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 [21] and wavelet matrices [8], i.e., we construct the wavelet tree or matrix not for the original text but its Huffman encoded counterpart [24]. A potential disadvantage is that there may be more levels than in an ordinary wavelet tree as the some Huffman codes may require more bits than the character it encodes. It is possible to balance the tree to obtain $O(\lg \sigma)$ levels [41], which we do not consider in this section. Still, the total length of all bit vectors of a Huffman-shaped wavelet tree or matrix is at most as long as the total length of the corresponding normal wavelet tree or wavelet matrix. 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 [21] but also more recent FM-indices [20] use Huffman-shaped wavelet trees. In general, Huffman-shaped wavelet trees reduce the required space and average query time, but increase construction time. Note that there also exists other compressed variants of wavelet trees, e.g., the *compressed* wavelet tree [29], which is used as auxiliary data structure to efficiently compute the Burrows-Wheeler transform [5] and captures other measures of repetitiveness.

In Section 9.1, we describe how to compute the Huffman codes such that they can be used in wavelet trees and matrices. Then, in Section 9.2, we explain how our algorithms described before—the sequential, shared memory parallel ones for wavelet trees and 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 9.3, 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.

9.1 Huffman Codes for Wavelet Trees and Wavelet Matrices

First, we briefly describe the construction of Huffman codes [24] with a focus on codes that can be used for wavelet trees and wavelet matrices.

9.1.1 Computing Huffman Codes. Given a text T over an effective alphabet Σ and its histogram Hist. In the well-known algorithm for Huffman codes, we start with a tree $t_{\{\alpha\}}$ for each $\alpha \in \Sigma$ with weight $w(t_\alpha) := \text{Hist}[\alpha]$. 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 the roots of t_A and t_B. We repeat this process until there is only one tree left. The leaves of this tree correspond to the 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 26.

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 all codes can have different lengths, this can lead to *disappearing* intervals, which 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 $c = (01)_2$, 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 answering 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

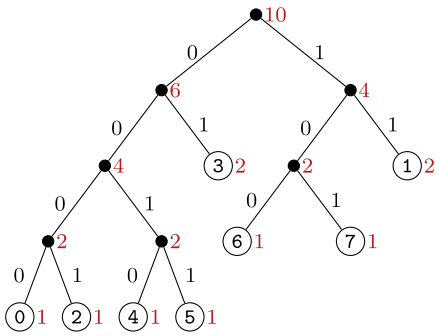


Fig. 26. A Huffman tree for our running example $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$. Each character is represented by a leaf. The dark red node labels are the total number of occurrences of all characters represented in the subtree. The Huffman code of a character is the concatenation of all edge labels on the path from the root to the character's leaf.

intervals should occur on the right-hand side of the bit vector they disappear in, as then no other intervals are affected by their disappearance. This also simplifies queries. If we require a bit whose 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 differently for the wavelet tree and matrix, as the order of the intervals at each level differs for both.

9.1.2 Huffman Codes for Wavelet Trees. When computing Huffman codes 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 ascendingly. 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 $T = [\,0, 1, 3, 7, 1, 5, 4, 2, 6, 3\,]$ are depicted in Figure 27.

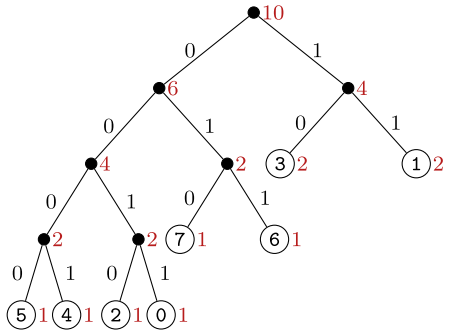
To obtain the canonical Huffman, we first compute the (normal) Huffman codes for our text. Then, we order these codes by length. Finally, we start with the code word $chc = (0^\ell)_2$, where ℓ is the length of our shortest code word. Now, chc is the first canonical code word. We increase chc by one 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).

9.1.3 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. [8] that are also optimal prefix free codes. We give an example of Huffman-like codes for our running example in Figure 28.

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

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 26 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).

Fig. 27. Huffman codes corresponding to the Huffman tree in Figure 26 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 orange parts show the characters represented at the corresponding position in the array and is not part of the wavelet tree.

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. In our example in Figure 28, during the first three steps, C contains the following elements: (1) $C = \{0_2, 1_2\}$, (2) $C = \{00_2, 01_2, 10_2, 11_2\}$, and (3) $C = \{000_2, 001_2, 100_2, 101_2\}$.

9.2 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 4, because we cannot compute

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).

Fig. 28. Huffman-like codes [8] 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 orange parts show the characters represented at the corresponding position in the array and is not part of the wavelet matrix.

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. Now, we 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 5.1) and *sorting* (Section 5.1) 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 9.3.

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 6, 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 6.3 for a detailed description of wavelet tree construction using domain decomposition.

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. The only difference in merging 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 is sequential. This is because computing the Huffman codes is fast compared to the computation of the histograms for our tested alphabet sizes.

9.3 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.github.com/kurpicz/pwm.

For our experiments, we used the hardware described in Section 2.4.2 and conducted the experiments on a single Big node.

9.3.1 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 [19]. To the author's best knowledge, there are no other implementations of Huffman-shaped wavelet tree and wavelet matrix construction algorithms publicly available.

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.

Construction Time. We first look at the construction times. In Figure 29, 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 39.) In general, we obtain half the throughput we get when we construct the *normal*-shaped wavelet tree, compare Section 5.4. 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 9.2.

