

Parallel Wavelet Tree Construction*

Julian Shun

Carnegie Mellon University

jshun@cs.cmu.edu

Wavelet trees have received significant attention due to their applications in compressed data structures. We present several work-efficient parallel algorithms for wavelet tree construction that have poly-logarithmic span, improving upon the linear span of the recent parallel algorithms by Fuentes-Sepulveda et al. We experimentally show that on 40 cores our algorithms outperform the existing parallel algorithms by 2–12x and achieve up to 27x speedup over the sequential algorithm on a variety of real-world and artificial inputs. Our algorithms show good scalability with both increasing input size and increasing alphabet size. We also discuss how to extend our algorithms to variants of the standard wavelet tree.

1 Introduction

The *wavelet tree* was first described by Grossi et al. [15], where it was used in compressed suffix arrays. It is a space-efficient data structure that can support access, rank and select queries on a sequence in time logarithmic in the alphabet size of the sequence. Since its initial use, wavelet trees have found many other applications, for example in compressed representations of sequences, permutations, grids, graphs, self-indexes based on the Burrows-Wheeler transform [4], images [22] among many others (see [24, 23] for surveys of applications). In addition to access, rank and select queries, Makinen and Navarro [21] observed that wavelet trees can also be used for two-dimensional range queries. While there have been many papers on wavelet trees and their applications, the construction of wavelet trees has not been widely studied. This is not surprising, since the standard sequential method for constructing a wavelet tree in $O(n \log \sigma)$ work is very straightforward. However, constructing the wavelet tree of large sequences (with large alphabets) can require a lot of time, and hence it is important to parallelize the construction. A step in this direction was taken recently by Fuentes-Sepulveda et al. [12] who describe parallel algorithms for constructing wavelet trees that require $O(n)$ span (number of parallel time steps).

*We thank Simon Gog and Matthias Petri for commenting on a draft of this paper and helping with the code in [13]. We also thank Matthias Petri for suggesting the optimization discussed in the Appendix. We thank Leo Ferres for kindly providing the code from [12].

In this paper, we describe parallel algorithms for wavelet tree construction that exhibit even more parallelism. In particular, the span of our algorithms is poly-logarithmic, in contrast to the algorithms of Fuentes-Sepulveda et al. which require $O(n)$ span. We first describe an algorithm that constructs the tree level by level, and requires $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ span. We then describe a second algorithm which requires $O(W_{\text{sort}} \log \sigma)$ work and $O(S_{\text{sort}} + \log n)$ span, where W_{sort} and S_{sort} are the work and span, respectively, of the parallel stable integer sorting routine used in the algorithm. Using a work-efficient integer sort [26, 19], the work bound is $O(n \log \sigma)$ and span bound is $O(\sigma^\epsilon + \log n)$ for some constant $0 < \epsilon < 1$. For alphabets of poly-logarithmic size, this gives us a work-efficient algorithm with $O(\log n)$ span. Using a non-work-efficient integer sort [2], we can obtain a work bound of $O(n \log \log n \log \sigma)$ and span bound of $O(\log n)$ for all alphabets. We describe a variant of this algorithm that is work-efficient and has $O(\log n \max(1, \log \sigma / \log \log n))$ span for all alphabets. In addition to having good theoretical bounds, our algorithms are also efficient in practice. We implement our algorithms using Cilk Plus [20] and show experiments on a 40-core shared-memory machine (with hyper-threading) indicating that they outperform the existing parallel algorithms for wavelet tree construction by 2–12x and achieve up to 27x speedup over the sequential algorithm. We then discuss the parallel construction of rank/select structures on binary sequences, which are an essential component to wavelet trees. Finally, we discuss how to adapt our algorithms to constructing variants of wavelet trees—multiary wavelet trees, Huffman-shaped wavelet trees, and wavelet matrices.

2 Preliminaries

We state complexity bounds of algorithms in the work-span model, where the *work* T_1 is equal to the number of operations required and the *span* T_∞ is equal to the number of time steps required. Then if P processors are available, using Brent’s scheduling theorem [19] we can bound the running time by $O(T_1/P + T_\infty)$. The parallelism of an algorithm is equal to $O(T_1/T_\infty)$. For sequential algorithms, work and span are equivalent. We say a parallel algorithm is *work-efficient* if its asymptotic work complexity matches that of the best sequential algorithm.

We denote a sequence of length n by $S = s_0 s_1 \dots s_{n-1}$, where s_i is the i ’th symbol of S . We denote an alphabet by $\Sigma = [0, \dots, \sigma - 1]$, where σ is the size of the alphabet. For a sequence S , the operation $\text{access}(S, i)$ returns the symbol at position i of S . The operation $\text{rank}_c(S, i)$ returns the number of times c appears in S from positions 0 to i . The operation $\text{select}_c(S, i)$ returns the position of the i ’th occurrence of c in S .

A *wavelet tree* is a data structure that supports access, rank and select operations on a sequence in $O(\log \sigma)$ time (in this paper, we use $\log x$ to mean the base 2 logarithm of x unless specified otherwise). The standard wavelet tree is a binary tree where each node represents a range of the symbols in Σ using a bitmap (binary sequence). We assume $\sigma \leq n$ as the symbols can be mapped to a contiguous range if this does not hold. The structure of the wavelet tree is defined recursively as follows: The root represents the symbols $[0, \dots, \sigma - 1]$. A node v which represents the symbols $[a, \dots, b]$ stores a bitmap which has a 0 in position i if the i th symbol in the range $[a, \dots, b]$ is in $[a, \dots, 2^{\lceil \log(b-a) \rceil - 1} - 1]$ and 1 otherwise. If

$b - a > 1$ then it will have a left child that represents the symbols $[a, \dots, 2^{\lceil \log(b-a) \rceil - 1} - 1]$ and a right child that represents the symbols $[2^{\lceil \log(b-a) \rceil - 1}, \dots, b]$. The recursion stops when the range is 2 or less. We note that the original wavelet tree description in [15] splits the range into approximately half for each child, however the definition we use gives the same query times and leads to a simpler description of our algorithms.

