Parallel External Memory Wavelet Tree and Wavelet Matrix Construction*

Jonas Ellert and Florian Kurpicz

Technische Universität Dortmund, Department of Computer Science, Germany
{jonas.ellert,florian.kurpicz}@tu-dortmund.de

Abstract. We present the first parallel external memory wavelet tree and matrix construction algorithm. The algorithm's throughput is nearly the same as the hard disk drives' throughput, using six cores. We also present the fastest (parallel) semi-external construction algorithms for both wavelet trees and matrices.

Keywords: External Memory · Parallel Algorithm · Wavelet Tree.

1 Introduction

The *wavelet tree* [9] is a compact data structure that can answer access, rank, and select queries for a text over an alphabet $[0, \sigma)$ in $\mathcal{O}(\lg \sigma)$ time, while requiring only $n\lceil \lg \sigma \rceil (1 + o(1))$ bits of space. The *wavelet matrix* [3] is an alternative representation with the same space and time bounds for construction and answering queries. Both are used in many applications, e. g., text indexing [9], compression [15,10], and as an alternative to fractional cascading [13].

Our Contributions. First, we develop semi-external memory wavelet tree construction algorithms, where semi-external means we keep data that requires random access in main memory and all other data in external memory. Our implementations outperform the only previously available implementations by a factor of up to 1.43 regarding their running time, while using up to 16.6 times less memory. We then describe parallel fully external wavelet tree construction algorithms, which almost achieve a throughput that is only 1.19 times slower than the maximum throughput achievable on the hard disk drive using six threads. In general, we can achieve a speedup of up to 3.51 using six threads, and are mostly limited by I/Os. Finally, all algorithms are able to compute the wavelet matrix with the same space and time requirements, making them the fastest and first (semi-)external wavelet matrix construction algorithms.

Related Work. To our best knowledge, there exist no other external memory wavelet tree construction algorithms. The succinct data structure library [8] contains algorithms that can construct wavelet trees in semi-external memory.

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

Still, engineering of efficient wavelet tree and matrix construction algorithms in other models of computations is an ongoing problem with very recent advances. First, Fischer et al. [5] introduced bottom-up wavelet construction algorithms that are very fast and memory efficient in practice, and result in the fastest sequential and shared memory parallel wavelet tree and matrix construction algorithms. Also, Kaneta [11] recently presented a practical implementation of the $\mathcal{O}\big(n\lceil\lg\sigma/\sqrt{\lg n}\rceil\big)$-construction time algorithm, which uses word packing techniques in word-RAM, and has been (independently) introduced by Babenko et al. [2] and Munro et al. [16].

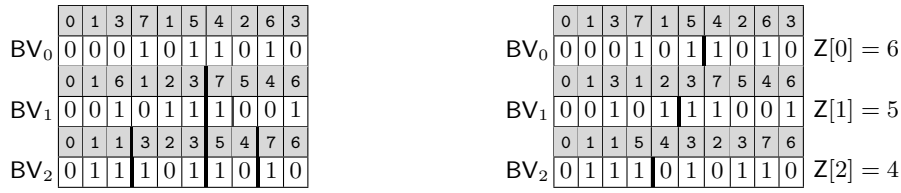
2 Preliminaries

(In this paper, we use the notation introduced by Fischer et al. [5].) 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 $\text{bits}(\alpha)$. 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 $0 \leq k < \lceil\lg\sigma\rceil$. The *bit prefix* of size k of $\alpha \in \Sigma$ are the k MSBs, 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 $\text{rank}_0(\mathsf{BV}, i)$ returns the number of 0's in $\mathsf{BV}[0, i)$, whereas $\text{select}_0(\mathsf{BV}, i)$ returns the position of the i-th 0 in BV. We define $\text{rank}_1(\mathsf{BV}, i)$ and $\text{select}_1(\mathsf{BV}, i)$ analogously. Both, rank and select queries on a bit vector of size n can be answered in $\mathcal{O}(1)$ time using succinct dictionary data structures that require only $o(n)$ bits space [17].

2.1 The Wavelet Tree

Let T be a text of length n over an alphabet $[0, \sigma)$. The *wavelet tree* (WT) [9] of T is a complete and balanced binary tree. Each node of the WT represents characters in $[\ell, r) \subseteq [0, \sigma)$. The root of the WT 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 $l + 2 \geq r$. The characters in $[\ell, r)$ at the corresponding node v are represented using a bit vector BV_v such that the i-th bit in BV_v is $\text{bit}\big(d(v), T_{[\ell,r)}[i]\big)$, where $d(v)$ is the depth of v in the WT, 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 $[\ell, r)$.

Wavelet trees can be used to generalize access, rank, and select queries from bit vectors to alphabets of size σ. Answering these queries then requires $\mathcal{O}(\lg\sigma)$ time. To do so, the bit vectors of the WT are augmented by binary rank and select data structures. For further information on queries, we point to [17].



(a) Level-wise wavelet tree. (b) Wavelet matrix.

Fig. 1: The level-wise (a) WT and the WM (b) of $T = [0, 1, 3, 7, 1, 5, 4, 2, 6, 3]$. The light gray ( ) arrays contain the characters represented at the corresponding position in the bit vector and are not a part of the WT and WM. In (a), thick lines represent the borders of the intervals, which are not stored explicitly. In (b), thick lines represent the number of zeros, which are stored in the Z-array.

