Fast and Simple Parallel Wavelet Tree and Matrix Construction*

Johannes Fischer[1] and Florian Kurpicz[1]

1 **Technische Universität Dortmund, Department of Computer Science,**
johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de

Abstract

The wavelet tree (Grossi et al. [SODA, 2003]) and wavelet matrix (Claude et al. [Inf. Syst., 47:15–32, 2015]) are compact indices for texts over an alphabet $[0, \sigma)$ that support *rank*, *select* and *access* queries in $O(\lg \sigma)$ time. We first present new practical sequential and parallel algorithms for wavelet matrix construction. Their unifying characteristics is that they construct the wavelet matrix bottom-up, i.e., they compute the last level first. We also show that this bottom-up construction can easily be adapted to wavelet *trees*. In practice, our best sequential algorithm is up to twice as fast as the currently fastest sequential construction algorithm (serialWT), simultaneously saving a factor of 2 in space. On 4 cores, our best parallel algorithm is at least twice as fast as the currently fastest parallel algorithm (recWT), while also using less space. This scales up to 32 cores, where we are about equally fast as recWT, but still use only about 75 % of the space. An additional theoretical result shows how to adapt any wavelet *tree* construction algorithm to the wavelet *matrix* in the same (asymptotic) time, using only little extra space.

1998 ACM Subject Classification E.4 CODING AND INFORMATION THEORY

Keywords and phrases text indexing, compressed data structures, parallel algorithms, wavelet matrix, wavelet tree

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The *wavelet tree* (WT), introduced in 2003 by Grossi et al. [7], is a space-efficient data structure that can answer *access*, *rank*, and *select* queries for a text over an alphabet $\Sigma = [0, \sigma)$ in $\mathcal{O}(\lg \sigma)$ time, requiring $\mathcal{O}(n \lg \sigma)$ bits space and additional rank and select data structures on bit vectors. WTs are often utilized for compression [8, 12]. A detailed overview of the history of wavelet trees and many of their applications (not only for text indexing) are given in detail by Ferragina et al. [4] and Navarro [14].

A variant of the WT, the *wavelet matrix* (WM), was introduced in 2011 by Claude and Navarro [2] and is also a compact index for texts that supports the access, rank and select queries. Asymptotically it requires the same space and it has the same query times – $\mathcal{O}(\lg \sigma)$ – for access, rank and select queries as a WT. But in practice the WM is often faster than a WT for rank and select queries [2], as it saves one call to a binary rank/select data structure per query.

* This work was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

Related Work There exists lots of theoretical work when it comes to WT construction. One task is reducing the construction time of WTs below $\mathcal{O}(n \lg \sigma)$. Babenko et al. [1] and Munro et al. [13] independently obtained a construction time of $\mathcal{O}\big(n\lceil \lg \sigma / \sqrt{\lg n}\rceil\big)$. Recently, Shun [18] has parallelized the word packing approach by Babenko et al. [1] to decrease the time for parallel WT construction to $\mathcal{O}(\sigma + \lg n)$ requiring $\mathcal{O}(n\lceil \lg \sigma / \lg n\rceil)$ work. Another important ratio is the additional space required. Claude et al. [3] and Tischler [19] showed how to reduce the additional space required during the construction of the wavelet tree.

Due to the ubiquity of multi-core processors, there is a need for shared memory parallel construction algorithms for WTs and WMs. Fuentes-Sepúlveda et al. [5] described the first practical parallel WT-construction algorithm, requiring $\mathcal{O}(n)$ time and $\mathcal{O}(n \lg \sigma)$ work. Faster practical approaches have been presented by Shun [17] and Labeit et al. [9], both requiring $\mathcal{O}(\lg n \lg \sigma)$ time and $\mathcal{O}(n \lg \sigma)$ work. When it comes to WMs, there is not much work directly dedicated to it. Sometimes, when a WT-construction algorithm is presented, it is mentioned that the algorithm can also be adopted to compute the WM, e.g., [17, 18], but there are no dedicated (practical) parallel WM-construction algorithms. The only (sequential and semi-external) implementation of a wavelet matrix construction algorithm can be found in the SDSL (succinct data structure library) [6].

Our Contribution First, we present two sequential and parallel WM-construction algorithms, which can also easily be adapted to compute the WT. This results in the fastest sequential WM- and WT-construction algorithms (*psWM* and *psWT*) that are up to twice as fast as *serialWT* [17], the previously fastest implementation, while requiring only half as much space. Next, we parallelize our algorithms and obtain the fastest parallel WM- and WT-construction algorithms for up to 32 cores. Utilizing more than 32 cores, *recWT* [9] (the fastest parallel WT-construction algorithm) remains faster. Last, we show that the WT and the WM are equivalent, in the sense that every algorithm that can compute the former can also compute the latter in the same time with only $n + \sigma + 2\sigma\lceil \lg n\rceil + o(n + \sigma)$ bits of additional space.

2 Preliminaries

Let $\mathsf{T} = \mathsf{T}[0] \dots \mathsf{T}[n-1]$ be a text of length n over an alphabet $\Sigma = [0, \sigma)$. Each character $\mathsf{T}[i]$ can be represented using $\lceil \lg \sigma \rceil$ bits. In this paper, the *most significant bit* (MSB) is the leftmost bit and the *least significant bit* (LSB) is the rightmost bit. We denote this binary representation of a character $\alpha \in \Sigma$ as $\text{bits}(\alpha)$, e.g. $\text{bits}(3) = (\texttt{011})_2$. Whenever we write a binary representation of a value, we indicate it by a subscript two. The k-th bit (from MSB to LSB) of a character α is denoted by $\text{bit}(k, \alpha)$ for all $0 \le k < \lceil \lg \sigma \rceil$. Given $\alpha \in \Sigma$, the *bit prefix* of size k of α are the k most significant bits, i.e., $\text{prefix}(k, \alpha) = (\text{bit}(0, \alpha) \dots \text{bit}(k-1, \alpha))_2$. *Reversing* the significance of the bits is denoted by reverse, e.g. $\text{reverse}((\texttt{001})_2) = (\texttt{100})_2$. We interpret sequences of bits as integer values.

The *bit-reversal* permutation[1] of length k (denoted by ρ_k) is a permutation of $[0, 2^k)$ with $\rho_k(i) = (\text{reverse}(\text{bits}(i)))_2$. For example, $\rho_4 = (0, 2, 1, 3) = ((\texttt{00})_2, (\texttt{10})_2, (\texttt{01})_2, (\texttt{11})_2)$. ρ_k and ρ_{k+1} can be computed from another, as $\rho_{k+1} = (2\rho_k(0), \dots, 2\rho_k(2^k - 1), 2\rho_k(0) + 1, \dots, 2\rho_k(2^k - 1) + 1)$ and $\rho_k = (\rho_{k+1}(0)/2, \dots, \rho_{k+1}(2^k - 1)/2)$, where we can realize the division by a single bit shift.

Given a bit vector BV of size n, the operation $\text{rank}_0(\mathsf{BV}, i)$ returns the number of 0's in BV up to position i whereas $\text{select}_0(\mathsf{BV}, i)$ asks for the position of the i-th 0 in BV. The

[1] `http://oeis.org/A030109`, last accessed 14.02.2017.

operations $\text{rank}_1(\mathsf{BV}, i)$ and $\text{select}_1(\mathsf{BV}, i)$ work analogously. We omit to name the bit vector if it is clear where the operation is executed.