Along with the bitmaps, each node stores a *succinct* rank/select structure (whose size is sub-linear in the bitmap length) to allow for constant time rank and select queries. The structure of a wavelet tree requires $n \lceil \log \sigma \rceil + o(n \log \sigma)$ bits (the lower order term is for the rank/select structures). The tree topology (parent and child pointers) requires $O(\sigma \log n)$ bits, though this can be reduced or removed by modifying the queries accordingly [21, 6].

We will use the basic parallel primitives, prefix sum and filter [19]. *Prefix sum* takes an array X of length n , an associative binary operator \oplus , and an identity element \perp such that $\perp \oplus x = x$ for any x , and returns the array $(\perp, \perp \oplus X[0], \perp \oplus X[0] \oplus X[1], \dots, \perp \oplus X[0] \oplus X[1] \oplus \dots \oplus X[n-1])$. We will use \oplus to be the $+$ operator on integers. *Filter* takes an array X of length n , a predicate function f and returns an array X' of length n' containing the elements in $x \in X$ such that $f(a)$ returns true, in the same order that they appear in X . Filter can be implemented using prefix sum, and both require $O(n)$ work and $O(\log n)$ span [19].

3 Previous Work

Fuentes-Sepulveda et al. [12] describe a parallel algorithm for constructing a wavelet tree. They observe that for an alphabet where the symbols are contiguous in $[0, \sigma - 1]$, the node at which a symbol s is represented at level i of the wavelet tree can be computed as $s \gg \lceil \log \sigma \rceil - i$ in constant time. With this observation they can compute the bitmaps of each level independently. Each level is computed sequentially, requiring $O(n)$ work and span. Thus, their algorithm requires an overall work of $O(n \log \sigma)$ and $O(n)$ span. They describe a second algorithm which splits the input sequence into P sub-sequences, where P is the number of processors available. Each sub-sequence is computed sequentially and independently. Then the sub-sequences are merged. The merging step is non-trivial and requires $O(n)$ span. Thus the algorithm again requires $O(n \log \sigma)$ work and $O(n)$ span.

Since individual queries require sequential traversal of the wavelet tree, they cannot be parallelized. However multiple queries can be answered in parallel since they do not modify the tree. Furthermore, they can be batched to take advantage of cache locality [12].

Arroyuelo et al. [1] explore the use of wavelet trees in distributed search engines. They do not construct the wavelet tree for the entire text in parallel, but instead sequentially construct the wavelet tree for parts of the text on each machine.

Tischler [29] and Claude et al. [8] discuss how to reduce the space usage of sequential wavelet tree construction. Foschini et al. [11] describe an improved algorithm for sequentially constructing the wavelet tree in compressed format, requiring $O(n + \min(n, nH_h) \log \sigma)$ work, where H_h is the h 'th order entropy of the input. This bound can be better than $O(n \log \sigma)$ for compressible sequences (though still $O(n \log \sigma)$ in the worst case). The approach only works if the object produced is the wavelet tree compressed using run-length encoding. Parallelizing these techniques is an interesting direction for future work.

Succinct data structures supporting rank and select queries on binary sequences in con-

stant time have been widely studied (see e.g. [18, 5, 27, 25, 3]). They have many uses in succinct data structures, and are an essential component of wavelet trees. There has also been significant research on adapting these structures to practice (see [30] and the references within). The space and query times of wavelet trees with different shapes and using different compression techniques were experimentally studied by Grossi et al. [17].

4 Parallel Wavelet Tree Construction

We now describe our algorithms for wavelet tree construction. The construction requires a rank/select data structure for binary sequences. For now we assume that such structures can be created in linear work and logarithmic depth, and defer the discussion to Section 6.

Our first algorithm, *levelWT*, constructs the wavelet tree level-by-level. On each level, the nodes and their bitmaps are constructed in parallel in $O(n)$ work and $O(\log n)$ span, which gives an overall complexity of $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ span since there are $O(\log \sigma)$ levels in the tree. The pseudo-code for *levelWT* is shown in Figure 1. We maintain a bitmap of length n shared by all nodes on each level (Line 4). Nodes will simply store its start point in this array along with its bitmap length. To keep track of which nodes need to be constructed on each level, we maintain an array A of information for nodes to be added at the next level. Each entry in A stores the starting point (*start*) in the level bitmap for the node, the bitmap length (*len*), node identifier (*id*) and length of the alphabet range that it represents (*range*). We represent an entry of A as a 4-tuple (*start, len, id, range*).

Initially, the array A contains just the root node, with a starting index of 0, length of n (it represents all elements), node ID of 0, and a range of σ (Line 2). The array A' is used as the output array for the level. Then the algorithm proceeds one level at a time for $\lceil \log \sigma \rceil$ levels (Line 3). The bitmaps on each level are determined by the $l - 1$ 'st highest bit in the symbols, so we use a mask to determine the sign of this bit in the symbols (Line 4). Each node needs to determine its offset into the bitmap B (Line 4), and this is computed using a prefix sum and the offsets are stored in array O (Lines 5–7). We then loop through all the nodes on the current level in parallel (Lines 8–24). For each node, it sets its bitmap pointer and length in the Nodes array (Line 10). If the alphabet range of the node is 2 or less, then it has no children (Line 11). We then just loop over its symbols in S in parallel, and set the bit in the bitmap according to the sign of the symbol's $l - 1$ 'st highest bit (Lines 12–13). Otherwise, we must rearrange the symbols in S (stored into S') so that they are in the correct order on the next level. To do this in parallel, we first count the number of symbols that will go to the left child ($l - 1$ 'st bit is 0), and the offset of each such symbol using a prefix sum (Lines 15–17). With the prefix sum array X and result X_s , the symbols that go to the right child can be computed as well using the formula $X_s + i - X[i]$ —the number of symbols with an $l - 1$ 'st bit of 0 is X_s , so this is the number of symbols to offset, and then the number of symbols with $l - 1$ 'st bit of 1 up to index i is $i - X[i]$ (this is similar to rank queries on binary sequences). In Lines 18–20, the bits in the bitmap and positions in S' are set in parallel.