Level-Wise Wavelet Tree. In this paper, we consider *level-wise* wavelet trees. Here, we concatenate the bit vectors of all nodes at the same depth. Since we lose the tree topology, the resulting bit vectors correspond to a *level* that is equal to the depth of the concatenated nodes and the concatenated bit vectors correspond to *intervals* in the level. We store only a single bit vector BV_ℓ for each level $\ell \in [0, \lceil \lg \sigma \rceil)$, see Figure 1a. This retains the functionality, but reduces the redundancy for the succinct dictionaries needed to answer rank and select queries on the bit vectors in constant time [13,14]. We can also easily identify the interval in which a character is represented at any level:

Observation 1 (Fuentes-Sepúlveda et al. [6]) *Given a character $T[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the WT, the interval pertinent to $T[i]$ in BV_ℓ can be computed by* bit_prefix$(\ell, T[i])$.

Wavelet Matrix. A variant of the wavelet tree, the *wavelet matrix* (WM), was introduced in 2011 by Claude [3]. It requires the same space as a WT and has the same asymptotic running times for access, rank, and select; but in practice it is often faster than a WT for rank and select queries [3], as it needs less calls to binary rank/select data structures. For the definition of the WM, we need additional notations: *Reversing* the significance of the bits is denoted by reverse, e. g., reverse$((\mathtt{001})_2) = (\mathtt{100})_2$. 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, $\rho_2 = (0, 2, 1, 3) = ((\mathtt{00})_2, (\mathtt{10})_2, (\mathtt{01})_2, (\mathtt{11})_2)$. ρ_k and ρ_{k+1} can be computed from another, as $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k - 1), 2\rho_k(0) + 1, \ldots, 2\rho_k(2^k - 1) + 1)$.

In a WM the tree structure is discarded completely and we use the array $\mathsf{Z}[0, \lceil \lg \sigma \rceil)$ to store the number of zeros at each level ℓ in $\mathsf{Z}[\ell]$. BV_0 contains the MSBs of each character in T in text order (this is the same as the first level of a WT). For $\ell \geq 1$, BV_ℓ is defined as follows. Assume that a character α is represented at position i in $\mathsf{BV}_{\ell-1}$. Then the position of its ℓ-th MSB in BV_ℓ depends on $\mathsf{BV}_{\ell-1}[i]$ in the following way: if $\mathsf{BV}_{\ell-1}[i] = 0$, bit$(\ell, \alpha)$

is stored at position $\text{rank}_0(\mathsf{BV}_{\ell-1}, i)$; otherwise ($\mathsf{BV}_{\ell-1}[i] = 1$), it is stored at position $\mathsf{Z}[\ell-1] + \text{rank}_1(\mathsf{BV}_{\ell-1}, i)$. For an example, see Figure 1b. Similar to the intervals in the bit vectors 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 have to consider the reversed bit prefixes:

Observation 2 *Given a character $T[i]$ for $i \in [0, n)$ and a level $\ell \in [1, \lceil \lg \sigma \rceil)$ of the WM, reverse*($\text{bit\_prefix}(\ell, T[i])$) *indicates the interval pertinent to $T[i]$ in BV_ℓ. Namely, $\mathsf{BV}_\ell[i] = \text{bit}[\ell, S[i]]$, where S is T stably sorted using the reversed bit prefixes of length ℓ of the characters as key.*

2.2 The External Memory Model

The *external memory model* [1] measures the transfer of data between the main memory of size M (also called *internal memory*) and a secondary memory (also called *external memory*) that is assumed to be of unlimited size and slower in terms of memory access than the main memory. Also, data can only be transferred in *blocks* of size B between main and secondary memory. Transfers of blocks are called *I/O operations* (*I/Os* for short) and are the main cost measure of the external memory model.

For *semi-external* algorithms, we assume that we have random access on either the input or output—but not both. This relaxation allows for algorithms that cannot be efficiently be expressed in the external memory model. The model is used in practice, e. g., the succinct data structure library (SDSL) [8] provides semi-external WT construction algorithms (among others).

Computing the Effective Alphabet. We construct WTs and WMs using an *effective* alphabets, i. e., every character of the effective alphabet occurs in the text. Therefore, we can store $\lfloor w / \lg \sigma \rfloor$ characters in one w-bit computer word. To obtain the effective alphabet, we have to scan the text twice. First, we compute the histogram of all characters of the text, second we compute a transformation from the alphabet to the effective alphabet, and finally we scan the original text but store the text in the effective alphabet. We denote the number of blocks that must be transferred to scan the text by $\text{scan}(n\lceil \lg \sigma \rceil) = \lceil \lceil n \lceil \lg \sigma \rceil / w \rceil / B \rceil$.

In main memory, most implementations assume that the text is available in main memory over the effective alphabet. Our (semi-)external WT and WM construction algorithms mimic the behavior by assuming that the text is available in secondary memory over the effective alphabet. If that is not the case, all our algorithms require additional $2\lceil n/B \rceil + \text{scan}(n\lceil \lg \sigma \rceil)$ I/Os and $\mathcal{O}(n)$ time to compute the histogram and store the text over an effective alphabet.