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 2 h (which is the time limit for all our experiments in this part of the article). 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

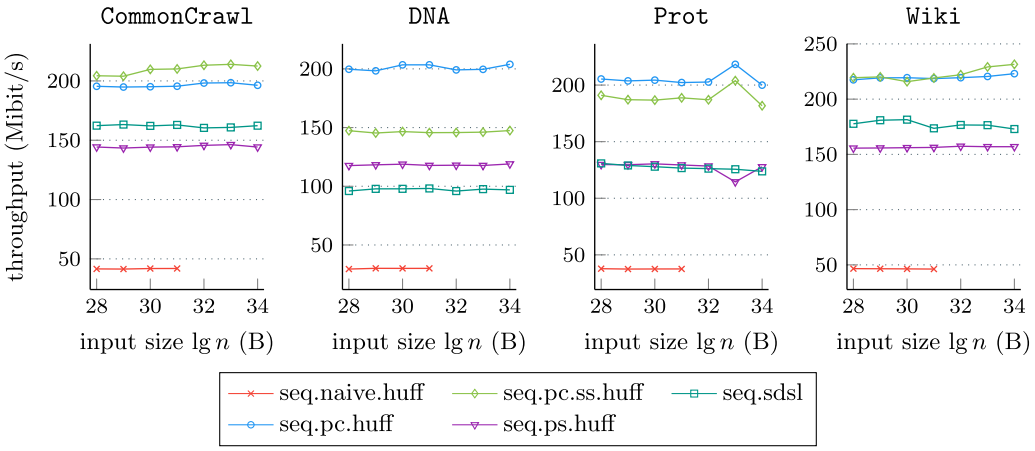


Fig. 29. Throughput of the sequential Huffman-shaped wavelet *tree* construction algorithms.

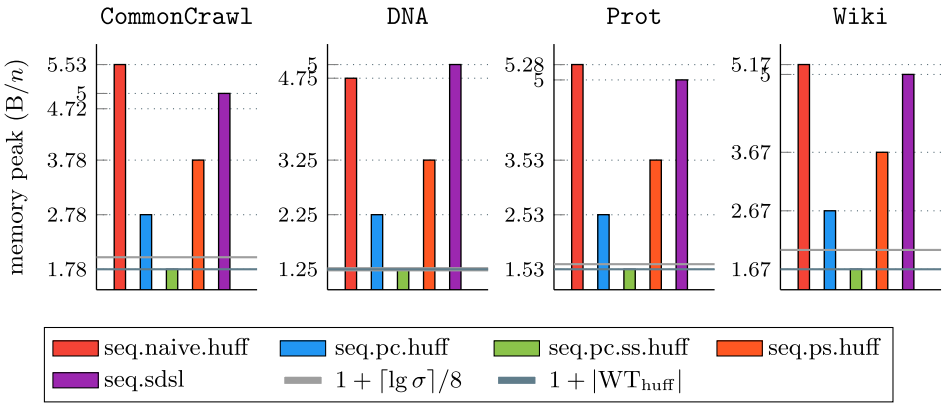


Fig. 30. Memory peaks of wavelet *tree* construction algorithms for $n = 2^{31}$. Also depicted is the memory required by the text and by the *normal*-shaped wavelet tree ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character), the text and the Huffman-shaped wavelet tree ($1 + |\mathrm{WT}_{\mathrm{huff}}|$).

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

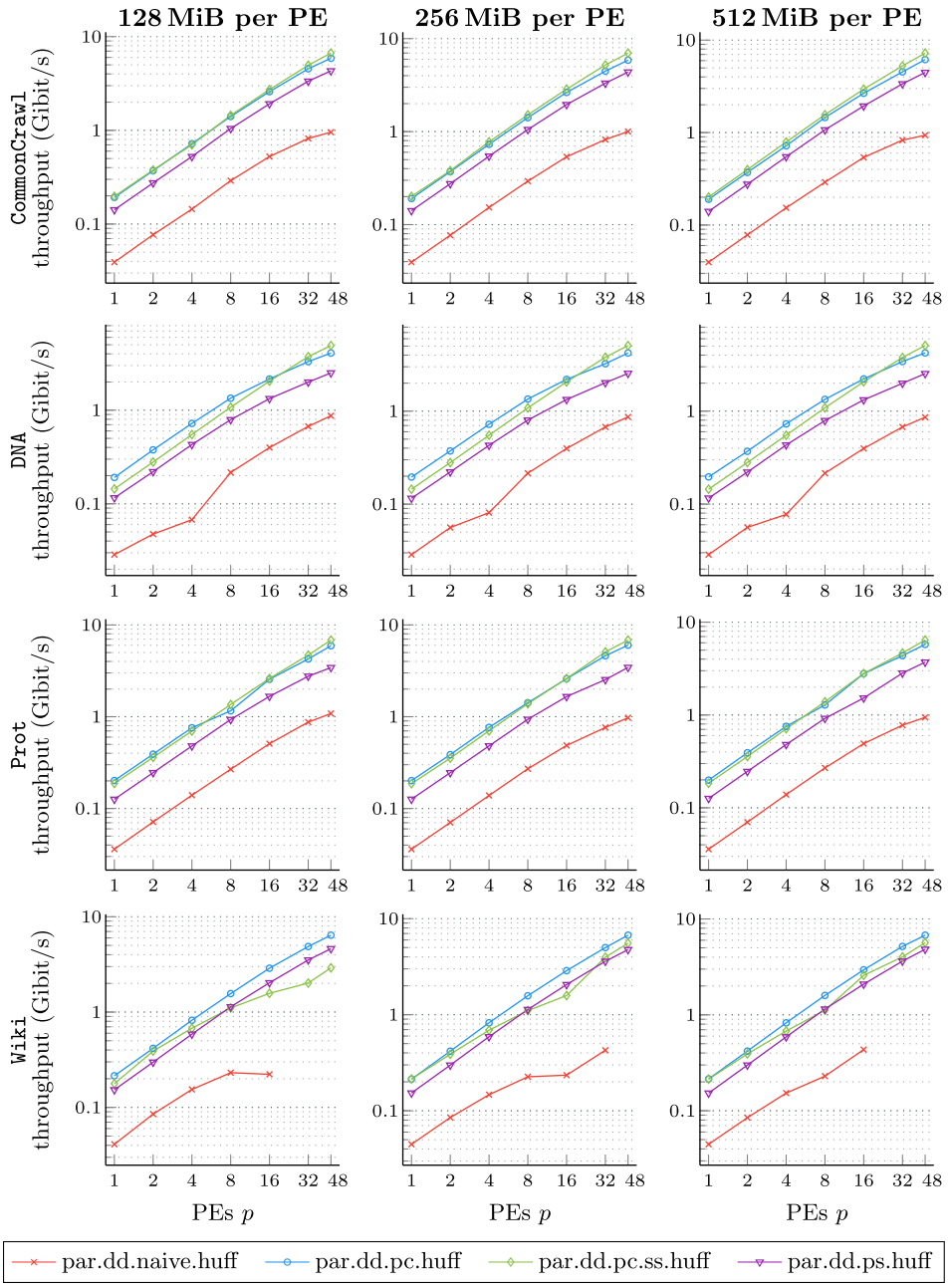


Fig. 31. Weak scaling Huffman-shaped wavelet *tree* construction experiments.

the input is of greater importance here, as different Huffman codes result in different sizes of the levels; see Section 2.4.1 for the empirical entropy H_0 of the used inputs.