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 with $\mathsf{B}[0] = 0$ and $\mathsf{B}[i] = \mathsf{A}[i-1] + \mathsf{B}[i-1]$ for all $i \in [1, n)$.[2] In parallel, the prefix sum can be computed in $\mathcal{O}(\lg n)$ time and $\mathcal{O}(n)$ work.

2.1 Wavelet Trees

Given a text T of length n over an alphabet $\Sigma = [0, \sigma)$, the *wavelet tree* (WT) of T is a complete balanced binary tree. Each node of WT represents characters in $[\ell, r) \subseteq [0, \sigma)$. The root of WT represents characters in $[0, \sigma)$, i.e., all characters. The left and right child of a node that represents characters in $[\ell, r)$ represent the characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r)$, resp. A node is a leaf if $\left|\mathsf{T}_{[\ell,r)}\right| \leq 2$, with $\mathsf{T}_{[\ell,r)} = \{\mathsf{T}[i] : 0 \leq i < n \text{ and } \mathsf{T}[i] \in [\ell, r)\}$. The characters in $[\ell, r)$ at a node v are represented using a bit vector BV'_v such that the i-th bit in BV'_v is $\text{bit}\big(h(v), \mathsf{T}_{[\ell,r)}[i]\big)$, where $h(v)$ is the height of v in WT, i.e., the length of the path from the root to v.

There are two variants of the WT: the *pointer-based* and the *level-wise* WT. The pointer-based WT utilizes pointers to represent the tree structure. In addition, each node v stores a pointer to the bit vector BV'_v, see Figure 1a. In the level-wise WT, we concatenate the bit vectors of all nodes with the same height in a pointer-based WT. Therefore, we store only a single bit vector BV'_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$, see Figure 1b. This retains the functionality from the pointer-based WT [10, 11]. Characters represented by one node of the pointer-based WT form a continuous interval in BV'_ℓ for each level ℓ. Furthermore, given such an interval $[a, b]$ in BV'_k where the characters in $[\ell, r) \subseteq \Sigma$ are represented, the intervals where the characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r)$ are represented in BV'_{k+1} are subintervals of $[a, b]$. The interval of a WT at which a character is represented at level ℓ is encoded by its bit prefix of length ℓ.

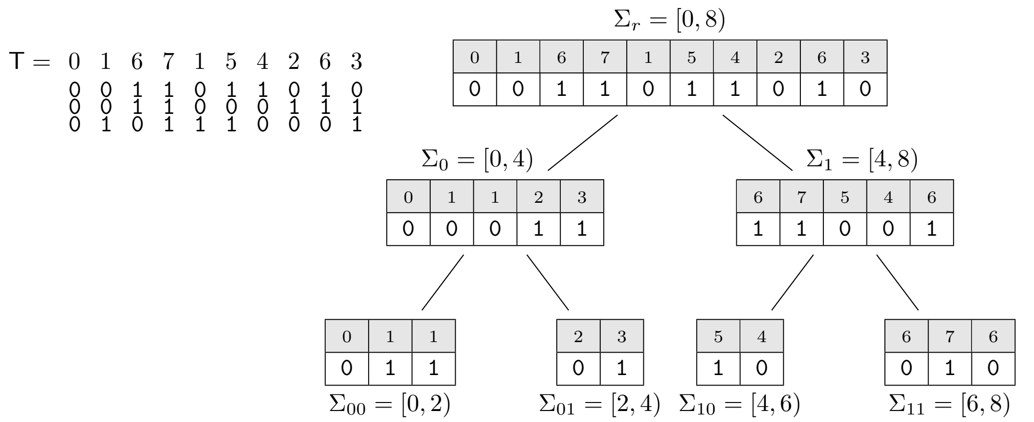
▶ Observation 1 (Fuentes-Sepulveda et al. [5]). Given a character $\mathsf{T}[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the WT, the interval pertinent to $\mathsf{T}[i]$ in BV'_ℓ can be computed by $\text{prefix}(\ell, \mathsf{T}[i])$.

The wavelet tree (both variants) can be used to generalize the operations access, rank and select to alphabets of size σ. Answering these queries requires $\mathcal{O}(\lg \sigma)$ time. To do so, the bit vectors are augmented by a rank and select data structure. We point to [2] for a detailed description of the operations. In the following, we work with the level-wise WT.

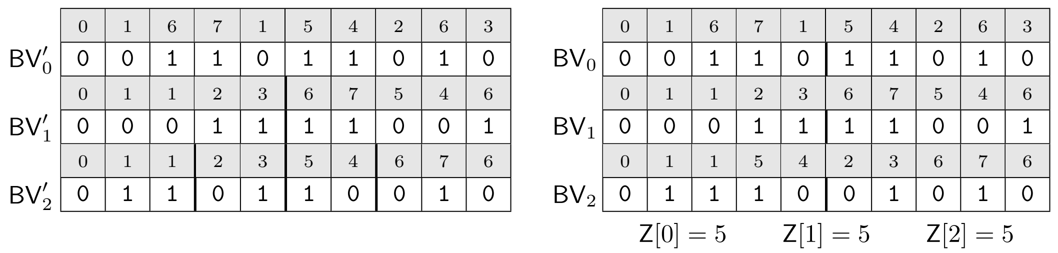
2.2 Wavelet Matrices

The *wavelet matrix* (WM) [2] works similar to a level-wise WT. However, we discard the tree structure, i.e., the parent-child relation and thus the condition that each character is represented in an interval that is covered by the character's interval in the previous level. Again, we have a bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$. In addition to the bit vectors, we store the number of zeros for each level ℓ (denoted by $\mathsf{Z}[\ell]$). BV_0 contains the MSBs of each character in T in text order (it is the same as the first level of a WT). Our new WM-algorithms are based on the following observation, similar to Observation 1: If a character α is represented at position i in BV_ℓ, then the position of its $(\ell+1)$-th MSB in $\mathsf{BV}_{\ell+1}$ depends on $\mathsf{BV}_\ell[i]$. Namely, if $\mathsf{BV}_\ell[i] = 0$, $\text{bit}(\ell + 1, \alpha)$ is stored at position $\text{rank}_0(\mathsf{BV}_\ell, i)$; otherwise

[2] If not zero based, B is usually defined as $\mathsf{B}[0] = \mathsf{A}[0]$ and $\mathsf{B}[i] = \mathsf{A}[i-1] + \mathsf{B}[i-1]$ for all $i \in [1, n)$.



(a) The text T and its binary representation on the left-hand side and the pointer-based WT of T on the right-hand side. Σ_α for $\alpha \in \{r, 0, 1, 00, 01, 10, 11\}$ denotes the characters that are represented by the bit vector.



(b) The level-wise WT of T. Thick lines are borders of the intervals corresponding to the nodes.

(c) The WM of T. The thick lines highlight the number of zeros at each level.

Figure 1 The pointer-based, level-wise WT, and the WM for T= 0167154263 over $\Sigma = [0, 8)$. The light gray ( ) arrays contain the characters represented at the position and are not part of the WT and WM.