3 Construction in Semi-External Memory

In this section, we briefly discuss how to adapt the fast WT and WM construction algorithms presented by Fischer et al. [5] to the semi-external memory model.

To this end, we first discuss the *bottom-up* construction, the new approach to compute the WT and WM [5]. Here, we construct the histogram of the text, and then use that histogram to compute histograms for all other bit prefixes without another text access. Generally speaking, if we have the histogram of length-ℓ bit prefixes, we can simply compute the histogram of the bit prefixes of length $\ell - 1$ by ignoring the last bit of the current prefix, e. g., the number of characters with bit prefix $(0\,1)_2$ is the total number of characters with bit prefixes $(0\,1\,0)_2$ and $(0\,1\,1)_2$, since both share the bit prefix $(0\,1)_2$. Using these histograms, we can compute the interval starting positions for all levels of the WT and WM.

Now, we have a look at the *space requirements* of this technique. The histogram of all characters requires $\sigma\lceil\lg n\rceil$ bits of space. We can always reuse that space for any histograms at a previous levels ℓ, which require $\lceil\sigma \lg n\rceil/2^{\lceil\lg\sigma\rceil-\ell}$ bits of space. Storing the borders requires the same space as storing the histogram. Note that we do not require a histogram for the first level, and we also do not require the starting positions resulting from the histogram of all characters. Since we require at most $\sigma\lceil\lg n\rceil/2$ 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 require $\sigma\lceil\lg n\rceil$ bits of space for both the histogram and the starting positions in total. If we require access to all histograms and cannot reuse the space, we need $2\lceil\sigma\lg n\rceil$ bits of space.

Random Access on the Output. Our first semi-external WT construction algorithm is the semi-external variant of the (single scan) prefix counting WT construction algorithm [5]. Here, we first compute the histogram for all characters in T and compute all histograms and interval borders without another scan of the text in $\mathcal{O}(n)$ time, $\mathrm{scan}(n\lceil\lg\sigma\rceil)$ I/Os, and $\sigma\lceil\lg n\rceil$ bits space, as described above. Next, we scan the text once again and fill all the bit vectors accordingly using the precomputed borders, i. e., for each symbol, we look at the border for each of the symbol's bit prefixes and set the corresponding bit in each bit vector accordingly (one bit per level) and then we update the borders. This requires $\mathcal{O}(n\lg\sigma)$ time in total for all levels. Setting the bits in the bit vectors still requires random access. 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 WT 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 $\lceil\lg\sigma\rceil$ threads. We denote this algorithm by *se.par.pc*.

Random Access on the Input. Next, we consider a modified and semi-external version of the prefix sorting WT construction algorithm [5]. Here, each level of the WT is written in sequential order, which lets us efficiently stream the bit vectors of the WT. Again, we precompute all borders of the intervals. Then, for each level ℓ, we use counting sort with the length-ℓ bit prefixes as keys to sort the text, such that we can fill the bit vector from left to right. Counting sort requires $\mathcal{O}(n)$ time, given the borders array, hence the running time does

not differ from se.pc. Since we require a stable sort, we cannot sort the text in place [18] and thus need additional $n\lceil\lg\sigma\rceil$ bits of space. 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 by sorting, we use a new in-place algorithm that rearranges the text as required by the WT in $\mathcal{O}(n)$ time. We decompose the text into $\Theta(\sqrt{n})$ blocks of size $\Theta(\sqrt{n})$ and use two buffers of the same size. Then, we separate the text using one buffer for symbols corresponding to a one bit and the other for the other bits. Whenever a buffer is full, we can write it to a part of the text, because the part is already written to the buffers. In the end, we have to rearrange the blocks (and shift some of them). We denote this variant by *se.ps.ip*. It requires less space, but is one of the slowest algorithms (see §5).

Lemma 1. *The semi-external algorithms se.pc, se.ps, and se.ps.ip compute the* WT *of a text of length n over an alphabet of size σ in $\mathcal{O}(n\lg\sigma)$ time using $\mathcal{O}(\text{scan}(n\lceil\lg\sigma\rceil))$ I/Os, and $n\lceil\lg\sigma\rceil+\sigma\lceil\lg n\rceil$ (se.pc) and $2n\lceil\lg\sigma\rceil+\sigma\lceil\lg n\rceil$ (se.ps) bits of main memory including input and output, respectively.*

Adaption to the Wavelet Matrix. Our semi-external memory WT construction algorithms can easily be extended to compute the WM 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 [5]. Also, this change does not affect the running time or the memory requirement; it only affects the content of the border array and subsequently the resulting bit vectors.

4 Wavelet Tree Construction in External Memory

If we replace the sorting in se.ps with any external memory sorting algorithm we obtain an external memory version of se.ps. However, sorting in external memory is (in practice) expensive. Now, we present dedicated external memory WT and WM construction algorithms. For the sequential algorithm we first explain how to build the WM, and then show how to adapt the algorithm to produce the WT.