Memory Peak. In Figure 30, 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

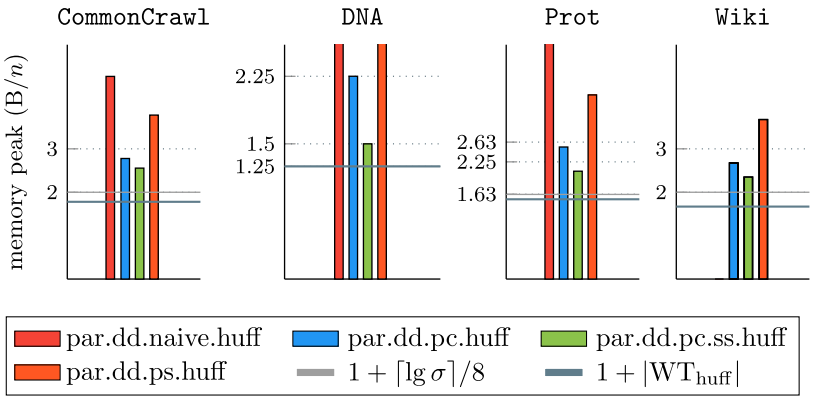


Fig. 32. Memory peaks of the parallel wavelet *tree* construction algorithms using 48 PEs and 256 MiB input per PE. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

Huffman-shaped wavelet matrix construction are depicted in Figure 40.) 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 8. 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 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.

9.3.2 Evaluation of Parallel Algorithms. Since there are no other parallel Huffman-shaped wavelet tree or wavelet matrix construction algorithms, we only present the results of our algorithms. We conducted a weak scaling experiment where we constructed the wavelet tree for 128 MiB, 256 MiB, and 512 MiB per PE and 1, 2, 4, 8, 16, 32, or 48 PEs. 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 31. (In Figure 41, 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.

Table 6. Throughput (Gibits/s) of the Huffman-shaped Wavelet Tree and Wavelet Matrix Construction Algorithms in Our Weak Scaling Experiment when Using 1 (t_1) or 48 (t_{48}) PEs and 512 MiB input per PE

	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

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.

Finally, par.dd.pc.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 PE and is 1.07 Gibits/s faster on 48 PEs.

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 PEs. Using 16 PEs, the algorithms are of similar speed. When we use more than 16 PEs, par.dd.pc.ss.huff is faster than par.dd.pc.huff. This is also the same on Prot, however, the difference in throughput is smaller. Finally, on Wiki, par.dd.pc.huff is the fastest algorithm.

We give the throughput of all algorithms on one and on 48 PEs in Table 6. 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 PEs 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 2.1.3) 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 PE but faster if two or more PEs are used. Therefore, our parallel algorithms do not only scale well, because they are slow when executed as sequential algorithm.

Memory Peak. We give the memory peak of our parallel Huffman-shaped wavelet tree construction algorithms in Figure 32. (The results for the corresponding wavelet matrix construction are shown in Figure 42.) 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.

10 CONCLUSIONS

We have presented a new technique for wavelet tree and wavelet matrix construction—the *bottom-up* construction—that allows for very fast construction times in practice. Using this technique, we

provide fast, scaling, and memory efficient sequential, parallel (shared memory and distributed memory), and external construction algorithms, which we also parallelized.

Since the wavelet tree is an important part of more complex text indices, most notably the FM-index [15], making it an important part of most DNA read aligners [32], this work building blocks for the parallel and memory efficient construction of such more complex and advanced tools that are used in practice. This also holds true for the many more applications of wavelet trees that we briefly touched in this article.

Future Work. As we focused on the construction of the bit vectors of the wavelet trees it remains an open question how to efficiently parallelize the construction of the *binary* rank and select data structures, which are necessary to navigate in the wavelet tree, in practice. Efficiently answering queries, especially in the distributed setting may also pose additional difficulties due to load balancing problems that might occur. Another interesting direction of research is the combination of the tested models of computation, e.g., providing algorithms that make used of shared, distributed, and external memory at the same time.

A ADDITIONAL PLOTS FOR WAVELET MATRIX CONSTRUCTION

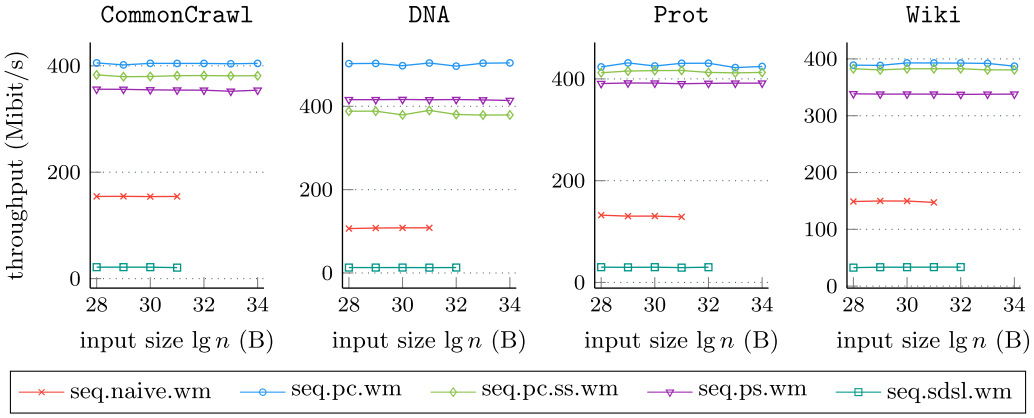


Fig. 33. Throughput of the sequential wavelet *matrix* construction algorithms.