($\mathsf{BV}_\ell[i] = 1$), it is stored at position $\mathsf{Z}[\ell] + \text{rank}_1(\mathsf{BV}_\ell, i)$. In other words, $\mathsf{BV}_\ell[i] = \text{bit}(\ell, \mathsf{T}'[i])$, i.e., the ℓ-th MSB of the i-th character of T' in text order, where T' is T stably sorted using the reversed bit prefixes of length ℓ of the characters as key. Similar to the intervals in BV'_ℓ of the WT, characters of T form intervals in BV_ℓ of the WM. Again, the intervals at level ℓ correspond to bit prefixes of size ℓ but due to the construction of the WM we consider the reversed bit prefixes.

▶ Observation 2. Given a character $\mathsf{T}[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the WM, the interval pertinent to $\mathsf{T}[i]$ in BV_ℓ can be computed by reverse(prefix($\ell, \mathsf{T}[i]$)).

As with WTs, if the bit vectors are augmented by (binary) rank and select data structures, the WM can be used to answer access, rank and select queries in $\mathcal{O}(\lg \sigma)$ time. We refer to [2] for a detailed description of these queries. For an example of a WT see Figure 1c.

3 New Wavelet Matrix Construction Algorithms

Throughout this section, let T be a text of length n over an alphabet $\Sigma = [0, \sigma)$. As shown in Observation 2, each level ℓ of the WM contains disjunct intervals corresponding to the reversed length-ℓ bit prefixes of the characters in T. This enables us to start on the *last* level $\lceil \lg \sigma \rceil - 1$, and then iteratively work through the other levels in a *bottom-up* manner until

Algorithm 1: Sequential WM Construction with Prefix Counting (pcWM)

```
1  function pcWM(text T, size n, size of alphabet σ)
2      for i = 0 to n − 1 do
3          Hist[T[i]]++                          // Compute histogram of the characters in T and
4          BV_0[i] = Bit(0, T[i])                // fill the first level of the WM.
5      for i = 0 to 2^(⌈lg σ⌉−1) − 1 do
6          Z[⌈lg σ⌉ − 1] = Z[⌈lg σ⌉ − 1] + Hist[2i]     // Number of 0s in the last level.
7      for ℓ = ⌈lg σ⌉ − 1 to 1 do
8          for i = 0 to 2^ℓ − 1 do
9              Hist[i] = Hist[2i] + Hist[2i + 1]      // Update the histogram for the next level.
10         for i = 1 to 2^ℓ − 1 do
11             SPos[ρ_ℓ(i)] = SPos[ρ_ℓ(i − 1)] + Hist[ρ_ℓ(i − 1)] // Compute new starting positions.
12         Z[ℓ − 1] = SPos[1]                    // Number of 0s is the position of the first 1.
13         for i = 0 to n − 1 do
14             pos = SPos[prefix(ℓ, T[i])]++         // Get starting position for the bit prefix,
15             BV_ℓ[pos] = bit(ℓ, T[i])              // update it, and set the bit in the bit vector.
```

the matrix is fully constructed. To get this process started, we need to know the borders of the intervals on the last level, for which we must first compute the *histogram* of the text characters (as in the first phase of *counting sort*). On subsequent levels $\ell < \lceil \lg \sigma \rceil$ we utilize the fact that we can quickly compute the histograms of the considered bit prefixes of size ℓ from the histogram of bit prefixes of size $\ell + 1$, *without rescanning the text*. This and the fact that we never actually *sort* the input text T is the main distinguishing feature of our new algorithms from the previous ones. We assume that arrays are initialized with 0s.

3.1 Sequential Wavelet Matrix Construction Algorithms

Our first WM-construction algorithm (*pcWM*, see Algorithm 1) starts with the computation of the number of occurrences of each character in T to fill the initial histogram $\mathsf{Hist}[0, \sigma)$, see line 3. In addition, the first level of the WM is computed, as it contains the MSBs of all characters in text order (line 4). This requires $\mathcal{O}(n)$ time and $\sigma \lceil \lg n \rceil$ bits space for the histogram. Later on we require additional $\sigma \lceil \lg n \rceil$ bits to store the starting positions of the intervals (see SPos in Algorithm 1). Using the histogram, we can also compute the number of 0s in the last level of the WM, i.e., total number of characters with a 0 as LSB (line 6). Since the histogram contains σ entries this requires $\mathcal{O}(\sigma)$ time and no additional space.

Next, we compute the bit vectors and number of zeros for each other level, starting with the last one (see loop starting at line 7). Initially, we have a histogram for all characters in T. During each iteration (each time we want to compute level ℓ) we require the histogram for all bit prefixes of size $\ell - 1$ of the characters in T. Therefore, if we have the histogram of bit prefixes of size ℓ, we can simply compute the histogram of the bit prefixes of size $\ell - 1$ by ignoring the last bit of the current prefix, e.g., the amount of characters with bit prefix $(\mathtt{01})_2$ is the total number of characters with bit prefixes $(\mathtt{010})_2$ and $(\mathtt{011})_2$. We can do so in $\mathcal{O}(\sigma)$ time requiring no additional space, as we already stored the histogram for σ characters and can reuse the space, see line 9.

Using the updated histogram, 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. The starting position of the interval representing characters with bit prefix 0 is always 0, therefore we only compute the starting positions for all other bit prefixes, see line 10. To be able to access them by their bit prefix, we need to compute the prefix sum in bit-reversal permutation order, see line 11. Again, this requires $\mathcal{O}(\sigma)$ time and no additional space, as we already have considered the space to save the starting positions of the intervals. Using the starting positions of the intervals, i.e., the prefix sum over the histogram, we can easily get the number of zeros in the level above by looking at the number of even bit prefixes, see line 12.

Last, we need to compute the bit vector for the current level ℓ. To do so, we simply scan T once from left to right and consider the bit prefix of length $\ell - 1$ of each character. Since we have computed the position in the bit vector where the ℓ-th MSB of the characters needs to be stored, we can simply put it there and increase the position for characters with the same bit prefix by one, see lines 14 and 15. This requires $\mathcal{O}(n)$ time and no additional space.

Since we need to compute $\mathcal{O}(\lg \sigma)$ levels and also store the bit-reversal permutation which requires another $\sigma\lceil\lg n\rceil$ bits of additional space, this results in the following lemma.

▶ **Lemma 1.** *Algorithm* pcWM *computes the* WM *of a text of length n over an alphabet of size σ in $\mathcal{O}(n \lg \sigma)$ time using $3\sigma\lceil\lg n\rceil$ bits of space in addition to the input and output.*

3.2 Parallel Wavelet Matrix Construction Algorithms

The naïve way to parallelize the pcWM algorithm is to parallelize it such that each processor is responsible for the construction of one level of the WM. To this end, each processor needs to first compute the corresponding histogram of the level, and then the resulting starting positions of the intervals. This results in the following Lemma.

▶ **Lemma 2.** *The* WM *can be constructed in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg \sigma)$ work requiring $6\sigma\lceil\lg n\rceil$ bits of space in addition to the input and output.*

The disadvantage of this naïve parallelization is that we cannot efficiently utilize more than $\lceil\lg\sigma\rceil$ processors. To use more processors, instead of parallelizing level-wise, we do the following. Each processor (we denote the number of processors by p) gets a slice of the text of size $\Theta(\frac{n}{p})$ and computes the corresponding slices of the bit vectors on *all* levels. On level ℓ, each processor 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 prefix sum operation, these local histograms are then combined such that in the end each processor knows where to write its bits (arrays $\mathsf{SPos}_c[0, \sigma)$ for $0 \leq c < p$). 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 SPos_c.