4.1 Sequential Construction in External Memory

Each level ℓ of the WM 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 stable sorting the text $T_{\ell-1}$ of the previous level by the $(\ell-1)$-th bit. This property of the WM has been originally described as *all zeros of the level go left, and all the ones go right* [3]. If we know T_ℓ, then we can easily build BV_ℓ by taking the ℓ-th bit of each symbol of T_ℓ in left-to-right order. Thus, we can construct the entire WM 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

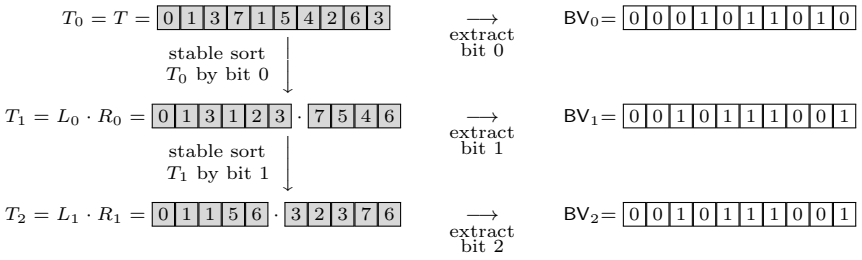


Fig. 2: Constructing the WM for our running example by partitioning the text and extraction of bits. The resulting WM can also be seen in Fig. 1b.

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 Fig. 2. The number $\mathsf{Z}[\ell]$ of zeros in each level is $|L_\ell|$.

Adaptation to the Wavelet Tree. Our external WM construction algorithm can easily be adapted to construct the WT instead. As described in §2, the bit vector belonging to any node of the WT always occurs in the WM, too. Only the order of these intervals is different. Our L_ℓ and R_ℓ buffers therefore already contain all the correct nodes, 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 node, 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 (see §2). We simply keep the histograms of all levels in main memory.

Analysis. We will first look at the I/O complexity of the WT/WM construction algorithm. The $\mathsf{Z}[\ell]$ values have size $\lceil \lg n \rceil \lceil \lg \sigma \rceil$ bits and are insignificant in terms of I/O complexity. Each reordered text T_ℓ has size $n\lceil \lg \sigma \rceil$ bits, which can be stored using $\mathrm{scan}(n\lceil \lg \sigma \rceil)$ blocks in external memory. We read each of these texts exactly once, and write all texts except for the initial text exactly once as well, resulting in $(2\lceil \lg \sigma \rceil - 1) \cdot \mathrm{scan}(n\lceil \lg \sigma \rceil)$ I/Os. The resulting WM or WT is written exactly once using another $\mathrm{scan}(n\lceil \lg \sigma \rceil)$ I/Os. All used structures in external memory are both written and read exclusively sequentially.

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

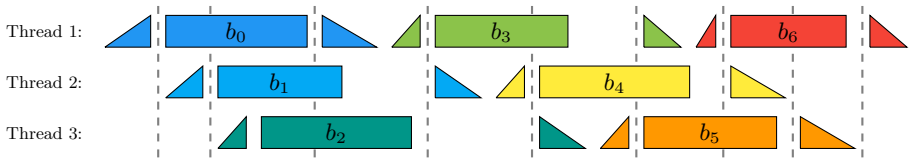


Fig. 3: Domain decomposition for a text $T = b_0|b_1|b_2|b_3|b_4|b_5|b_6$ split into seven segments. Here, ◺ means loading b_i from external memory, ▭ means computing the local tree for b_i, and ◺ means writing the local tree of b_i to external memory. Only one of the three threads is allowed to read/write at a time, as indicated by the dashed synchronization barriers. (Best viewed in color.)

time for the WM construction is $\mathcal{O}(n \lg \sigma)$. The WT construction needs additional $\mathcal{O}(\sigma)$ time to compute the histograms of all levels. In terms of space, the WM construction is fully external and only needs $\mathcal{O}(1)$ bits of main memory, since all data structures are kept in external memory. For the WT we need $2\lceil \sigma \lg n \rceil$ additional bits to store the histograms.

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

4.2 Parallel Construction in External Memory

For a more generic approach, we present a meta-algorithm based on the internal memory domain decomposition, see, e. g., [5,7,12]. Let p be the number of available threads, then in the internal memory setting we split the text into p segments, and compute the WT of each segment on a different thread, using a sequential construction algorithm of our choice. After that, the so called *local trees* can be merged into one *global tree*. In the external memory setting the length of the segments depends on the amount M of main memory. Assume that the sequential construction algorithm needs $s(n, \sigma)$ bits of memory for a text of length n over the alphabet $[0, \sigma)$. Then, the length k of each segment must satisfy $s(k, \sigma) \leq M/p$. This way all threads can work simultaneously.

Each thread runs a simple loop: load the next text segment from external memory into internal memory, compute the WT of the segment, and write it back to external memory. Only one thread is allowed to read/write at a time (see Fig. 3). In terms of external memory layout, we store the local trees in text order, and each local tree as the concatenation of its levels (see LT in Fig. 4). When merging the local trees into the global tree, we simply perform a single scan over the local trees and *zip* the corresponding intervals together. Since the length of each interval must be known in order to copy the right amount of bits, we need the histograms of all text parts during the merge phase. However, many of the fastest sequential WT construction algorithms either build the histograms or can

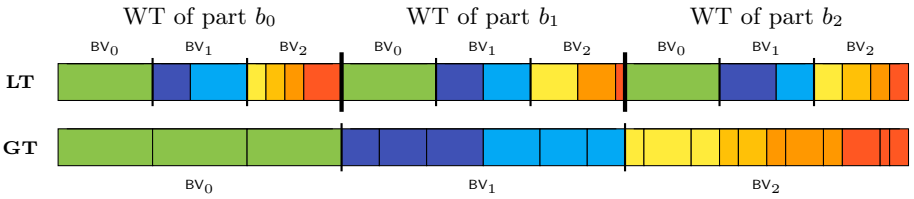


Fig. 4: External memory layout of local (LT) and global (GT) WTs for $T = b_0b_1b_2$. Best viewed in color, as colors indicate parts of local trees that are zipped together.

easily be modified to do so [5]. We are not using parallelism during the merge phase, since we are only copying bit vectors. In practice we are limited by the speed of the external memory, even when using only a single thread. Clearly, if we use a WM algorithm as a subroutine, our algorithm produces the WM instead.