Children nodes are placed into A' in Lines 21–22 if the number of symbols represented per child is greater than 0. The node IDs in the entries are computed as twice the current node ID for the left child and twice the current node ID plus one for the right child. The

```

1: procedure LEVELWT( $S, n, \sigma$ )
2:   Nodes = {},  $L = \lceil \log \sigma \rceil$ ,  $S' = S$ ,  $A = \{(0, n, 0, \sigma)\}$ ,  $A' = \{\}$ 
3:   for  $l = 0$  to  $L - 1$  do ▷ Process level by level
4:     mask =  $2^{L-l-1}$ ,  $B$  = bitmap of length  $n$  ▷ Mask and bitmap for this level
5:      $O = \{\}$  ▷ Used to store offsets into bitmap
6:     parfor  $j = 0$  to  $|A| - 1$  do { $O[j] = A[j].len$ }
7:     Perform prefix sum on  $O$  to get offsets into  $B$ 
8:     parfor  $j = 0$  to  $|A| - 1$  do
9:       start =  $A[j].start$ , len =  $A[j].len$ , id =  $A[j].id$ ,  $r = A[j].range$ 
10:      Nodes[id].bitmap =  $B + O[j]$ , Nodes[id].len = len
11:      if  $r \leq 2$  then ▷ Node has no children
12:        parfor  $i = 0$  to len - 1 do
13:          if ( $S[i + start] \& \text{mask} \neq 0$ ) then  $B[O[j] + i] = 1$  else  $B[O[j] + i] = 0$ 
14:        else
15:           $X = \{\}$  ▷ Array used to store target positions into  $S'$ 
16:          parfor  $i = 0$  to len - 1 do { if ( $S[i + start] \& \text{mask} = 0$ )  $X[i] = 1$  else  $X[i] = 0$  }
17:          Perform prefix sum on  $X$  to get offsets of “left” characters ( $X_s$  is the result)
18:          parfor  $i = 0$  to len - 1 do
19:            if ( $S[i + start] \& \text{mask} \neq 0$ ) then  $B[O[j] + i] = 1$ ,  $S'[start + X_s + i - X[i]] = S[i + start]$ 
20:            else  $B[O[j] + i] = 0$ ,  $S'[start + i] = S[i + start]$ 
21:            if  $X_s > 0$  then  $A'[2j] = (start, X_s, id, 2^{\lceil \log r \rceil - 1})$  ▷ Left child
22:            if  $(len - X_s) > 0$  then  $A'[2j + 1] = (X_s, len - X_s, 2id + 1, r - 2^{\lceil \log r \rceil - 1})$  ▷ Right child
23:          Filter out empty  $A'$  entries and store into  $A$ 
24:          swap( $S, S'$ )
25:   return Nodes

```

Figure 1: levelWT: Inter-level parallel algorithm for wavelet tree construction.

starting point in the bitmap of the next level is the same as the current starting point for the left child, and is the current starting point plus the number of elements on the left for the right child. The length is stored from the computation before. If the current range is r then the range of the left child is $2^{\lceil \log r \rceil - 1}$ and range of the right child is $r - 2^{\lceil \log r \rceil - 1}$.

After each level, the non-empty entries of A' are filtered out using a parallel prefix sum and stored into A (Line 23). The roles of S and S' are swapped for the next level (Line 24).

We note that setting the bits in B (Lines 13, 19 and 20) must be done atomically since multiple threads may write to the same word concurrently. This can be implemented using a loop with a compare-and-swap until successful. An optimization is to only perform atomic writes if a word is shared between two nodes on a level (there can be at most 2 shared words, since the bits for each node are contiguous in B), and for the remaining words in parallel at the granularity of a word. Inside each word, the updates are done sequentially. This optimization allows us to use non-atomic writes for all but two words for each node.

We also note that we are able to stop early when the range of a node is 2 or less since the construction of the previous level provided this information. The algorithm of Fuentes-Sepulveda processes all levels independently (and also our second algorithm described later), so it is not easy to stop early.

We now analyze the complexity of the levelWT algorithm. For each level, there is a total of $O(n)$ work performed, since each symbol is processed a constant number of times. The prefix sums on Lines 7 and 17, and the filter on Line 23 require linear work and

$O(\log n)$ span. There are a total of $\lceil \log \sigma \rceil$ levels so the total work is $O(n \log \sigma)$ and span is $O(\log n \log \sigma)$. Since $\sigma \leq n$, we have a span bound that is poly-logarithmic in n . We obtain the following theorem:

Theorem 4.1. *levelWT requires $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ span.*

We now describe our second wavelet tree construction algorithm, which constructs all level of the wavelet tree in parallel. We refer to this algorithm as *sortWT*. Since preceding levels cannot provide information to later levels, we must do independent computation per level to obtain the correct ordering of the sequence for the level. We will make use of the observation of Fuentes-Sepulveda et al. [12] that the node at which a symbol s is represented at level l of the wavelet tree (the root is at level 0) is encoded in the top l bits. For level l , our algorithm sorts S using the top l bits as the key, which gives the correct ordering of the sequence for the level. We note that the sort must be stable since the relative ordering of nodes with the same top l bits must be preserved in the wavelet tree. Since the keys are in a bounded range ($\sigma \leq n$), we can use parallel integer sorting.

```

1: procedure SORTWT( $S, n, \sigma$ )
2:   Nodes = {},  $L = \lceil \log \sigma \rceil$ 
3:   parfor  $l = 0$  to  $L - 1$  do
4:     mask =  $2^{L-l-1}$ ,  $B$  = bitmap of length  $n$  ▷ Mask and bitmap for this level
5:     if  $l = 0$  then ▷ No sorting required for first level
6:       parfor  $i = 0$  to  $\text{len} - 1$  do { if ( $S[i] \& \text{mask} \neq 0$ )  $B[i] = 1$  else  $B[i] = 0$  }
7:       Nodes[0].bitmap = 0, Nodes[0].len =  $n$ 
8:     else
9:        $S' = S$  stably sorted by top  $l$  bits
10:      parfor  $i = 0$  to  $\text{len} - 1$  do { if ( $S'[i] \& \text{mask} \neq 0$ )  $B[i] = 1$  else  $B[i] = 0$  }
11:       $O =$  indices  $i$  such that  $(S'[i] \gg L - l) \neq (S'[i - 1] \gg L - l)$ 
12:      parfor  $i = 0$  to  $|O| - 1$  do
13:        Nodes[ $2^l - 1 + i$ ].bitmap =  $O[i]$ , Nodes[ $2^l - 1 + i$ ].len =  $O[i + 1] - O[i]$ 
14:   return Nodes