This approach works, but it comes with the problem that two or more processors may want to concurrently write bits to the same computer word, resulting in *race conditions*. To avoid these race conditions, one would have to implement mechanisms for exclusive writes, which would result in unacceptably slow running times.

Instead, we do the following. Having computed the arrays of starting positions SPos_c on level ℓ, we use this array to globally *sort* the input text stably in parallel according to its length-ℓ bit prefixes. The resulting sorted text $\mathsf{T}_{\text{sorted}}$ is then again split into slices of size $\Theta(\frac{n}{p})$. Then each processor scans its local slice from left to right and writes the corresponding bits to the bit-vector BV_ℓ. To avoid all race conditions, we further make sure that the size of each slice of the text is a multiple of w, where w is the number of bits in a computer word ($w = 64$ in our implementation).

Algorithm 2: Parallel WM Construction with Prefix Sorting (psWM)

1 **function** *psWM(Text* T*, size* n*, size of alphabet* σ*)*
2   **parfor** $c = 0$ **to** $p-1$ **do**
3   **for** $i = c\frac{n}{p}$ **to** $(c+1)\frac{n}{p}$ **do**
4       $\mathsf{Hist}_c[\mathsf{T}[i]]{+}{+}$ *// Compute histogram of the characters in* T *and*
5       $\mathsf{BV}_0[i] = \mathrm{bit}(0, \mathsf{T}[i])$ *// fill the first level of the WM.*
6   Perform parallel prefix sum with respect to $\rho_{\lceil \lg \sigma \rceil}$ to compute SPos_c
7   $\mathsf{Z}[\lceil \lg \sigma \rceil - 1] = \mathsf{SPos}_0[1]$
8   **for** $\ell = \lceil \lg \sigma \rceil - 1$ **to** 1 **do**
9   **parfor** $c = 0$ **to** $p-1$ **do**
10   **for** $i = 0$ **to** $2^\ell - 1$ **do**
11         $\mathsf{Hist}_c[i] = \mathsf{Hist}_c[2i] + \mathsf{Hist}_c[2i+1]$ *// Update the histogram for the next level.*
12   Perform parallel prefix sum with respect to ρ_ℓ to compute SPos_c
13     $\mathsf{Z}[\ell] = \mathsf{SPos}_0[1]$
14     $\mathsf{T}_{\mathrm{sorted}} = \mathrm{ParallelCountingSort}(\mathsf{T}, \mathsf{SPos})$ *// Sort* T *with respect to bit prefixes and* ρ_ℓ*.*
15   **parfor** $c = 0$ **to** $p-1$ **do**
16   **for** $i = c\frac{n}{p}$ **to** $(c+1)\frac{n}{p}$ **do**
17         $\mathsf{BV}_\ell[i] = \mathrm{bit}(\ell, \mathsf{T}_{\mathrm{sorted}}[i])$ *// Set the bit in the bit vector.*

The resulting algorithm is shown in Algorithm 2. First, each of the p processors computes the local histogram (Hist_c for $c \in [0, p)$) of its slice and, at the same time, fills BV_0 (lines 4 and 5). Next, we compute the local starting positions (SPos_c for $c \in [0, p)$), i.e., the prefix sum of $[\mathsf{SPos}_0[0], \mathsf{SPos}_1[0], \ldots, \mathsf{SPos}_{p-1}[0], \ldots\ldots, \mathsf{SPos}_0[\sigma-1], \mathsf{SPos}_1[\sigma-1], \ldots, \mathsf{SPos}_{p-1}[\sigma-1]]$, with respect to $\rho_{\lceil \lg \sigma \rceil}$, see line 6. All this requires $\mathcal{O}(\lg p + \sigma)$ time, $\mathcal{O}(n + p\sigma)$ work and $3p\sigma\lceil \lg n \rceil$ bits of space using p processors. In line 6 "respect to $\rho_{\lceil \lg \sigma \rceil}$" means that character $\rho_{\lceil \lg \sigma \rceil}(i)$ follows character $\rho_{\lceil \lg \sigma \rceil}(i-1)$ for all $i \in [1, \lceil \lg \sigma \rceil)$. We obtain the number of zeros at the last level during this step, i.e., the position of the first one at the first processor.

Using the information (Hist and SPos), we can compute the corresponding values for all sizes $\ell \in [0, \lceil \lg \sigma \rceil)$ of bit prefixes. For each level (see loop starting at line 8) the time and work required are the same as during the first step. There is no additional space required since we can reuse the space used during the previous iteration.

We use the local starting positions to sort the text, see line 14. Each processor knows the starting positions for its local text. We require additional $n\lceil \lg \sigma \rceil$ bits of space (which can be reused at each level) to store the sorted text $\mathsf{T}_{\mathrm{sorted}}$. After this sorting, each processor can simply insert its bits at the corresponding position in BV_ℓ (last line of Algorithm 2).

This leads to the following lemma.

▶ **Lemma 3.** *Algorithm* psWM *computes the WM of a text of length* n *over an alphabet of size* σ *in* $\mathcal{O}\left(\lg \sigma \left(\frac{n}{p} + \lg p + \sigma\right)\right)$ *time and* $\mathcal{O}(\lg \sigma (n + p\sigma))$ *work requiring* $3p\sigma\lceil \lg n \rceil + n\lceil \lg \sigma \rceil$ *bits of space in addition to the input and output utilizing* p *processors.*

The algorithm can efficiently use up to $p \leq \frac{n}{\sigma}$ processors. Utilizing that many processors yields optimal $\mathcal{O}(n \lg \sigma)$ work with $\mathcal{O}(\lg \sigma\, (\sigma + \lg n))$ time. Using more processors would only increase the required work, without achieving a better running time than on n/σ processors.

Name	n	σ	source	Name	n	σ	Source
XML	$2.1 \cdot 10^8$	96	PC	jdk13c	$6.97 \cdot 10^7$	113	LW
DNA	$2.1 \cdot 10^8$	16	PC	linux-2.4.5.tar	$1.16 \cdot 10^8$	254	LW
ENG	$2.1 \cdot 10^8$	224	PC	rctail96	$1.14 \cdot 10^8$	93	LW
PROT	$2.1 \cdot 10^8$	25	PC	rfc	$1.16 \cdot 10^8$	120	LW
SRC	$2.1 \cdot 10^8$	229	PC	sprot34.dat	$1.09 \cdot 10^8$	66	LW
chr22.dna	$3.5 \cdot 10^7$	5	LW	w3c2	$1.04 \cdot 10^8$	254	LW
etext99	$1.05 \cdot 10^8$	145	LW	random1	$1 \cdot 10^8$	254	RN
gcc-3.0.tar	$8.76 \cdot 10^7$	148	LW	random2	$1 \cdot 10^8$	65534	RN
howto	$3.94 \cdot 10^7$	195	LW	words	$1.4 \cdot 10^8$	2245405	WMT

Table 1 List of texts we used for our experiments. We obtained the texts from the following sources: The *Pizza & Chili corpus* (PC)[3], the *lightweight corpus* (LW)[4], uniformly distributed random numbers (RN), and word based alphabets computed from Russian news articles from 2011 from the *Conference on Machine Translation* (WMT)[5].

Using sorting for the parallel construction of the WT has already been considered by Shun [17] (sortWT). In their approach, the WT is computed from the first level to the last, and for each level the whole text needs to be sorted using the bit prefix as key (comparison based sorting). Our approach uses counting sort and makes use of the fact that we can compute the intervals for the current level using the intervals of the succeeding level.