Analysis. We will first look at the I/O complexity of our meta-algorithm. The input text, the concatenation of all local 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 local trees once, taking $2\,\text{scan}(n\lceil\lg\sigma\rceil)$ sequential I/Os. Reading all local 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 local tree. Therefore, 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 local tree. Thus, the total number of I/Os is bound by $4\,\text{scan}(n\lceil\lg\sigma\rceil) + \sigma\lceil n/k\rceil$. In practice we use the entire internal memory as a write buffer while merging the local 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 threads are computing local trees. The total I/O time (including the merge phase) is bound by $\mathcal{O}(n + \sigma\lceil n/k\rceil)$. The time during which all threads are computing local trees is bound by $\lceil n/pk\rceil \cdot t(k,\sigma)$. In terms of main memory we use $p \cdot s(k,\sigma)$ bits for up to p simultaneous executions of our internal memory construction algorithm over text segments of size k. Additional $\mathcal{O}(\lceil n/k\rceil\sigma\lg n)$ bits are needed to store all histograms.

Lemma 3. *Let $t(n,\sigma)$ and $s(n,\sigma)$ be the time and space used by an internal memory WT construction algorithm, and let $p, k \in \mathbb{N}^+$. The external memory algorithm ext.dd computes the WT of a text of length n over an alphabet of size σ using $4\,\text{scan}(n\lceil\lg\sigma\rceil) + \sigma\lceil n/k\rceil$ I/Os. If p threads are available, it takes $\mathcal{O}(n + \sigma\lceil n/k\rceil) + \lceil n/pk\rceil \cdot t(k,\sigma)$ time and $\mathcal{O}(\lceil n/k\rceil\sigma\lg n) + p \cdot s(k,\sigma)$ bits of internal memory including input and output.*

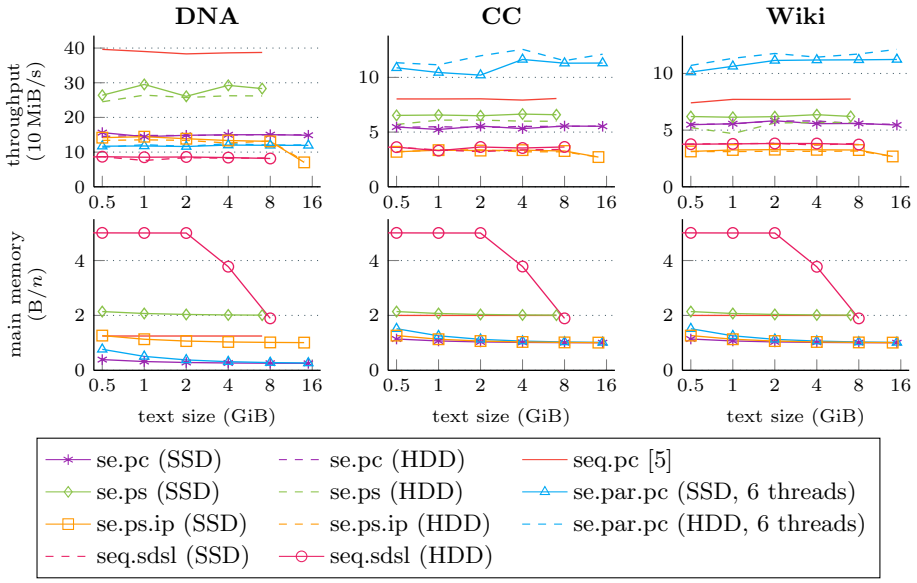


Fig. 5: Throughput and main memory peak of semi-external WT construction.

5 Experiments

For our experiments, we used a machine equipped with 16 GiB RAM, eight Hitachi HUA72302 HDDs each with a capacity of 1.8 TiB and two Samsung SSD 850 EVO SSDs each with a capacity of 465.8 GiB, and an Intel Xeon CPU i7-6800K (6 cores with frequency 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 [4] development snapshot (26-09-2017). We compiled all source code using `g++` 7.4 with flags `-O3` and `-march=native`, and express parallelism using OpenMP 4.5. We test our algorithms using both (a) four HDDs and (b) two SSDs. Before starting the timer, we compute the text over the effective alphabet and store it on disk, which is the input. Running times are the median of three executions. The implementations used for the evaluation are available from `https://github.com/kurpicz/pwm`.