```

Figure 2: sortWT: Sorting-based parallel algorithm for wavelet tree construction.

The pseudo-code for sortWT is shown in Figure 2. Similar to Figure 1, we use a mask and a bitmap B for each level (Line 4). For the first level, no sorting of S is required, so we simply fill the bitmap according to the highest bit of each symbol (Lines 5–6). For the remaining levels, we must first stably sort S by the top l bits to obtain the symbols in the correct order (Line 9). The sorted sequence is stored in S' . The bitmap is filled as before (Line 10). To compute the length of each node's bitmap we use a prefix sum and filter to find all the indices where the symbol's top l bits differ from the previous symbol's top l bits in S' (Line 11). These mark the bitmaps of each node since S' is sorted by the top l bits. The length of each bitmap can be computed by the difference in indices. Lines 12–13 set the bitmap pointers and lengths for the nodes on the current level. The IDs of the nodes of the current level start at $2^l - 1$, since there are up to $2^l - 1$ nodes in previous levels. Updates to the bitmaps are done in parallel at word granularity so atomic updates are not required.

We now discuss the complexity of the algorithm. The sort on Line 9 requires W_{sort} work and S_{sort} span. The prefix sum and filter on Line 11 requires $O(n)$ work and $O(\log n)$

span. The overall work is $O(W_{\text{sort}} \log \sigma)$ and since all levels can be computed in parallel, the overall span is $O(S_{\text{sort}} + \log n)$ for sorting and prefix sums. This gives the following theorem:

Theorem 4.2. *sortWT requires $O(W_{\text{sort}} \log \sigma)$ work and $O(S_{\text{sort}} + \log n)$ span.*

Using stable integer sorting we obtain a work bound of $O(n \log \sigma)$ and span bound of $O(\sigma^\epsilon + \log n)$ for $0 < \epsilon < 1$ [26, 19]. For $\sigma = O(\log^c n)$ for any constant c this gives a span bound of $O(\log n)$ (by setting ϵ appropriately). If we sacrifice work-efficiency, we can use a non-work-efficient stable integer sorting algorithm [2] to obtain a work bound of $O(n \log \log n \log \sigma)$ and span bound of $O(\log n)$.

Due to the high space usage and hence memory footprint of sortWT (discussed later in this section), we that processing the levels one-by-one leads to improved performance in practice, although increases the span by a factor of $O(\log \sigma)$. We call this modified version *msortWT*.

Relating sortWT to levelWT. We also note that sortWT can be modified to have the same high-level structure and work/depth complexity as levelWT as follows. We process the levels one-by-one and observe that we can save the sorted sequence S' to use in the next level. On level l , S' is already sorted by the top $l - 1$ bits, so we only need to sort l by the l 'th highest bit within each of the sub-sequences sharing the same top $l - 1$ bits. The sub-sequences can be identified with a prefix sum and filter, and for each sub-sequence we apply an integer sort. Since the keys we are sorting on have a constant range (they are either 0 or 1), the integer sort runs in linear work and $O(\log n)$ span. This gives an overall work of $O(n \log \sigma)$ and overall span of $O(\log n \log \sigma)$, matching that of levelWT. This approach is similar to levelWT in that each node is reordering the elements in the sequence it represents into elements for its left and right children, independently of other nodes. In some sense, Lines 15–20 of levelWT is implementing an integer sort on a range of size 2. In practice, we found this modified version of sortWT to be more expensive due to the overhead of calling a general-purpose integer sort.

Theoretically we can improve the span by constructing in parallel chunks of $O(\log \log n)$ levels (the i 'th level in the chunk sorting by i bits), and saving the result of the last level in the chunk to be used for the next chunk. This will involve keys of at most $O(\log \log n)$ bits, or keys in a range up to $O(\log n)$, so the integer sort can then still be done in $O(\log n)$ span. This improves the overall span to $O(\log n \log \sigma / \min(\log \sigma, \log \log n))$, or equivalently $O(\log n \max(1, \log \sigma / \log \log n))$.

Space usage. We analyze the space usage of the algorithms, excluding the input and output, which require $O(n \log \sigma)$ bits. levelWT requires two auxiliary arrays for the prefix sum each level (which can be reused per level), which takes $O(n \log n)$ bits. For sortWT, since all levels are processed in parallel, and each level requires $O(n \log n)$ bits for the integer sort, the total space usage is $O(n \log n \log \sigma)$ bits. msortWT reduces the space usage to $O(n \log n)$ bits, since the levels are processed one-by-one.