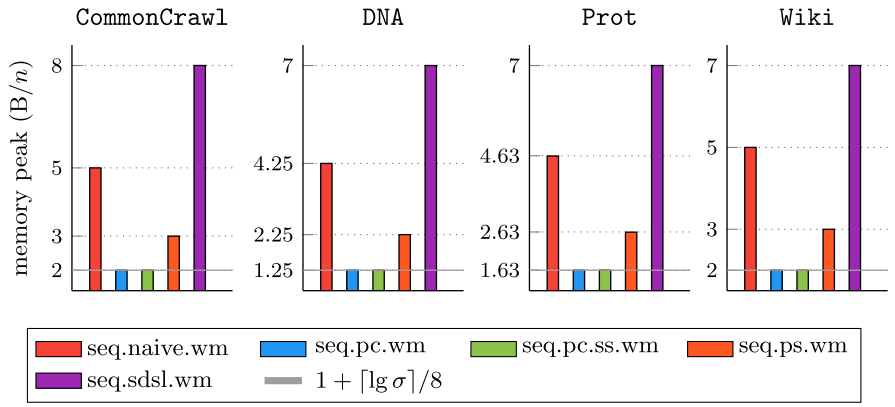


Fig. 34. Memory peaks of sequential wavelet *matrix* construction algorithms for $n = 2^{31}$. We also depict is the memory required to store the text and the wavelet matrix ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character).

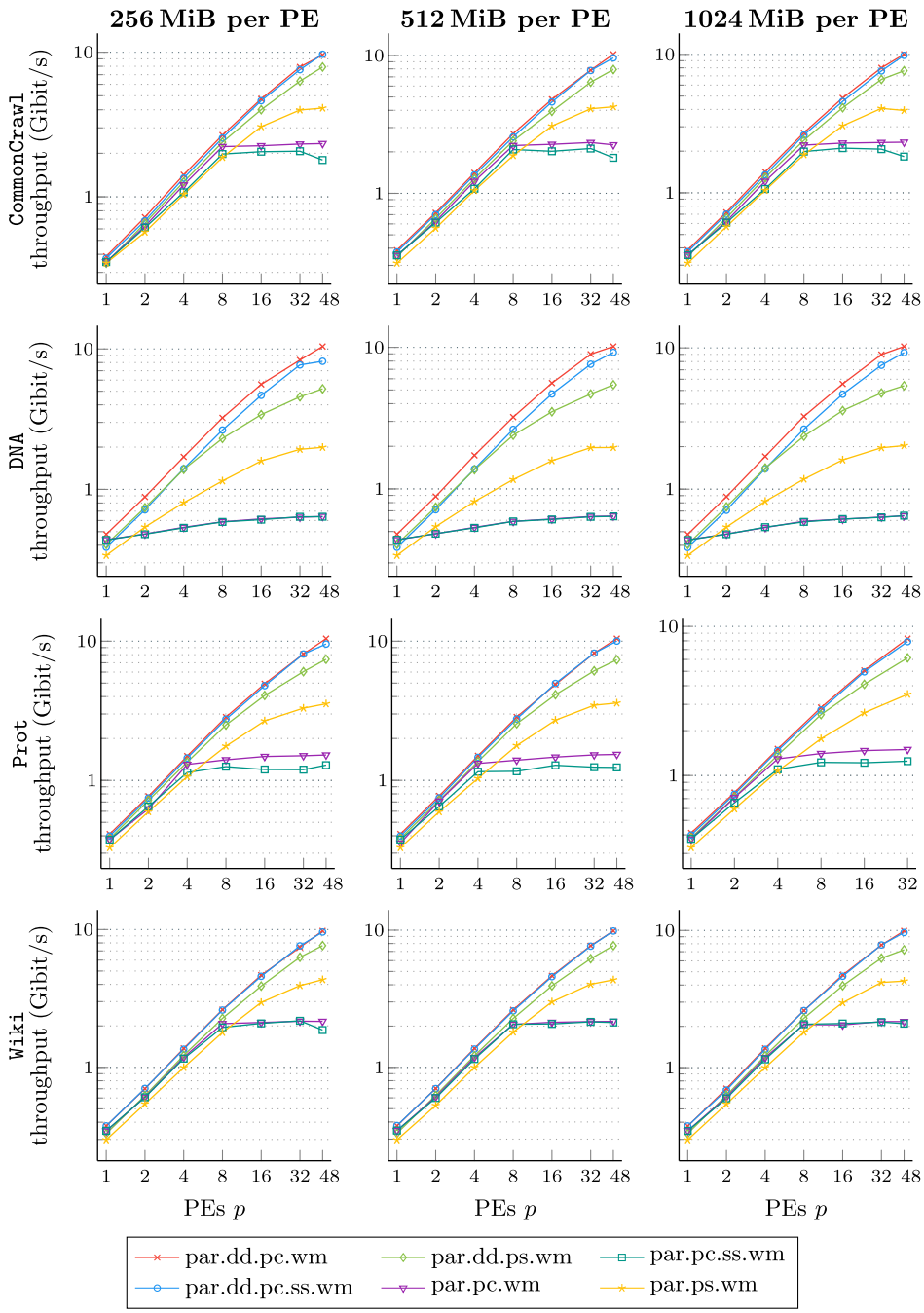


Fig. 35. Weak scaling parallel wavelet *matrix* construction experiments.

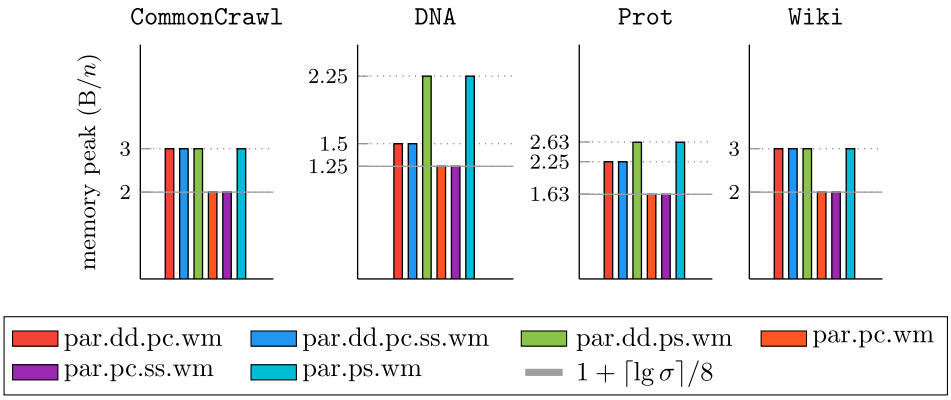


Fig. 36. Memory peaks of the parallel wavelet *matrix* construction algorithms using 48 PEs and 256 MiB input per PE. We use small inputs to have results for par.dd, still we could not include par.sort. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