We compare the following algorithms: **se.pc**, **se.par.pc**, **se.ps**, and **se.ps.ip** are the semi-external memory WT and WM algorithms described in §3, **seq.sdsl** is the semi-external memory algorithm contained in the SDSL, and **seq.pc** the fastest *main memory* WT algorithm [5], which we use as baseline. Our external (and parallel) WT and WM algorithms are the only external construction algorithms. Hence, we cannot compare **ext.ps** and **ext.dd** (see §4) with other algorithms in the same model. The construction times for WTs and WMs are nearly identical in both models.

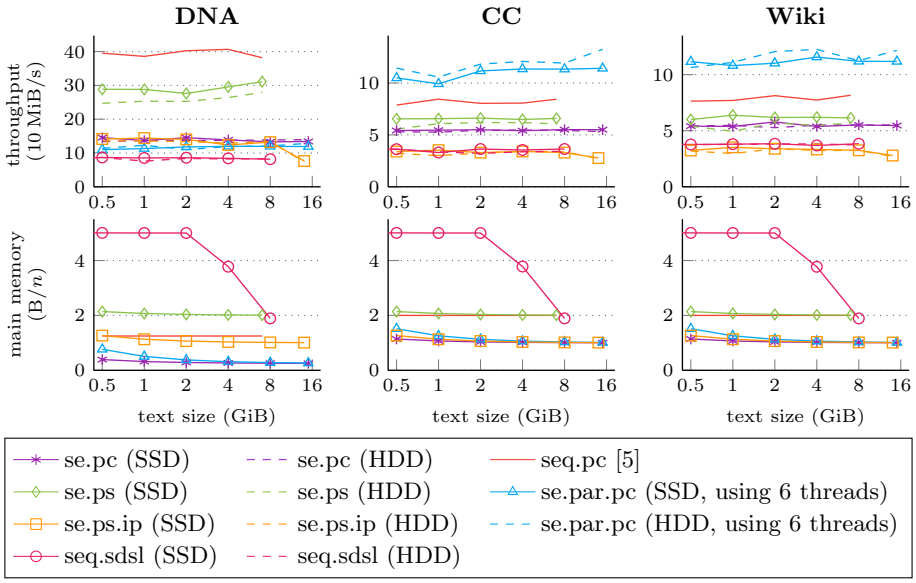


Fig. 6: Throughput and main memory peak of semi-external WM construction. Same experiment for the WM as reported in Fig. 5 for the WT.

We use the following real world inputs of sizes up to 128 GiB. When needing a smaller size, we consider a prefix of that size.

DNA ($\sigma = 4$) is a collection of DNA data from the 1000 Genomes Project (`http://internationalgenome.org/data`),

CC ($\sigma = 242$) contains websites (without HTML tags) that have been crawled by the Common Crawl corpus (`http://commoncrawl.org`), and

Wiki ($\sigma = 213$) are recent Wikipedia dumps containing XML files that are available from (`https://dumps.wikimedia.org`).

Semi-External Memory Construction Algorithms. An overview of the throughput and the required main memory of our semi-external WT construction algorithms can be found in Fig. 5. The results of the semi-external WM construction algorithms can be found in Fig. 6. Not plotted data means that the algorithm could not process the input size with the given main memory. The main memory algorithm seq.pc is used as base line and—as expected—always fastest sequential algorithm. The fastest semi-external memory algorithm on all inputs is se.ps. However, se.ps requires the most main memory—even more than seq.pc. The second fastest algorithm is se.pc. In addition, it is also the most memory efficient one, requiring less than all other tested algorithms. On DNA, se.ps.ip achieves a similar throughput to se.pc and is faster than seq.sdsl. On all other inputs se.ps.ip and seq.sdsl are always the slowest. Still, on all instances except for