It should be noted that both algorithms (pcWM and psWM) can be adjusted to compute the level-wise WT instead of the WM. To do so, we just have to replace ρ by the identity permutation in Algorithms 1 and 2. Then, the resulting starting positions of the intervals are for bit prefixes in increasing order, i.e., the starting positions of the intervals for a WT, see Observations 1 and 2.

4 Experiments

We implemented our algorithms *pcWM*, *psWM*, *pcWT* and *psWT* using C++. Due to space constraints we focus on the WM-construction algorithms. The running times of the WT-construction algorithms is nearly the same, see Table 5 in the Appendix. We compiled our code using `g++` 6.2 with flags `-O3` and `-march=native` and provide a tuned sequential implementation, as well as parallel implementations utilizing *openMP* 4.5. Our implementations are available from `https://github.com/kurpicz/pwm`.

We compare with the implementations of Shun [17] (*serialWT* and *levelWT*) and Labeit et al. [9] (*recWT*). Other implementations (as the WM- and WT-construction algorithms in the SDSL) were already proved slower and/or more space consuming. The running times of the construction algorithms implemented in the SDSL are listed in Table 4 in the Appendix. Here, *serialWT* is the fastest sequential WT-construction algorithm and *recWT* is the fastest parallel WT-construction algorithm. Both *serialWT* and *recWT* are parallel WT-construction algorithms utilizing *Cilk Plus* for the parallelization. The code of *serialWT*, *levelWT* and *recWT* has been compiled using their provided makefiles.

[3] `http://pizzachili.dcc.uchile.cl/texts.html`, last accessed 14.02.2017.
[4] `http://people.unipmn.it/manzini/lightweight/corpus/`, last accessed 14.02.2017.
[5] `http://statmt.org/wmt16/translation-task.html`, last accessed 14.02.2017.

	[This paper]				Shun [17]			Labeit et al. [9]	
	pcWM		psWM		serialWT	levelWT		recWT	
Text	T_1	T_4	T_1	T_4	T_1	T_1	T_4	T_1	T_4
XML	2.446	<u>0.724</u>	**2.159**	0.759	3.190	4.920	2.190	5.118	1.389
DNA	1.462	0.550	**1.419**	<u>0.432</u>	2.050	2.840	1.260	3.362	0.904
ENG	2.956	<u>0.844</u>	**2.753**	0.886	4.260	6.230	2.640	6.713	1.809
PROT	1.686	<u>0.525</u>	**1.504**	0.535	3.190	4.380	1.760	5.299	1.426
SRC	2.891	<u>0.838</u>	**2.617**	0.882	4.000	5.910	2.640	6.457	1.731
chr22.dna	0.170	0.108	**0.163**	<u>0.058</u>	0.260	0.363	0.146	0.423	0.113
etext99	1.465	<u>0.416</u>	**1.350**	0.454	2.230	3.170	1.430	3.499	0.949
gcc-3.0.tar	1.208	<u>0.355</u>	**1.099**	0.370	1.610	2.460	1.060	2.568	0.684
howto	0.550	<u>0.160</u>	**0.500**	0.169	0.772	1.130	0.500	1.200	0.324
jdk13c	0.815	<u>0.240</u>	**0.714**	0.253	1.110	1.740	0.777	1.841	0.492
linux-2.4.5.tar	1.617	<u>0.454</u>	**1.464**	0.496	2.190	3.260	1.420	3.529	0.932
rctail96	1.357	<u>0.395</u>	**1.225**	0.421	1.810	2.770	1.220	2.896	0.801
rfc	1.412	<u>0.408</u>	**1.270**	0.428	2.040	2.900	1.280	3.141	0.861
sprot34.dat	1.313	<u>0.387</u>	**1.187**	0.404	1.800	2.750	1.170	2.904	0.793
w3c2	1.431	<u>0.411</u>	**1.354**	0.644	1.880	2.870	1.280	2.956	0.802
random1	1.305	<u>0.377</u>	**1.096**	0.422	3.400	4.350	1.650	5.755	1.538
random2	**3.566**	<u>1.085</u>	6.032	1.732	6.790	6.810	6.830	11.50	3.090
words	**7.303**	<u>3.438</u>	10.72	4.324	11.10	10.90	6.490	17.56	4.733

Table 2 Running times of the algorithms on the PC-system in seconds. T_1 denotes the running time using one core and T_4 denotes the running time using all four cores. The fastest sequential running time is is highlighted using bold font and the fastest parallel running time is underlined.

The measurement of the memory usage of our algorithms and *serialWT* was done using *malloc_count*.[6] The memory usage of all other algorithms was measured using the function *getrusage*, as malloc_count is incompatible with the Cilk Plus implementations. For our experiments we use real-world and artificial texts, see Table 1. We provide a script to collect and prepare all considered corpora at `https://github.com/kurpicz/tcc`.

We conducted our experiments on two different machines.

PC-System equipped with an Intel Core i5-4670 processor (four cores with frequency up to 3.4 GHz and cache sizes: 32 kB L1, 256 kB L2 and 6144 kB L3) and 16 GB RAM.

Server-System equipped with two Intel Xeon E5-2676 processor (16 cores with frequency up to 2.4 GHz and cache sizes: 384 kB L1, 3 MB L2 and 30 MB L3) and 256 GB RAM.

Results – Construction Time First, we compare the results on the PC-System, i.e., few cores with high base frequency. Table 2 compares the speedup of the construction algorithm on the *PC* hardware. In the sequential case, *psWM* is the fastest construction algorithm on all but five texts, there *pcWM* is faster. The difference in runtime between *psWM* and *pcWM* is less than five percent on average. Using *psWM* we are up to 3.1 times faster than *serialWT*, which is the previously fastest sequential construction algorithm. On average *psWM* is 1.59

[6] `https://github.com/bingmann/malloc_count`, last accessed 14.02.2017.

times faster than *serialWT*. In the parallel case, *pcWM* is the fastest construction algorithm, *psWM* being on a close second place. Only on two texts *psWM* is faster. Those are texts with a very small alphabet ($\sigma = 5$ and $\sigma = 16$). Again, the difference between *psWM* and *pcWM* is around four percent. On average, *pcWM* is 2.12 times faster than *recWT* and at least 1.04 times faster. Compared with *recWM*, *psWM* is 1.99 times faster on average.

Second, we compare the results on the Server-System where we have 32 cores with a lower base frequency, see Table 3. In the sequential case *psWM* is the fastest construction algorithm with *pcWM* being the second fastest. On average *psWM* is three percent faster than *pcWM* and at most 2.6 times (1.47 times on average) faster than *serialWM* (the previously fastest WT-construction algorithm). There is a different situation in the parallel case, where the speed of *psWM* comes only close to the speed of *recWT* (the currently fastest WT-construction algorithm). Here, *psWM* is 36 % slower on average, as *recWT* scales very good.

Results – Memory Consumption The disadvantage of *psWM* when it comes to scaling can be redeemed when it comes to memory consumption. All algorithms show a similar footprint on both systems, see Tables 6 and 7 in the Appendix. The lowest memory consumption is archived by *pcWM*, which is matching our theoretical assumptions. Next, *psWM* requires 35 % more memory than *pcWM* but still 27 % less than *recWT* when both are executed in parallel. In the sequential case, *pcWM* and *psWM* require 50 % and 25 % less space than *serialWT*. The memory consumption of *levelWT* is enormous, requiring around 77 % more memory than *pcWM* in both cases (sequential and parallel).