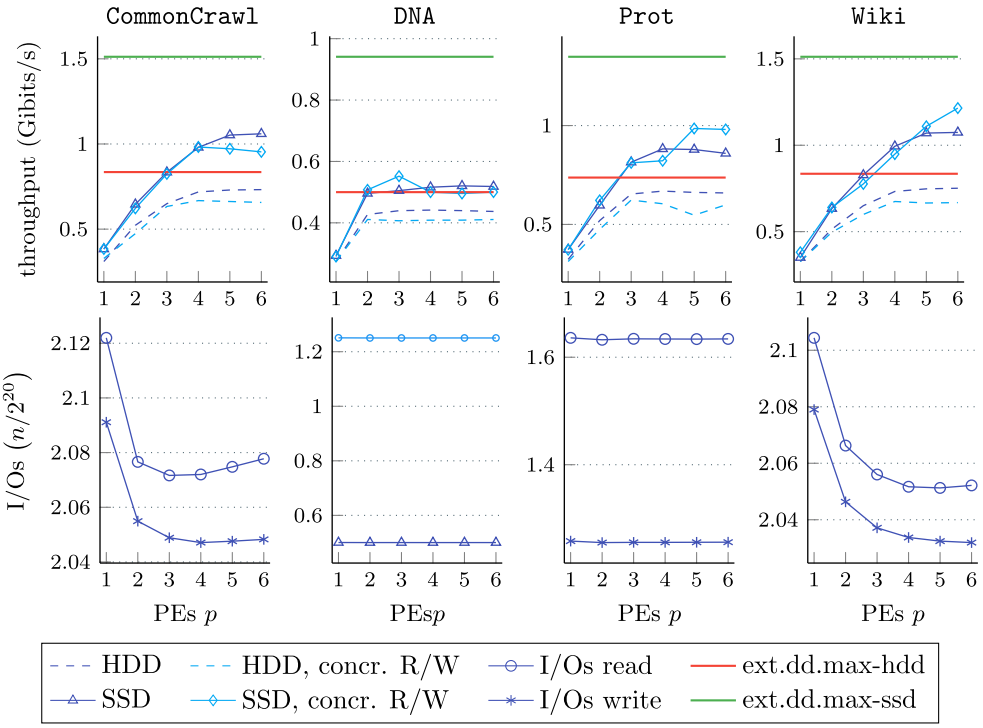


Fig. 37. Throughput of our parallel wavelet *matrix* construction algorithm for inputs of size 20 GiB per PE.

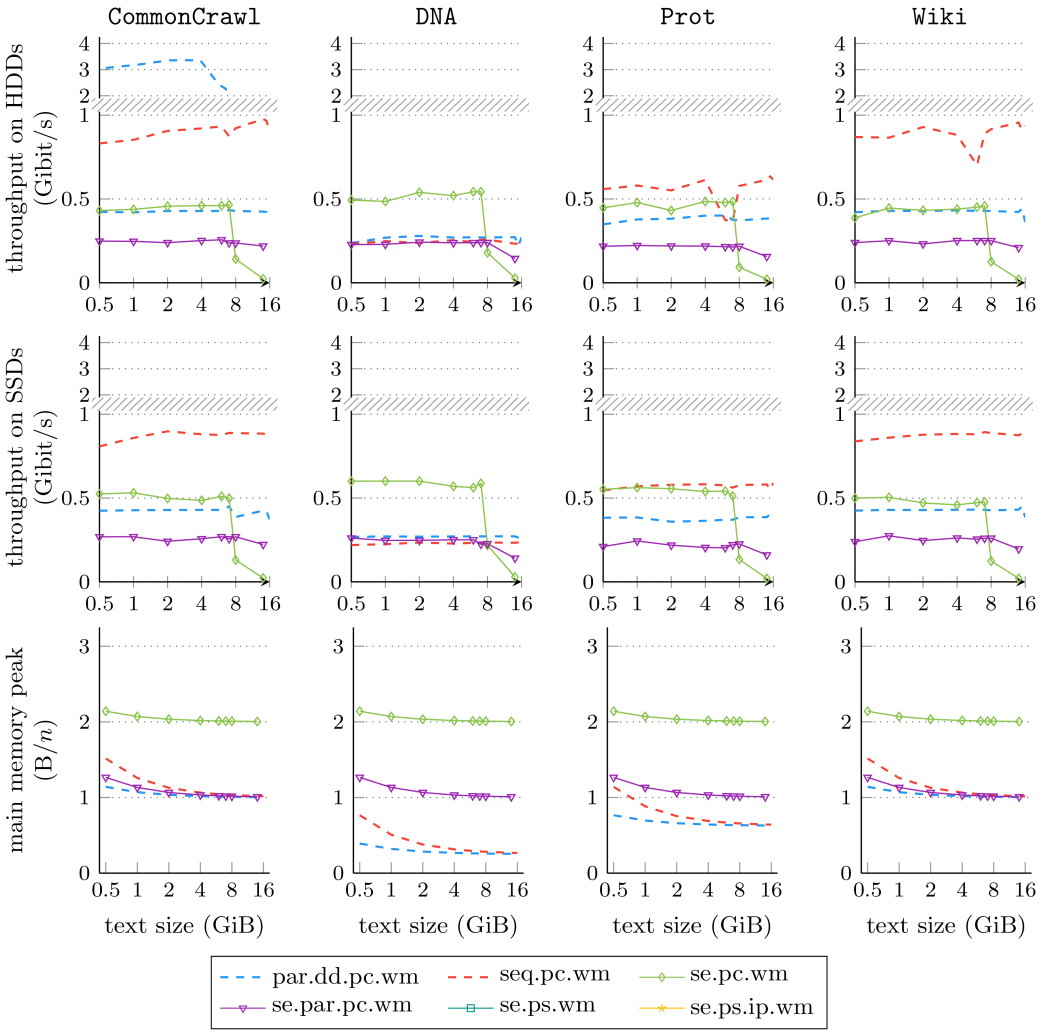


Fig. 38. Throughput and main memory peaks of the semi-external wavelet *matrix* construction algorithms when using the HDDs (first row) and when using the SSDs (second row). The parallel algorithms are using all six PEs. In the last row, we give the main memory peak, which is independent of the used 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.