5 Experiments

Implementations. We compare our implementations of the parallel algorithms with existing parallel implementations and a sequential implementation. We implement levelWT, sortWT using work-efficient integer sorting, and msortWT. The implementations all use the levelwise representation of the bitmaps, where one bitmap of length n is stored per level, and nodes have pointers into the bitmaps. For levelWT, atomic writes to the bitmaps are only performed when a word on the bitmap is shared by another node. We use the parallel prefix sum, filter and stable integer sorting routines from the Problem Based Benchmark Suite [28]. We implement a sequential version of wavelet tree construction (*serialWT*) that uses a levelwise bitmap representation, and its performance is competitive with the times of the sequential algorithm reported in [17]. We also tried the serial implementation in SDSL [13] for constructing a balanced wavelet tree, but found it to be slower than serialWT on the inputs. However, the SDSL implementation is more space-efficient, and sometimes faster on the Burrows-Wheeler transformed inputs, and the experiments and discussion can be found in the Appendix.

We compare our implementations with the implementation of Fuentes-Sepulveda et al. [12] that computes each level of the wavelet tree independently in parallel (*FEFS*). The code was kindly provided to us by the authors. We observed that the speedup of the code was limited for large alphabets as it performed a number of memory allocations proportional to the number of nodes in the wavelet tree in parallel, which causes high contention. We also tried their second implementation (*FEFS2*) in which each processor computes a partial wavelet tree which are then merged together, but unfortunately, we were unable to get it to run on most input files.

Experimental Setup. We run our experiments on a 40-core (with 2-way hyper-threading) machine with 4×2.4 GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache), and 256GB of main memory. The parallel codes use Cilk Plus [20] to express parallelism. We compile our code with the g++ compiler version 4.8.0 (which supports Cilk Plus) with the -O2 flag. The times reported are based on a median of 3 trials.

We use a variety of real-world and artificial sequences. The real-world sequences include strings from <http://people.unipmn.it/manzini/lightweight/corpus/>, XML code from a Wikipedia sample (*wikisamp*), protein data from <http://pizzachili.dcc.uchile.cl/texts/protein/> (*proteins*), the human genome from <http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz> (*HG18*), and the document array of text collections (*trec8*). The artificial sequences (*rand*), parameterized by σ , are generated randomly by drawing each symbol uniformly from the range $[0, \dots, \sigma - 1]$. We use these sequences to experiment with large alphabets. The length and alphabet size of the sequences are listed in Table 1.

Due to the various choices for rank/select each of which has different space/time trade-offs we do not include their construction times in the wavelet tree construction time. Furthermore, the time for rank/select structure construction is small compared to wavelet tree construction. We modified the FEFS code accordingly. The times for our implementations include generating the parent and child pointers for the nodes, although these could be removed using techniques from [21, 6]. FEFS and FEFS2 do not generate these pointers.

Results. Table 1 shows the single thread and 40 core running times (with hyper-threading)

Text	n	σ	serialWT (T_1)	levelWT (T_1)	levelWT (T_{40})	sortWT (T_1)	sortWT (T_{40})	msortWT (T_1)	msortWT (T_{40})	FEFS (T_1)	FEFS (T_{40})	FEFS2 (T_1)	FEFS2 (T_{40})
chr22	$3.35 \cdot 10^7$	4	0.486	0.611	0.018	0.786	0.046	0.768	0.029	0.817	0.419	0.708	0.181
etext99	$1.05 \cdot 10^8$	146	4.12	6.99	0.28	10.5	0.393	9.79	0.364	8.72	1.51	—	—
HG18	$2.83 \cdot 10^9$	4	35.5	45.5	1.32	57.7	1.88	56.9	1.67	61.4	32.2	52.7	15.5
howto	$3.94 \cdot 10^7$	197	1.65	2.69	0.105	4.07	0.161	3.86	0.144	2.95	0.63	—	—
jdk13c	$6.97 \cdot 10^7$	113	2.51	3.9	0.159	6.13	0.234	5.63	0.22	5.18	0.977	—	—
proteins	$1.18 \cdot 10^9$	27	31.3	52.2	1.82	75.3	2.59	71.3	2.28	59.9	13.9	47.4	4.13
rtail96	$1.15 \cdot 10^8$	93	3.54	5.88	0.231	10.2	0.373	9.39	0.34	8.04	1.56	6.11	2.13
rfc	$1.16 \cdot 10^8$	120	3.8	6.51	0.261	10.1	0.37	9.28	0.348	8.75	1.6	—	—
sprot34	$1.1 \cdot 10^8$	66	3.69	6.26	0.248	9.48	0.36	8.8	0.328	7.38	1.51	5.76	0.67
trec8	$2.43 \cdot 10^8$	528155	33.3	50.4	2.08	138	5.5	104	4.55	*	*	—	—
w3c2	$1.04 \cdot 10^8$	256	3.82	6.66	0.275	10.6	0.388	9.78	0.357	8.59	1.39	—	—
wikisamp	10^8	204	3.52	6.16	0.264	9.78	0.374	9.08	0.349	7.64	1.36	—	—
rand-2 ⁸	10^8	2 ⁸	5.76	8.58	0.36	14.3	0.652	12.1	0.533	10	1.52	8.31	0.857
rand-2 ¹⁰	10^8	2 ¹⁰	6.88	11	0.456	19	0.857	15.9	0.708	12.6	2.65	10.2	0.99
rand-2 ¹²	10^8	2 ¹²	8.32	12.4	0.525	24.5	1.11	20.4	0.922	16.13	5.6	12.2	1.01
rand-2 ¹⁶	10^8	2 ¹⁶	11.2	16.4	0.655	36	1.59	29.5	1.34	28.8	13.1	15.5	1.49
rand-2 ²⁰	10^8	2 ²⁰	14	20.4	0.772	49.5	2.19	40.6	1.85	*	*	20.9	4.31

Table 1: Comparison of running times (seconds) of wavelet tree construction algorithms on a 40-core machine with hyper-threading. T_{40} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread. *Did not finish due to thrashing.

on the inputs for the different implementations. The parallel times reported are the best time for all thread counts between 1 and 80. For FEFS2, we only report numbers for the inputs on which the code ran successfully on.