5 From the Wavelet Tree to the Wavelet Matrix

The structure of a WT and a WM are very similar. If we compare the bit vectors of the WT and the WM at level ℓ we see two similarities. First, both bit vectors contain the ℓ-th MSB of each character of T and second, the bits are grouped in intervals with respect to the bit prefix of size ℓ of the corresponding character. Thus the number and sizes of the intervals is the same. The difference is the position of the intervals within each level. At level ℓ, the intervals in BV'_ℓ of a WT 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, the second corresponds to characters with bit prefix 1, and so on. The intervals in BV_ℓ of a WM occur in increasing order with respect to the ρ_ℓ of the characters in T.

We can make use of these similarities by showing that each algorithm that can compute a WT can also compute a WM in the same asymptotic time. The computed data structures for our running example can be found in Figure 2.

▶ **Lemma 4.** *We can compute an array X and a bit vector U with rank and select data structures in time $\mathcal{O}(n+\sigma)$ and space $n + \sigma + \sigma\lceil \lg n\rceil + o(n+\sigma)$ bits, such that*

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

where $bp = \mathrm{prefix}(\ell, \mathrm{rank}_0(\mathsf{U}, \mathrm{select}_1(\mathsf{U}, i+1))$ and $\mathit{off} = i - \mathrm{rank}_1(\mathsf{U}, \mathrm{select}_0(\mathsf{U}, bp \ll (\lceil \lg\sigma\rceil - \ell)))$, with $\ll k$ denoting a left bit shift (by k bits), i.e., affixing k zeros on the right hand side.

Proof. First, we count the number of occurrences of each character in T. This requires $\mathcal{O}(n)$ time and $\sigma\lceil \lg n\rceil$ bits of space. We store them such that $\mathsf{X}[i] = |\{j \in [0,n) \colon \mathsf{T}[j] = i\}|$. Next, we store the number of occurrences unary utilizing a bit vector, i.e., setting the first $\mathsf{X}[0]$

	[This paper]				Shun [17]			Labeit et al. [9]	
	pcWM		psWM		serialWT	levelWT		recWT	
Text	T_1	T_{32}	T_1	T_{32}	T_1	T_1	T_{32}	T_1	T_{32}
XML	3.363	0.766	**3.121**	0.292	4.530	6.970	0.475	7.143	<u>0.231</u>
DNA	**2.000**	0.907	2.063	0.213	2.890	3.990	0.248	4.763	<u>0.154</u>
ENG	4.039	0.769	**3.950**	0.336	6.040	8.590	0.782	9.313	0.304
PROT	2.335	0.795	**2.172**	<u>0.229</u>	4.500	6.130	0.357	7.459	0.239
SRC	3.980	0.808	**3.794**	0.325	5.640	8.240	0.549	9.055	<u>0.288</u>
chr22.dna	0.239	0.186	**0.237**	<u>0.036</u>	0.375	0.505	0.049	0.605	0.059
etext99	2.012	0.454	**1.968**	0.200	3.140	4.440	0.280	4.915	<u>0.160</u>
gcc-3.0.tar	1.666	0.339	**1.606**	0.157	2.280	3.350	0.222	3.581	<u>0.116</u>
howto	0.761	0.181	**0.724**	0.078	1.080	1.560	0.101	1.687	<u>0.056</u>
jdk13c	1.124	0.272	**1.035**	0.116	1.540	2.370	0.158	2.531	<u>0.086</u>
linux-2.4.5.tar	2.231	0.448	**2.128**	0.198	3.080	4.520	0.304	4.785	<u>0.156</u>
rctail96	1.872	0.428	**1.742**	0.175	2.540	3.870	0.259	4.068	<u>0.135</u>
rfc	1.934	0.435	**1.847**	0.180	2.850	4.040	0.263	4.378	<u>0.142</u>
sprot34.dat	1.806	0.393	**1.718**	0.169	2.480	3.800	0.245	4.032	<u>0.131</u>
w3c2	**1.991**	0.412	2.028	0.225	2.590	4.010	0.271	4.092	<u>0.134</u>
random1	**1.792**	0.445	1.843	<u>0.179</u>	4.810	6.080	0.314	8.182	0.256
random2	**5.126**	0.780	14.35	0.861	9.590	12.20	0.645	16.14	<u>0.515</u>
words	**10.00**	1.553	26.90	3.564	15.20	22.40	1.480	24.54	<u>0.833</u>

Table 3 Running times of the algorithms on the Server-system in seconds. T_1 marks the running time using one core and T_{32} denotes the running time using 32 cores. The fastest sequential running time is is highlighted using bold font and the fastest parallel running time is underlined.

bits to one, followed by a single bit set to zero that is again followed by $\mathsf{X}[1]$ bits set to one, followed again by a single bit set to zero, and so on. We augment the bit vector with a rank and select data structure, resulting in $n + \sigma + o(n + \sigma)$ bits of space in total. The construction of the bit vector and rank and select data structure requires $\mathcal{O}(n + \sigma)$ time.

Next, for each level $\ell \in [2, \lceil \lg \sigma \rceil)$ we want to compute the first position of the intervals corresponding to the reverse bit prefixes of size ℓ in the WM. Since we only require bit prefixes of size up to $\lceil \lg \sigma \rceil - 1$ we first compute the number of occurrences of these bit prefixes, i.e., $\mathsf{X}[i] = \mathsf{X}[2i] + \mathsf{X}[2i + 1]$ for all $i \in [0, \lceil \sigma/2 \rceil)$. Next, we rearrange the number of occurrences (that are in lexicographically order) by swapping $\mathsf{X}[i]$ with $\mathsf{X}[\mathrm{reverse}(i)]$ for all $i \in [0, \lceil \sigma/2 \rceil)$. Now we have the number of occurrences in bit-reversal permutation order. This requires $\mathcal{O}(\sigma)$ time and no additional space. Furthermore, we now have $\lceil (\sigma \lg n)/2 \rceil$ bits of unused space. Next, we compute the prefix sum of the values starting with 0 in $\mathcal{O}(\sigma)$ time. Using these starting positions of the intervals, we can compute the starting positions of the intervals for each level $\ell \in [2, \lceil \lg \sigma \rceil)$ in $\mathcal{O}(\sigma)$ time using the $\lceil (\sigma/2) \lg n \rceil$ bits of unused space. Last, we restore the order for the starting positions, such that the starting positions for each level occur in increasing order with respect to their corresponding bit prefix, in time $\mathcal{O}(\sigma)$. Thus, the preprocessing requires $\mathcal{O}(n + \sigma)$ time and $n + \sigma + 2\sigma \lceil \lg n \rceil + o(n + \sigma)$ bits of space.