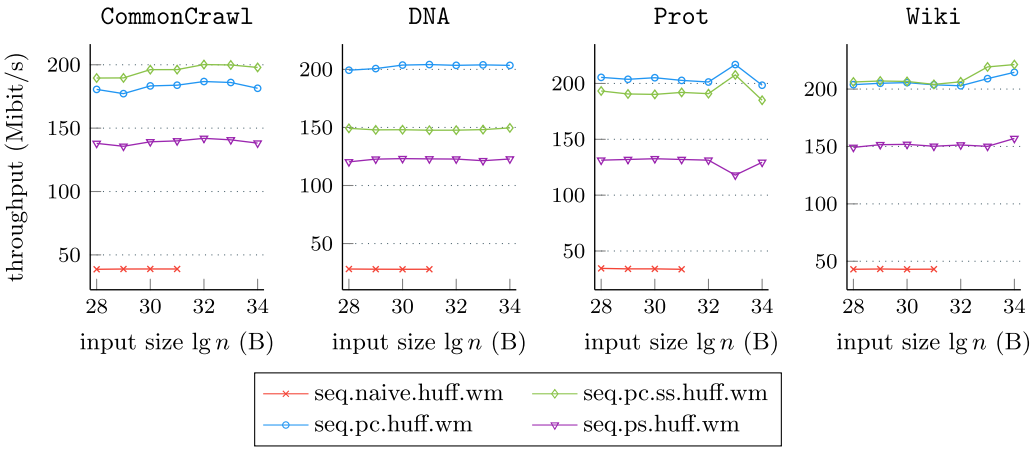


Fig. 39. Throughput of the sequential Huffman-shaped wavelet *matrix* construction algorithms.

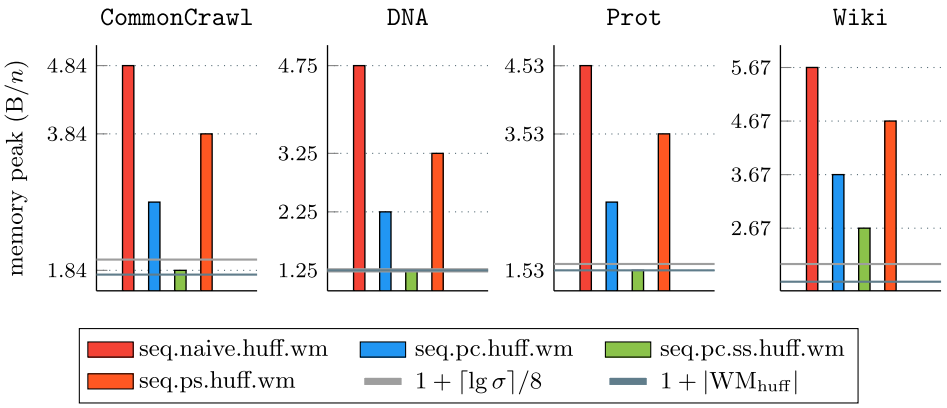


Fig. 40. Snapshot of the memory peaks of sequential wavelet *matrix* construction algorithms for $n = 2^{31}$. Also depicted is the memory required by the text and by the *normal*-shaped wavelet matrix ($1 + \lceil \lg \sigma \rceil / 8$ bytes per character) and the text and the Huffman-shaped wavelet matrix ($1 + |\mathrm{WT}_{\mathrm{huff}}|$).