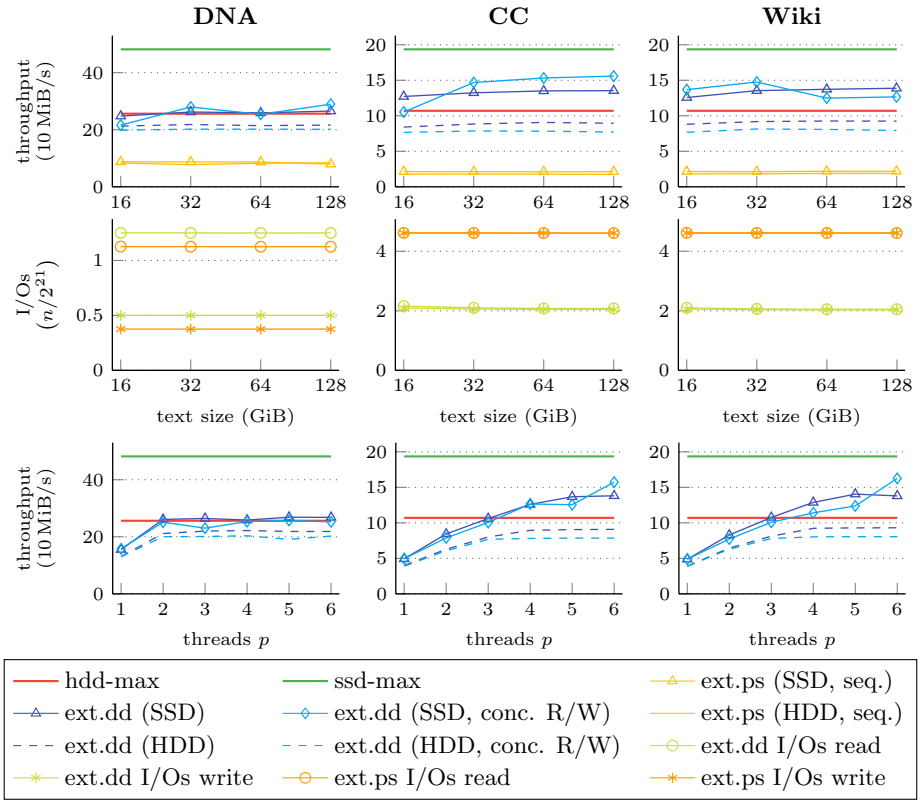


Fig. 7: Throughput (first row) and I/Os (second row) of external WT algorithms. Here, parallel algorithms uses all six threads. Throughput of ext.dd using 20 GiB input per thread (last row).

DNA, seq.sdsl requires five times more memory than se.pc. On DNA it even requires 16.6 times as much. The memory requirements of our semi-external algorithms per byte input is decreasing with larger inputs, as we use fixed-size buffers for our algorithms. When given inputs of size 4 GiB or more, seq.sdsl has to move the system swap, which explains the decrease in required memory. Therefore, se.pc is the fastest and most memory efficient semi-external memory algorithm. The throughput on SSDs is slightly better than on HDDs, except for se.par.pc, which has higher throughput on HDDs, which we cannot explain. It is also roughly twice as fast as se.pc, using slightly more memory, except on DNA (as expected).

External Memory Construction Algorithms. In Fig. 7 we show the throughput (first row) and I/Os (second row) of our external memory algorithms computing the WT. We show the same experiments for the WM computation in Fig. 8.

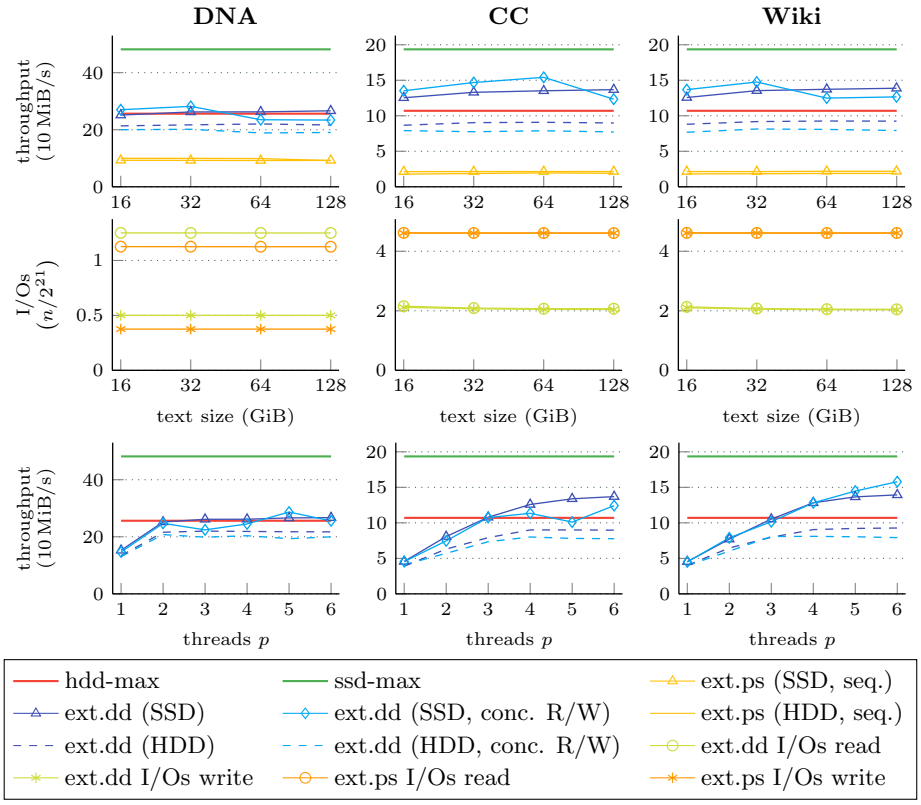


Fig. 8: Throughput (first row) and I/Os (second row) of external WT algorithms. Here, parallel algorithms uses all six threads. Throughput of ext.dd using 20 GiB input per thread (last row). Same experiment for the WM as reported in Fig. 7 for the WT.

We also give the maximum throughput (hdd-max and ssd-max) we achieved for reading the text and the WT once and writing the WT twice, which are exactly the external memory operations conducted by ext.dd. All algorithms have a nearly constant throughput, which is independent of the input size. The same is true for I/Os (both read and write). We also allow ext.dd to read and write concurrently (conc. R/W), which increases the throughput for SSDs on CC for inputs larger than 16 GiB, inputs of size up to 32 GiB on Wiki, and in general on DNA. For HDDs, it reduces the throughput on all text sizes by 4.55 % (DNA) to 11.12 % (Wiki). In the last row of Fig. 7 we show a weak scaling experiment of ext.dd. Using one thread, ext.dd is faster than ext.ps by a factor between 1.64 (DNA) to 2.14 (CC). Hence it is the fastest external memory WT construction algorithm. It also scales reasonably well, achieving a speedup of up to 3.51 (Wiki with conc. R/W). Here, concurrent read and write only increases throughput

Table 1: Characteristics of squential algorithms proposed in this paper.