Now we need to answer queries asking for a position $j \in [0, n)$ in BV_ℓ given a position $i \in [0, n)$ in BV'_ℓ for $\ell \in [0, \lceil \lg \sigma \rceil)$ in constant time. If $\ell \leq 1$ we know that $j = i$, because the

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

$\mathsf{U} = \texttt{10110101010101101}$

$\mathsf{X} =$

0	1	0 0	0 1	1 0	1 1
0	5	0	5	3	7

Figure 2 Example of the data structures and querying for T= `0167154263`.

bit vectors of the WT and WM are the same for the first two levels. Otherwise ($\ell > 1$) we use the bit vector to identify the bit prefix of length ℓ of the character responsible for setting the bit at position i. Let $pos = \text{select}_1(i+1)$ be the position of the $i+1$-th one in the bit vector. Therefore, $k = \text{rank}_0(pos)$ returns the rank of the character that corresponds to the position pos. The bit prefix $bp = \text{prefix}(\ell, k)$ of length ℓ can now be used to determine the starting position of the corresponding interval in BV_ℓ, i.e., $\mathsf{X}[2^\ell - 2 + bp]$ because we have reordered the entries level-wise. Now we need to compute the offset of the position from the starting position of the interval. To do so, we compute the smallest character contained in the interval by padding the bit prefix with $\lceil \lg \sigma \rceil - \ell$ 0s giving us a value $r = \text{select}_0(bp \ll \lceil \lg \sigma \rceil - \ell)$. Next, we determine the number of 1s occurring before the r-th 0 in U to compute the offset, i.e., $\text{off} = i - \text{rank}_1(r)$.

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

▶ **Example 5.** Given our running example of T= `0167154263`, we compute the bit vector U and the array X as shown in Figure 2. The first two levels of the WT and WM are the same, hence we give an example for the last level. We want to set the $i = 8$-th bit in BV'_2 to 0. Now we need to compute the corresponding position j in BV_2. To do so, we first identify the position of the $(i+1) = (8+1)$-th 1 in U, i.e., $p = \text{select}_1(9) = 15$. The value represented by this position may not correspond to the value of the considered character, but it has the same bit prefix of length 2 as the character. The bit prefix is $bp = \text{prefix}(2, \text{rank}_0(15)) = (\texttt{11})_2$. To get the first position in the interval in level 2, we need to pad the bit prefix with $\lceil \lg \sigma \rceil - \ell = 1$ zeros to get the smallest value with the bit prefix bp, i.e., $(\texttt{110})_2 = 6$. Now we can compute the offset of the position with respect to the first position of the interval. We identify the starting position of the interval containing 6, i.e., $\text{select}_0(6) = 7$. Then we get the number of 1s up to that position and subtract this value from i ($\mathit{off} = 8 - 7 = 1$) to get the offset. Using the bit prefix bp and the offset off, we can get the position where we have to set the bit using $\mathsf{X}[2^2 - 2 + bp] + \mathit{off} = 7 + 1 = 8$.

6 Conclusion

We presented two sequential and parallel WM-construction algorithms that utilized the structure of the WM to compute it bottom-up. This allows for fast sequential and parallel WM-construction algorithms that require just a little bit more memory than the input and output require. We then showed how to adopt these algorithms to compute the WT instead. Our experiments have shown that the our new algorithms are up to twice as fast as the previously known algorithms while requiring just a fraction of the memory (at most half as much). Our algorithms do not scale as well as the competitors, therefore, when using more than 32 processors our algorithms will be outperformed.

In addition to the practical work, we also have shown how to adopt general WT-construction algorithms to compute a WM in the same asymptotic runtime instead.

The presented algorithms are the first two WM-construction algorithms that are not just adopted WT-construction algorithms. We want to investigate further in this direction to get construction algorithms that scale better. The *domain-decomposition* approach by Fuentes-Sepulveda et al. [16] may also be applicable to WM construction.

Acknowledgments

We would like to thank Benedikt Oesing for implementing early prototypes of different WM-construction algorithms in his Bachelor's thesis [15] indicating promising approaches. Further thanks go to Nodari Sitchinava (U. Hawaii) for interesting discussions on the work-time paradigm.

References

1 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana A. Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 572–591. SIAM, 2015.

2 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015.

3 Francisco Claude, Patrick K. Nicholson, and Diego Seco. Space efficient wavelet tree construction. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7024 of *LNCS*, pages 185–196. Springer, 2011.

4 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009.

5 José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient wavelet tree construction and querying for multicore architectures. In *International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 150–161. Springer, 2014.

6 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. SIAM, 2003.

8 Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221. IEEE Computer Society, 2011.

9 Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. In *Data Compression Conference (DCC)*, pages 33–42. IEEE, 2016.

10 Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *Latin American Theoretical Informatics Symposium (LATIN)*, volume 3887 of *LNCS*, pages 703–714. Springer, 2006.

11 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007.

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

13 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016.

14 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

15 Benedikt Oesing. Effiziente Erstellung von Waveletmatrizen (B. S. Thesis in German), 2016. URL: `https://ls11-www.cs.tu-dortmund.de/fischer/abschlussarbeiten/wavelet`.

16 José Fuentes Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Parallel construction of wavelet trees on multicore architectures, 2016. URL: `http://arxiv.org/abs/1610.05994`.

17 Julian Shun. Parallel wavelet tree construction. In *Data Compression Conference (DCC)*, pages 63–72. IEEE, 2015.

18 Julian Shun. Improved parallel construction of wavelet trees and rank/select structures, 2016. (To appear in IEEE Data Compression Conference (DCC), 2017). URL: `http://arxiv.org/abs/1610.03524`.

19 German Tischler. On wavelet tree construction. In Raffaele Giancarlo and Giovanni Manzini, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 6661 of *LNCS*, pages 208–218. Springer, 2011.

A Additional Data from the Experiments

	Gog et al. [6]			
	WT construction		WM construction	
Text	in-memory	semi-extern	in-memory	semi-extern
XML	20.69	15.93	18.20	15.43
DNA	16.22	12.38	14.56	12.20
ENG	20.26	16.41	18.65	16.56
PROT	19.91	14.94	16.91	14.95
SRC	22.58	16.35	18.25	17.01
chr22.dna	2.427	2.028	2.435	2.003
etext99	10.31	8.160	9.326	7.857
gcc-3.0.tar	9.718	6.378	7.263	6.123
howto	3.018	2.933	3.519	2.820
jdk13c	6.472	5.728	6.438	5.296
linux-2.4.5.tar	13.29	8.913	9.838	8.562
rctail96	10.85	9.013	10.74	9.009
rfc	10.88	8.836	9.820	8.440
sprot34.dat	8.383	7.332	8.269	7.495
w3c2	10.21	8.166	9.704	8.576
random1	13.21	9.970	11.76	9.894
random2	36.85	20.05	63.50	20.89
words	72.55	36.93	155.1	37.07

Table 4 Running time of the WM- and WT-construction algorithms implemented in the SDSL on the PC-System in seconds. We used the *wt_int* and *wm_int* implementation (for the WT and WM, resp.) and constructed the data structures using *construct_im* for the in-memory construction and *construct* for the semi-external construction.