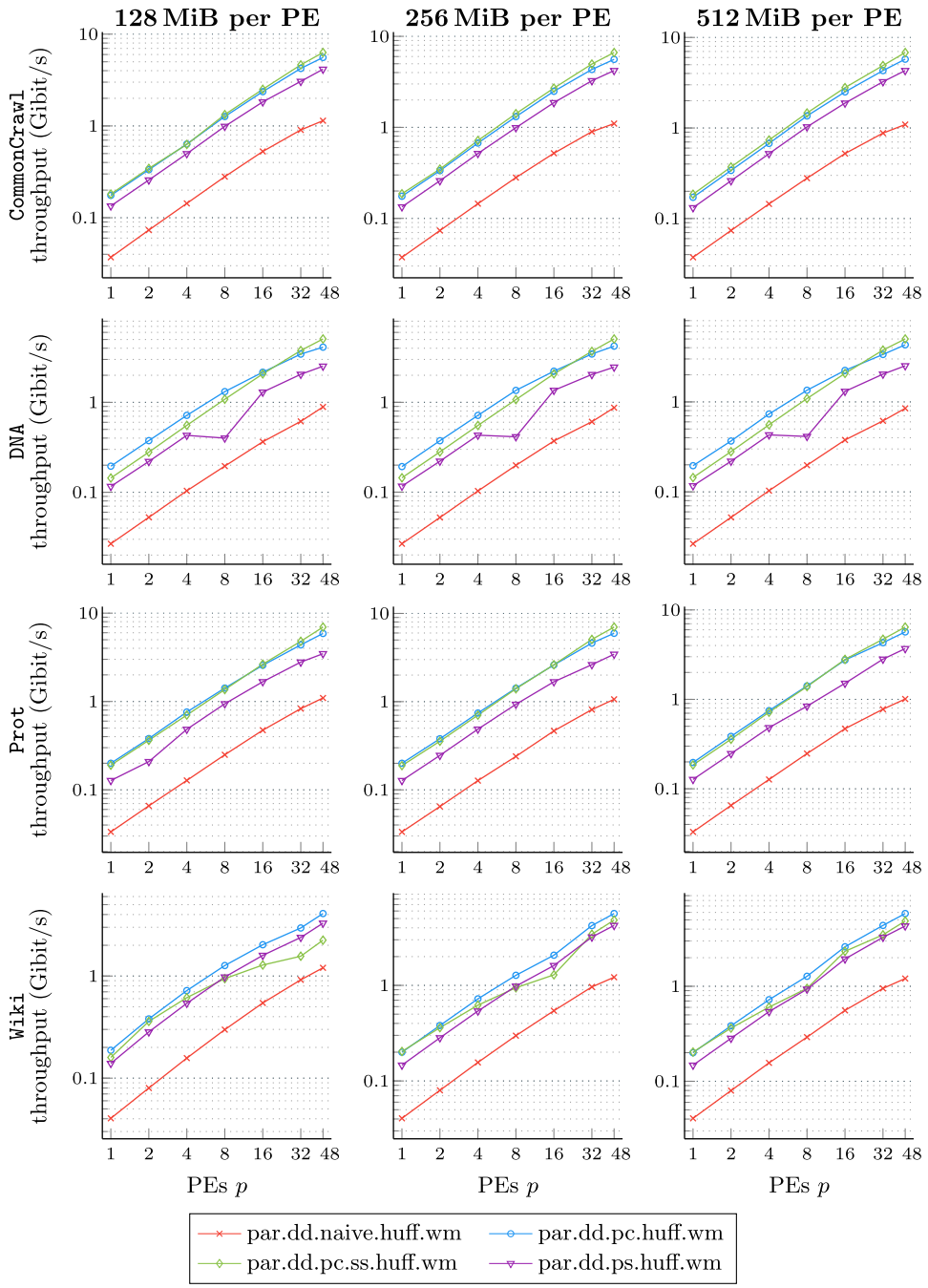


Fig. 41. Weak scaling Huffman-shaped wavelet *matrix* construction experiments.

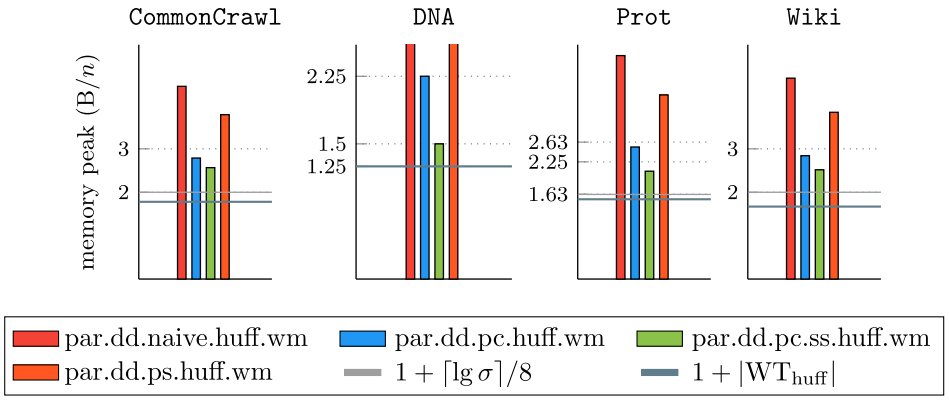


Fig. 42. Memory peaks of the parallel wavelet *matrix* construction algorithms using 48 PEs and 256 MiB input per PE. The memory required just for the input text and the wavelet tree ($1 + \lceil \lceil \lg \sigma \rceil / 8 \rceil$ bytes per character) is also shown.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. We also want to thank Julian Shun, who worked as Reproducibility Referee for this article, for his efforts.

REFERENCES

[1] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127. DOI: https://doi.org/10.1145/48529.48535

[2] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. 2015. Wavelet trees meet suffix trees. In *Proceedings of SODA*. SIAM, 572–591. DOI: https://doi.org/10.1137/1.9781611973730.39

[3] Johannes Bader, Simon Gog, and Matthias Petri. 2016. Practical variable length gap pattern matching. In *Proceedings of SEA* (LNCS, Vol. 9685). Springer, 1–16. DOI: https://doi.org/10.1007/978-3-319-38851-9_1

[4] Sudip Biswas, Tsung-Han Ku, Rahul Shah, and Sharma V. Thankachan. 2017. Position-restricted substring searching over small alphabets. *J. Discrete Algorithms* 46-47, 36–39. DOI: https://doi.org/10.1016/j.jda.2017.10.001

[5] Michael Burrows and David J. Wheeler. 1994. *A Block-Sorting Lossless Data Compression Algorithm*. Technical Report.

[6] Henri Casanova, Arnaud Legrand, and Yves Robert. 2008. *Parallel Algorithms*. CRC Press.

[7] Francisco Claude and Gonzalo Navarro. 2008. Practical rank/select queries over arbitrary sequences. In *Proceedings of SPIRE* (LNCS, Vol. 5280). Springer, 176–187. DOI: https://doi.org/10.1007/978-3-540-89097-3_18

[8] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.* 47 (2015), 15–32. DOI: https://doi.org/10.1016/j.is.2014.06.002

[9] Francisco Claude, Patrick K. Nicholson, and Diego Seco. 2011. Space efficient wavelet tree construction. In *Proceedings of SPIRE* (LNCS, Vol. 7024). Springer, 185–196. DOI: https://doi.org/10.1007/978-3-642-24583-1_19

[10] Paulo G. S. da Fonseca and Israel B. F. da Silva. 2017. Online construction of wavelet trees. In *Proceedings of SEA* (LIPIcs, Vol. 75). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 16:1–16:14. DOI: https://doi.org/10.4230/LIPIcs.SEA.2017.16

[11] Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: standard template library for XXL data sets. *Softw. Pract. Exp.* 38, 6 (2008), 589–637. DOI: https://doi.org/10.1002/spe.844

[12] Patrick Dinklage, Johannes Fischer, and Florian Kurpicz. 2020. Constructing the wavelet tree and wavelet matrix in distributed memory. In *Proceedings of ALENEX*. SIAM, 214–228. DOI: https://doi.org/10.1137/1.9781611976007.17

[13] Jonas Ellert and Florian Kurpicz. 2019. Parallel external memory wavelet tree and wavelet matrix construction. In *Proceedings of SPIRE* (LNCS, Vol. 11811). Springer, 392–406. DOI: https://doi.org/10.1007/978-3-030-32686-9_28

[14] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. 2009. The myriad virtues of wavelet trees. *Inf. Comput.* 207, 8 (2009), 849–866. DOI: https://doi.org/10.1016/j.ic.2008.12.010

[15] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581. DOI: https://doi.org/10.1145/1082036.1082039