Our results show that levelWT is faster than sortWT and msortWT both sequentially and in parallel. This is because sortWT and msortWT uses sorting, which has a larger overhead. msortWT is slightly faster than sortWT as for each level it already has a partially sorted sequence so there is less data movement from the sort. Compared to serialWT, levelWT is 1.3–1.8 times slower on a single thread, and 13–27 times faster on 40 cores with hyper-threading. The self-relative speedup of levelWT ranges from 23 to 34. sortWT and msortWT achieve self-relative speedups of 17–31 and 22–34, respectively. On 40 cores, our best parallel implementation outperforms FEFS and FEFS2 by 2–12 times, due to the limited parallelism in those algorithms. FEFS and FEFS2 only have about $O(\log \sigma)$ parallelism whereas the parallelism for our algorithms is much higher ($O(n/\log n)$ for levelWT, $O(n \log \sigma/(\sigma^\epsilon + \log n))$ for sortWT and $O(n/(\sigma^\epsilon + \log n))$ for msortWT). The parallelism in our algorithms is much higher than the number of cores available, and thus we achieve very good speedup.

In Figure 3, we plot the speedup of the parallel implementations relative to serialWT as a function of the number of threads for HG18 and rand-2¹⁶. For HG18, our implementations all scale well up to 40 cores, however FEFS and FEFS2 only scales up to 2 threads and 4 threads, respectively due to the small alphabet. For rand-2¹⁶, our implementations again exhibit good scalability. For FEFS, scalability is limited since it does 2¹⁶ memory allocations in parallel, which leads to high contention, and so it scales only up to 4 threads. FEFS2 is competitive with levelWT for up to 8 threads but with higher numbers of threads it does not scale well due to the linear-span merge step.

In Figure 4(a), we plot the 40-core parallel running time of our three implementations as a function of the alphabet size for random sequences of length 10⁸. We see that for fixed

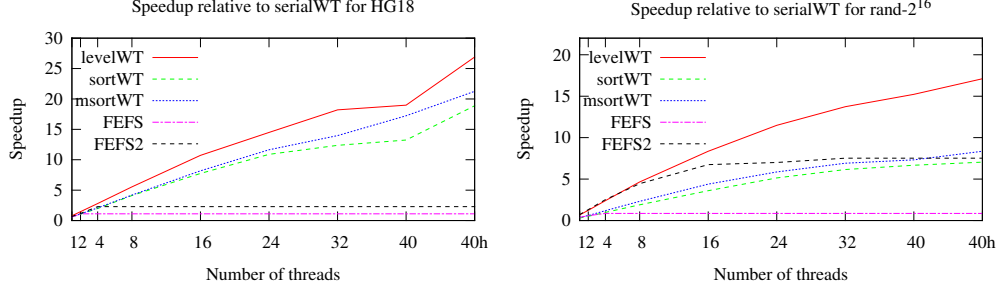


Figure 3: Speedup of implementations relative to serialWT for HG18 (left) and rand- 2^{16} (right). (40h) is 40 cores with hyper-threading.

n , the running times increase linearly with $O(\log \sigma)$, which is expected since the total work is $O(n \log \sigma)$. In Figure 4(b), we plot the running time of the implementations as a function of n for $\sigma = 2^8$ on random sequences, and see that the times increase linearly with n as expected. For both experiments, our algorithms exhibit similar speedups on 40 cores as we vary σ or n , since the core count is much lower than the parallelism T_1/T_∞ .

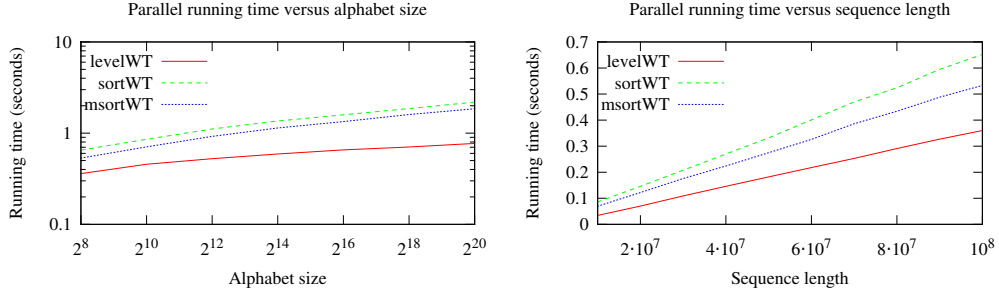


Figure 4: 40-core (with hyper-threading) running times versus σ (left, log-log scale) and versus n (right) on random sequences.

In summary, our experiments show that our three parallel algorithms for wavelet tree construction all scale well with the number of threads, input length and alphabet size. LevelWT outperforms sortWT and msortWT as it does not have the overheads of sorting, and outperforms the sequential wavelet tree construction algorithm with a modest number of threads. Our algorithms all outperform FEFS and FEFS2 with a modest number of threads as they exhibit much more parallelism (poly-logarithmic instead of linear span).

6 Parallel Construction of Rank/Select Structures

Wavelet trees make use of succinct rank/select structures which support constant time rank and select queries on binary sequences. Sequentially constructing these structures requires $O(n)$ work. In this section, we describe how to parallelize the construction of such a structure. This is necessary to achieve the poly-logarithmic span bounds for wavelet tree construction. We use the rank structure of Jacobson [18] and select structure of Clark [5].

The rank structure of Jacobson stores the rank of every $\log^2 n$ 'th bit in a first level directory, and the rank of every $\log n$ 'th bit in each of the ranges in a second level directory. Rank queries in each range of size $\log n$ can be answered by at most 2 table look-ups, where the table stores the rank of all bit-strings of length up to $\log n/2$. The first and second level directories can be constructed easily by first computing the prefix sums of the bit-string in $O(n)$ work and $O(\log n)$ span. The look-up table can be constructed in $o(n)$ work and $O(1)$ span, as we can compute the number of 1's in bit-strings of size $O(\log n)$ in $O(1)$ work and span, and there are $O(2^{\log n/2} \log n) = O(\sqrt{n} \log n)$ such bit-strings.