	[This Paper]							
	PC-System				Server-System			
	pcWT		psWT		pcWT		psWT	
Text	T_1	T_4	T_1	T_4	T_1	T_{32}	T_1	T_{32}
XML	2.503	0.749	2.162	0.759	3.462	0.776	3.116	0.294
DNA	1.482	0.565	1.432	0.435	2.052	0.946	2.059	0.211
ENG	3.029	0.873	2.761	0.886	4.143	0.782	3.981	0.327
PROT	1.729	0.541	1.509	0.530	2.390	0.785	2.176	0.230
SRC	2.973	0.857	2.620	0.909	4.055	0.779	3.787	0.330
chr22.dna	0.172	0.110	0.163	0.063	0.237	0.202	0.234	0.033
etext99	1.501	0.436	1.360	0.458	2.090	0.464	1.947	0.182
gcc-3.0.tar	1.249	0.358	1.103	0.399	1.694	0.369	1.572	0.157
howto	0.564	0.166	0.503	0.169	0.791	0.188	0.723	0.078
jdk13c	0.839	0.245	0.716	0.252	1.149	0.268	1.031	0.173
linux-2.4.5.tar	1.663	0.479	1.466	0.496	2.277	0.468	2.133	0.199
rctail96	1.406	0.419	1.213	0.418	1.932	0.427	1.782	0.174
rfc	1.441	0.424	1.287	0.435	2.000	0.449	1.852	0.179
sprot34.dat	1.346	0.382	1.194	0.407	1.864	0.420	1.729	0.168
w3c2	1.470	0.423	1.364	0.596	2.023	0.431	2.029	0.230
random1	1.347	0.401	1.107	0.413	1.854	0.439	1.87	0.297
random2	3.523	1.106	6.087	1.744	5.100	0.794	14.06	0.876
words	7.340	3.376	10.47	4.110	10.60	1.468	26.73	2.939

Table 5 Running time of our WT-construction algorithms in seconds.

	[This paper]				Shun [17]			Labeit et al. [9]	
	pcWM		psWM		serialWT	levelWT		recWT	
Text	T_1	T_4	T_1	T_4	T_1	T_1	T_4	T_1	T_4
XML	393.226.891	393.234.627	602.942.091	602.947.747	812.653.759	1.652.334.559	1.655.357.440	814.899.200	816.029.696
DNA	340.796.723	340.801.035	550.511.923	550.515.851	734.004.479	1.573.685.279	1.576.132.608	736.174.080	737.574.912
ENG	419.443.095	419.455.199	629.158.295	629.166.255	838.877.963	1.678.558.763	1.681.948.672	841.097.216	842.346.496
PROT	340.796.723	340.801.035	550.511.923	550.515.851	760.219.563	1.599.900.363	1.602.461.696	762.372.096	763.441.152
SRC	419.443.095	419.455.199	629.158.295	629.166.255	838.878.343	1.678.559.143	1.681.235.968	841.097.216	842.440.704
chr22.dna	47.520.539	47.523.395	82.074.267	82.077.763	116.619.275	254.969.295	256.491.520	118.177.792	118.284.288
etext99	210.567.319	210.579.423	315.844.631	315.852.591	421.120.407	842.641.299	845.193.216	422.952.960	423.899.136
gcc-3.0.tar	173.273.495	173.285.599	259.903.895	259.911.855	346.533.063	693.393.395	695.476.224	348.147.712	348.782.592
howto	78.856.855	78.868.959	118.278.935	118.286.895	157.703.355	315.545.823	317.390.848	159.232.000	159.866.880
jdk13c	130.752.571	130.760.307	200.481.467	200.487.123	270.208.123	549.396.355	551.194.624	271.781.888	272.568.320
linux-2.4.5.tar	232.522.135	232.534.239	348.776.855	348.784.815	465.038.399	930.511.843	932.839.424	466.911.232	467.804.160
rctail96	215.094.091	215.101.827	329.805.131	329.810.787	444.512.411	903.805.103	906.137.600	446.251.008	447.283.200
rfc	218.301.931	218.309.667	334.723.819	334.729.475	451.143.999	917.286.775	919.613.440	452.800.512	453.730.304
sprot34.dat	205.543.051	191.848.643	315.160.203	315.165.859	424.771.467	863.668.687	866.095.104	426.401.792	427.741.184
w3c2	208.414.615	208.426.719	312.615.575	312.623.535	416.823.359	834.034.635	836.349.952	418.430.976	419.704.832
random1	200.012.695	200.024.799	300.012.695	300.020.655	400.019.519	801.599.488	802.152.448	401.756.160	402.460.672
random2	401.057.526	404.868.286	601.057.526	602.240.526	800.019.519	1.602.027.520	1.603.272.704	802.070.528	803.254.272
words	1.015.263.092	1.359.208.860	1.582.097.880	1.716.319.024	2.267.358.675	4.538.253.312	4.541.308.928	2.270.994.432	2.273.808.384

Table 6 Memory usage (on the PC-System) of the algorithms in byte running sequential (T_1) and running on four cores (T_4).

	[This paper]				Shun [17]			Labeit et al. [9]	
	pcWM		psWM		serialWT	levelWT		recWT	
Text	T_1	T_{32}	T_1	T_{32}	T_1	T_1	T_{32}	T_1	T_{32}
XML	393.227.899	393.235.635	602.943.099	602.992.211	812.653.735	1.655.341.056	1.665.097.728	816.549.888	827.760.640
DNA	340.797.731	340.802.043	550.512.931	550.538.811	734.004.455	1.576.620.032	1.585.360.896	737.673.216	748.138.496
ENG	419.444.103	419.456.207	629.159.303	629.239.391	838.877.939	1.681.567.744	1.696.366.592	842.625.024	857.186.304
PROT	340.797.731	340.802.043	550.512.931	550.538.811	760.219.539	1.603.104.768	1.613.336.576	763.891.712	775.593.984
SRC	419.444.103	419.456.207	629.159.303	629.239.391	838.878.319	1.681.625.088	1.693.802.496	842.608.640	856.887.296
chr22.dna	47.521.547	47.524.403	82.075.275	82.095.347	116.619.251	257.740.800	263.512.064	119.537.664	125.804.544
etext99	210.568.327	210.580.431	315.845.639	315.925.727	421.120.383	845.721.600	856.596.480	424.329.216	434.515.968
gcc-3.0.tar	173.274.503	173.286.607	259.904.903	259.984.991	346.533.039	696.246.272	706.969.600	349.941.760	360.087.552
howto	78.857.863	78.869.967	118.279.943	118.360.031	157.703.331	318.619.648	324.866.048	160.735.232	168.452.096
jdk13c	130.753.579	130.761.315	200.482.475	200.531.587	270.208.099	552.583.168	561.020.928	273.592.320	280.100.864
linux-2.4.5.tar	232.523.143	232.535.247	348.777.863	348.857.951	465.038.375	933.470.208	942.235.648	468.389.888	476.266.496
rctail96	215.095.099	215.102.835	329.806.139	329.855.251	444.512.387	906.911.744	913.199.104	447.811.584	456.605.696
rfc	218.302.939	218.310.675	334.724.827	334.773.939	451.143.975	920.252.416	928.907.264	454.397.952	462.155.776
sprot34.dat	205.544.059	205.551.795	315.161.211	315.210.323	424.771.443	866.779.136	874.967.040	428.204.032	435.429.376
w3c2	208.415.623	208.427.663	312.616.583	312.696.671	416.823.335	837.099.520	848.023.552	420.302.848	429.563.904
random1	200.013.703	200.025.807	300.013.703	300.093.791	400.019.519	801.599.488	802.152.448	403.595.264	412.999.680
random2	401.058.534	404.869.294	601.058.534	616.936.382	800.019.519	1.602.027.520	1.603.272.704	803.725.312	816.619.520
words	1.015.264.064	1.359.209.832	1.582.098.816	2.882.351.704	2.267.358.503	4.539.990.016	4.563.759.104	2.272.497.664	2.295.189.504

Table 7 Memory usage (on the Server-System) of the algorithms in byte running sequential (T_1) and running on 32 cores (T_{32}).