[16] Johannes Fischer, Florian Kurpicz, and Marvin Löbel. 2018. Simple, fast and lightweight parallel wavelet tree construction. In *Proceedings of ALENEX*. SIAM, 9–20. DOI: https://doi.org/10.1137/1.9781611975055.2

[17] José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. 2017. Parallel construction of wavelet trees on multicore architectures. *Knowl. Inf. Syst.* 51, 3 (2017), 1043–1066. DOI: https://doi.org/10.1007/s10115-016-1000-6

[18] Travis Gagie. 2006. Large alphabets and incompressibility. *Inf. Process. Lett.* 99, 6 (2006), 246–251. DOI: https://doi.org/10.1016/j.ipl.2006.04.008

[19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *Proceedings of SEA* (LNCS, Vol. 8504). Springer, 326–337. DOI: https://doi.org/10.1007/978-3-319-07959-2_28

[20] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. 2019. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica* 81, 4 (2019), 1370–1391. DOI: https://doi.org/10.1007/s00453-018-0475-9

[21] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of SODA*. ACM/SIAM, 841–850.

[22] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. 2011. Wavelet trees: From theory to practice. In *Proceedings of CCP*. IEEE Computer Society, 210–221. DOI: https://doi.org/10.1109/CCP.2011.16

[23] Torben Hagerup. 1998. Sorting and searching on the word RAM. In *Proceedings of STACS* (LNCS, Vol. 1373). Springer, 366–398. DOI: https://doi.org/10.1007/BFb0028575

[24] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 9 (1952), 1098–1101. DOI: https://doi.org/10.1109/JRPROC.1952.273898

[25] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

[26] Yusaku Kaneta. 2018. Fast wavelet tree construction in practice. In *Proceedings of SPIRE* (LNCS, Vol. 11147). Springer, 218–232. DOI: https://doi.org/10.1007/978-3-030-00479-8_18

[27] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. 2009. Permuted longest-common-prefix array. In *Proceedings of CPM* (LNCS, Vol. 5577). Springer, 181–192. DOI: https://doi.org/10.1007/978-3-642-02441-2_17

[28] Dominik Kempa and Tomasz Kociumaka. 2019. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of STOC*. ACM, 756–767. DOI: https://doi.org/10.1145/3313276.3316368

[29] Dominik Kempa and Tomasz Kociumaka. 2020. Resolution of the burrows-wheeler transform conjecture. In *Proceedings of FOCS*. IEEE, 1002–1013. DOI: https://doi.org/10.1109/FOCS46700.2020.00097

[30] S. Rao Kosaraju and Giovanni Manzini. 1999. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comput.* 29, 3 (1999), 893–911. DOI: https://doi.org/10.1137/S0097539797331105

[31] Julian Labeit, Julian Shun, and Guy E. Blelloch. 2017. parallel lightweight wavelet tree, suffix array and FM-index construction. *J. Discrete Algorithms* 43 (2017), 2–17. DOI: https://doi.org/10.1016/j.jda.2017.04.001

[32] Ben Langmead and Steven L. Salzberg. 2012. Fast gapped-read alignment with Bowtie 2. *Nature Methods* 9, 4 (2012), 357. DOI: https://doi.org/10.1038/nmeth.1923

[33] Veli Mäkinen and Gonzalo Navarro. 2007. Rank and select revisited and extended. *Theor. Comput. Sci.* 387, 3 (2007), 332–347. DOI: https://doi.org/10.1016/j.tcs.2007.07.013

[34] Christos Makris. 2012. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.* 9, 2 (2012), 585–625. DOI: https://doi.org/10.2298/CSIS110606004M

[35] Udi Manber and Eugene W. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948. DOI: https://doi.org/10.1137/0222058

[36] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! but at what COST? In *Proceedings of HotOS*. USENIX Association.

[37] Message Passing Interface Forum. 1994. MPI: A Message-Passing Interface Standard. Retrieved from https://www.mpi-forum.org.

[38] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. 2016. Fast construction of wavelet trees. *Theor. Comput. Sci.* 638 (2016), 91–97. DOI: https://doi.org/10.1016/j.tcs.2015.11.011

[39] Gonzalo Navarro. 2014. Wavelet trees for all. *J. Discrete Algorithms* 25 (2014), 2–20. DOI: https://doi.org/10.1016/j.jda.2013.07.004

[40] Gonzalo Navarro. 2016. *Compact Data Structures—A Practical Approach*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9781316588284

[41] Gonzalo Navarro and Alberto Ordóñez Pereira. 2013. Compressing Huffman models on large alphabets. In *Proceedings of DCC*. IEEE, 381–390. DOI: https://doi.org/10.1109/DCC.2013.46

[42] Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *Proceedings of ICCS* (LNCS, Vol. 3036). Springer, 1–9. DOI: https://doi.org/10.1007/978-3-540-24685-5_1

[43] Robert Sedgewick. 1998. *Algorithms in C++—Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley-Longman.

[44] Julian Shun. 2015. Parallel wavelet tree construction. In *Proceedings of DCC*. IEEE, 63–72. DOI: https://doi.org/10.1109/DCC.2015.7

[45] Julian Shun. 2020. Improved parallel construction of wavelet trees and rank/select structures. *Inf. Comput.* 273 (2020), 104516. DOI: https://doi.org/10.1016/j.ic.2020.104516

[46] German Tischler. 2011. On wavelet tree construction. In *Proceedings of CPM* (LNCS, Vol. 6661). Springer, 208–218. DOI: https://doi.org/10.1007/978-3-642-21458-5_19

[47] German Tischler. 2017. Faster average case low memory semi-external construction of the burrows-wheeler transform. *Math. Comput. Sci.* 11, 2 (2017), 159–176. DOI: https://doi.org/10.1007/s11786-017-0296-2

[48] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111. DOI: https://doi.org/10.1145/79173.79181

[49] Da Zheng, Disa Mhembere, Vince Lyzinski, Joshua T. Vogelstein, Carey E. Priebe, and Randal C. Burns. 2017. Semi-external memory sparse matrix multiplication for billion-node graphs. *IEEE Trans. Parallel Distributed Syst.* 28, 5 (2017), 1470–1483. DOI: https://doi.org/10.1109/TPDS.2016.2618791

Received July 2020; revised February 2021; accepted March 2021