Clark's select structure stores the position of every $\log n \log \log n$ 'th 1 bit in a first level directory. Then for each range r between the positions, if $r \geq \log^2 n (\log \log n)^2$ the $\log n \log \log n$ answers in the range are stored directly. Otherwise the position of every $\log r \log \log n$ 'th 1 bit is stored in a second level directory. The sub-ranges r' in the second level directory are again considered, and if $r' \geq \log r' \log r (\log \log n)^2$, all answers in the range are stored directly. Otherwise, a look-up table is constructed for all bit-strings of length less than r' . To parallelize the construction of the select structure, we first compute the positions of all the 1 bits using a parallel filter in $O(n)$ work and $O(\log n)$ span. This allows all the sub-directories to be constructed in parallel. The look-up table can be constructed in $o(n)$ work and $O(1)$ span, similar to the rank structure.

More practical variants of these rank/select structures have been proposed, each with different space usage and query performance (see [30, 14] and the references within). The most recent ones [30, 14] follow a similar structure to the constructions described above, and can also be parallelized in $O(n)$ work and $O(\log n)$ span.

7 Extensions

The *Huffman-shaped wavelet tree*, where each node is placed at a level proportional to the length of its Huffman code, was introduced to improve compression and average query performance [11]. To construct this in parallel, we first compute the Huffman tree and prefix codes for each symbol using the algorithm by Edwards and Vishkin [9] in $O(\sigma + n)$ work and $O(\sigma + \log n)$ span. To figure out how to set the bitmaps in the internal nodes and rearrange S we must know the side of the tree each symbol is located. We can map each symbol to an integer corresponding to the location of its leaf in an in-order traversal of the Huffman tree, which can be done in parallel using an Euler tour algorithm in $O(\sigma)$ work and $O(\log \sigma)$ span [19]. At each internal node in the wavelet tree, the symbols to the left and to the right are in consecutive ranges, and we store the highest mapped integer of nodes in its left sub-tree, which can be done during the Euler tour computation. Then the decision of how to set the bitmap and where to place the symbol in S' can be made with a single comparison with the mapped integers. The construction then follows the strategy of levelWT. The overall work bound (including Huffman encoding) is $O(n \log \sigma)$ as the sum of bitmap sizes (which the number of times each symbol is processed is proportional to) is bounded by $O(n \log \sigma)$ [11]. The span is $O(\sigma \log n)$, as the tree can have up to σ levels.

Ferragina et al. [10] generalize the wavelet tree to a *multiary wavelet tree* where each node has up to h children for some value h , and stores arrays of symbols in the range $[0, \dots, h-1]$. The height of the tree is $O(\log_h \sigma)$. For $h = \log n / \log \log n$, the structure can

answer queries in $O(\log h / \log \log n)$ time (using a rank/select structure for larger alphabets). Our msortWT algorithm can be adapted to construct the multiary wavelet tree. We describe it for the case where $\log h$ is an integer, though it can be modified for the general case. We use $\log h$ bits in the key for the integer sort instead of a single bit. Instead of using a bitmap B we use an array where each entry holds a value in $[0, \dots, h-1]$ and sets it according to the value of the appropriate $\log h$ bits of each $S[i]$ (Lines 6 and 10 of Figure 2). There will be up to h^l nodes on level l , so the offset to the Nodes array on Line 14 is $h^l - 1$. There are $\log_h \sigma$ levels of computation. For $h = O(\log^c n)$, which is true in applications of multiary wavelet trees, the work bound is $O(n \log_h \sigma)$ and span is $O(\log n \log_h \sigma)$. The span can be improved to $O(\log n \log_h \sigma / \min(\log_h \sigma, \log \log n))$ by grouping the levels. One can use the original sortWT algorithm to improve the span to $O(\log n)$ when $\sigma = O(\log^c n)$, or sacrifice work-efficiency to obtain a $O(\log n)$ span for larger alphabets. We note that the rank/select structure on sequences with larger alphabets, used in the multiary wavelet tree, also needs to be parallelized, which we leave for further investigation.

The *wavelet matrix* [7] is a variant of the wavelet tree where on level l , all symbols with a 0 as their l 'th highest bit will be on the left side of the level's bitmap and all symbols with a 1 as their l 'th highest bit will be on the right side. Each level stores a single integer indicating the number of 0's on the level. It is shown that this representation still allows for logarithmic time queries [7] while being faster in practice. To modify levelWT to compute the wavelet matrix, we do not keep track of the node boundaries, nor reorder S . On level l , we simply count the number of symbols with their l 'th highest bit set to 0, and store their offsets into the level's bitmap using a prefix sum. Then the bitmap values are set in parallel. This gives an algorithm with $O(n \log \sigma)$ work, $O(\log n \log \sigma)$ span and $O(n)$ space. Since there are no dependencies among levels, we can compute all levels in parallel, reducing the span to $O(\log n)$, but increasing the space usage to $O(n \log n \log \sigma)$ bits.

Grossi and Ottaviano describe the wavelet trie, an extension of the wavelet tree for maintaining a dynamic sequence over a dynamic alphabet [16]. The parallelization of its construction and of performing batched updates is an interesting direction for future work.

8 Conclusion

We have described several parallel algorithms with poly-logarithmic depth for wavelet tree construction. Our algorithms improve upon the recent linear-span parallel algorithms by Fuentes-Sepulveda et al. [12]. On a 40 core machine, our implementations of our algorithms outperform the existing ones by 2–12x and achieves up to 27x speedup over the sequential algorithm on a variety of inputs. Our algorithms can be extended to variants of the wavelet tree. Future work include improving the space usage of our algorithms, and extending them to the GPU, distributed-memory and external-memory settings.

References

- [1] D. Arroyuelo, V. Gil-Costa, S. Gonzalez, M. Marin, and M. Oyarzun. Distributed search based on self-indexed compressed text. *Information Processing & Management*, 2012.