Name	Time	I/Os	Memory in Bits
se.pc	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(\text{scan}(n\lceil \lg \sigma \rceil))$	$n\lceil \lg \sigma \rceil + \sigma \lceil \lg n \rceil$
se.ps	$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(\text{scan}(n\lceil \lg \sigma \rceil))$	$2n\lceil \lg \sigma \rceil + \sigma \lceil \lg n \rceil$
ext.ps (computing WT)	$\mathcal{O}(n \lg \sigma + \sigma)$	$2\lceil \lg \sigma \rceil \cdot \text{scan}(n\lceil \lg \sigma \rceil)$	$2\sigma \lceil \lg n \rceil$
ext.ps (computing WM)	$\mathcal{O}(n \lg \sigma)$	$2\lceil \lg \sigma \rceil \cdot \text{scan}(n\lceil \lg \sigma \rceil)$	$\mathcal{O}(1)$

on Wiki. On DNA, ext.dd does only scale for up to two threads, which is as expected as the number of threads that can efficiently be used is limited by the size of the alphabet. Also, we see using six threads ext.dd's throughput on HDDs is between 17.3 MiB/s (CC) and 38.9 MiB/s (DNA) less than the maximum throughput. Using SSDs, its throughput is between 157.3 MiB/s (Wiki) and 268.0 MiB/s (DNA).

On Shared Memory Wavelet Tree Construction. We have not included the throughput of the currently fastest parallel shared *internal* memory wavelet tree construction algorithm *dd.pc* [5] in any of the plots, due to the huge difference in speed (compared with our semi-external and external memory construction algorithms). For completeness, we now list the throughput of this algorithm on the same hardware and running on six threads. Note that we could only run dd.pc for inputs up to size 4 GiB, as a result of the memory usage of the algorithm.

On DNA, the maximum throughput is 1209.32 MiB/s, on CC it is 431.70 MiB/s, and on Wiki it is 416.26 MiB/s. All these throughputs are more than the theoretical best result an external memory algorithm can achieve on this machine.

6 Conclusion

We presented the fastest semi-external memory WT and WM construction algorithm and the first parallel semi-external memory WT and WM construction algorithm based on the main memory algorithms by Fischer et al. [5]. Then, we showed the first external memory WT and WM construction algorithm. A summary of the characteristics of these sequential algorithms is given in Table 1. In addition, we also parallelized the external memory algorithm. On HDDs, the parallel version of our external memory WT and WM construction achieves nearly perfect throughput, compared to the throughput that we obtain when we read and write the same amount that is read and written during the algorithm. It remains an open problem if there is a parallel algorithm that can obtain the same relative throughput on SSDs.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)

2. Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.A.: Wavelet trees meet suffix trees. In: 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 572–591. SIAM (2015)
3. Claude, F., Navarro, G., Pereira, A.O.: The wavelet matrix: An efficient wavelet tree for large alphabets. Inf. Syst. **47**, 15–32 (2015)
4. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw., Pract. Exper. **38**(6), 589–637 (2008)
5. Fischer, J., Kurpicz, F., Löbel, M.: Simple, fast and lightweight parallel wavelet tree construction. In: 20th Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 9–20. SIAM (2018)
6. Fuentes-Sepúlveda, J., Elejalde, E., Ferres, L., Seco, D.: Efficient wavelet tree construction and querying for multicore architectures. In: 13th International Symposium on Experimental Algorithms (SEA). LNCS, vol. 8504, pp. 150–161. Springer (2014)
7. Fuentes-Sepúlveda, J., Elejalde, E., Ferres, L., Seco, D.: Parallel construction of wavelet trees on multicore architectures. Knowl. Inf. Syst. **51**(3), 1043–1066 (2017)
8. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms (SEA). LNCS, vol. 8504, pp. 326–337. Springer (2014)
9. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 841–850. SIAM (2003)
10. Grossi, R., Vitter, J.S., Xu, B.: Wavelet trees: From theory to practice. In: International Conference on Data Compression, Communications and Processing (CCP). pp. 210–221. IEEE (2011)
11. Kaneta, Y.: Fast wavelet tree construction in practice. In: 25th International Symposium on String Processing and Information Retrieval (SPIRE). LNCS, vol. 11147, pp. 218–232. Springer (2018)
12. Labeit, J., Shun, J., Blelloch, G.E.: Parallel lightweight wavelet tree, suffix array and fm-index construction. J. Discrete Algorithms **43**, 2–17 (2017)
13. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: 7th Latin American Theoretical Informatics Symposium (LATIN). LNCS, vol. 3887, pp. 703–714. Springer (2006)
14. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. **387**(3), 332–347 (2007)
15. Makris, C.: Wavelet trees: A survey. Comput. Sci. Inf. Syst. **9**(2), 585–625 (2012)
16. Munro, J.I., Nekrich, Y., Vitter, J.S.: Fast construction of wavelet trees. Theor. Comput. Sci. **638**, 91–97 (2016)
17. Navarro, G.: Compact Data Structures - A Practical Approach. Cambridge University Press (2016)
18. Sedgewick, R.: Algorithms in C - Parts 1–4: Fundamentals, Data Structures, Sorting, Searching (3. Ed.). Addison-Wesley-Longman (1998)