- [2] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 1991.
- [3] A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 1999.
- [4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994.
- [5] D. R. Clark. *Compact Pat Trees*. PhD thesis, 1996.
- [6] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, 2008.
- [7] F. Claude and G. Navarro. The wavelet matrix. In *SPIRE*, 2012.
- [8] F. Claude, P. K. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *SPIRE*, 2011.
- [9] J. A. Edwards and U. Vishkin. Parallel algorithms for Burrows-Wheeler compression and decompression. *Theor. Comput. Sci.*, 525, Mar. 2014.
- [10] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 2007.
- [11] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, 2006.
- [12] J. Fuentes-Sepulveda, E. Elejalde, L. Ferres, and D. Seco. Efficient wavelet tree construction and querying for multicore architectures. In *SEA*, 2014.
- [13] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA 2014*, 2014.
- [14] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 2013.
- [15] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, 2003.
- [16] R. Grossi and G. Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In *PODS*, 2012.
- [17] R. Grossi, J. S. Vitter, and B. Xu. Wavelet trees: From theory to practice. In *CCP*, 2011.
- [18] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, 1988.
- [19] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.
- [20] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 2010.
- [21] V. Makinen and G. Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 2007.
- [22] V. Makinen and G. Navarro. On self-indexing images—image compression with added value. In *DCC*, 2008.
- [23] C. Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 2012.
- [24] G. Navarro. Wavelet trees for all. In *CPM*, 2012.
- [25] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 2001.
- [26] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 1989.
- [27] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, 2002.
- [28] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.
- [29] G. Tischler. On wavelet tree construction. In *CPM*, 2011.
- [30] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, 2013.

Text	n	σ	serialWT (T_1)	SDSL (T_1)	levelWT (T_1)	levelWT (T_{40})	sortWT (T_1)	sortWT (T_{40})	msortWT (T_1)	msortWT (T_{40})	FEFS (T_1)	FEFS (T_{40})	FEFS2 (T_1)	FEFS2 (T_{40})
chr22	$3.35 \cdot 10^7$	4	0.48	0.86	0.59	0.017	0.762	0.044	0.741	0.029	0.77	0.4	0.66	0.19
etext99	$1.05 \cdot 10^8$	146	2.83	2.9	5.79	0.266	9.47	0.377	9.04	0.343	7.36	1.27	—	—
HG18	$2.83 \cdot 10^9$	4	31	63.2	36.7	1.17	43.3	1.6	41.4	1.42	55.8	29.9	49.1	14.9
howto	$3.94 \cdot 10^7$	197	1.2	1.2	2.14	0.1	3.58	0.157	3.42	0.14	2.43	0.6	—	—
jdk13c	$6.97 \cdot 10^7$	113	1.51	1.13	2.88	0.152	5.23	0.223	4.9	0.207	3.91	0.78	—	—
proteins	$1.18 \cdot 10^9$	27	26.1	30.2	43.9	1.69	65.1	2.34	61	2.11	47.7	11.17	40.3	4.12
rtail96	$1.15 \cdot 10^8$	93	2.11	2.31	4.61	0.221	8.83	0.35	8.28	0.327	6.31	1.33	—	—
rfc	$1.16 \cdot 10^8$	120	2.53	2.73	5.26	0.251	8.95	0.355	8.52	0.335	7.1	1.33	—	—
sprot34	$1.1 \cdot 10^8$	66	2.55	2.6	5.19	0.24	8.53	0.343	8.14	0.31	5.93	1.26	—	—
w3c2	$1.04 \cdot 10^8$	256	2.3	1.85	5.13	0.263	9.76	0.375	8.6	0.349	6.65	1.21	4.66	0.69
wikisamp	10^8	204	2.22	1.91	5.06	0.253	8.67	0.353	8.17	0.325	5.97	1.21	—	—

Table 2: Running times (seconds) of wavelet tree construction algorithms on a 40-core machine with hyper-threading *on the Burrows-Wheeler transform of the text*. T_{40} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread.

A Appendix

For certain applications (see [24, 23]), the wavelet tree is constructed on the Burrows-Wheeler transform of the sequence. In Table 2, we report the running times of the algorithms on the Burrows-Wheeler transform of the real-world texts. For sequential times, we also report the wavelet tree implementation from SDSL [13], which performs well on sequences with many sequences of repeated characters. It uses an optimization that groups the writes into the bitmap for repeated characters into a single write. This gives an advantage for the Burrows-Wheeler transformed texts, in which there are often many sequences of repeated characters. In contrast to serialWT, the SDSL implementation generates the wavelet tree structure first before updating the bitmaps. In the column labeled *SDSL* in Table 2, we report the SDSL times using the balanced tree option and without the construction of the rank/select structures. For our implementations, we use an optimization where consecutive 1 bits in a word are written together instead of individually. This is similar to SDSL but it finds consecutive 1 bits on each level instead of consecutive repeated characters in the original sequence. The optimization slightly improves the running time for the Burrows-Wheeler transformed texts, but makes negligible difference on the original texts. For some inputs, SDSL performs slightly faster than serialWT, though it is slower than or performs about the same as serialWT on other inputs.

Overall, the times are faster on the Burrows-Wheeler transformed texts than on the original texts. This is because the bits of the same value are grouped into larger chunks in the bitmaps, and hence there are fewer words that actually need to be updated (words with all 0 bits do not need to be updated after initialization). Furthermore, the optimization of writing consecutive 1 bits together has more of a benefit in this setting. The relative performance among our parallel algorithms remains the same, with levelWT being the fastest, followed by msortWT and then sortWT. Compared to the faster of serialWT and SDSL for each input, levelWT is 1.7–2.8 times slower on a single thread and 7–28 times faster on 40 cores with hyper-threading. The self-relative speedups of levelWT, sortWT and msortWT are 18–35, 17–28 and 24–29, respectively. Compared to FEFS and FEFS2, our fastest implementation levelWT is 2–13 times faster in parallel.