Bit-Parallel (Compressed) Wavelet Tree Construction

Patrick Dinklage*, Johannes Fischer*, Florian Kurpicz†, and Jan-Philipp Tarnowski*

*TU Dortmund University
Otto-Hahn-Str. 14
Dortmund, 44227, Germany
first.last@cs.tu-dortmund.de

†Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, 76131, Germany
kurpicz@kit.edu

Abstract

The wavelet tree is a data structure that indexes a text over an integer alphabet for efficient rank and select queries. Using the Huffman encoding, it can be stored in zero-order entropy-compressed space. We present a highly engineered open source implementation of an efficient sequential construction algorithm that makes use of bit parallelism via vector instructions. On hardware featuring ultrawide registers of up to 512 bits, it outperforms the currently fastest known practical sequential construction algorithms.

1 Introduction and Contributions

Wavelet trees are an integral building block for many compressed text indices, e.g., the FM-index [7] and the r-index [9]. Further applications include but are not limited to compression and computational geometry. We present the currently fastest sequential wavelet tree construction algorithm: the first open source implementation of the $\mathcal{O}\left(n \lg \sigma / \sqrt{\lg n}\right)$-time construction algorithm due [1, 14]. While it has already been implemented by Kaneta [13], their source code is not publicly available. Further more, our implementation supports ultrawide registers up to 512 bits. Using a common baseline algorithm, we can conclude that these ultrawide registers allow for a speedup of up to 2.5 compared with Kaneta's implementation. We extend the algorithm to also construct the Huffman-shaped wavelet tree and show that it outperforms the fastest practical algorithms known to date. Our results apply as well to the wavelet matrix [3], a popular alternative representation of the wavelet tree.

2 Preliminaries

Let $T \in \Sigma^n$ be the input string of length n over an integer alphabet $\Sigma = [0, \sigma)$ of size $\sigma \leq n$. For $i, j \in [0, n)$ and $i \leq j$, we denote by $T[i]$ the i-th character in T and by $T[i..j]$ the substring of T starting at position i and ending at position j, both included. For $a \in \Sigma$ and $i \in \mathbb{N}$, we define $\mathrm{rank}_a(T, i)$ to be the number of occurrences of a in $T[0..i]$. For convenience, we say $\mathrm{rank}_a(T) = \mathrm{rank}_a(T, n)$. The inverse operation, $\mathrm{select}_a(T, k)$ for $k \geq 1$, returns the position in T of the k-th occurrence of a. We call a string $B \in \{0, 1\}^*$ over the binary alphabet a *bit vector*. We argue about the running times of algorithms using the word RAM model of computation, allowing access to words of size $w = \Omega(\lg n)$ bits in constant time. By default, logarithms are base-two.

In the word RAM, we can *pack* up to $N = \lfloor w/\tau \rfloor$ *subwords* of size $\tau \leq w$ bits in a word and access them individually in constant time. For simplicity, we assume that τ

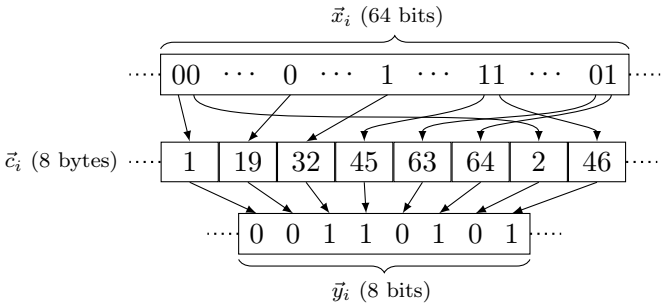


Figure 1: Example of the bit shuffle instruction with selection vector $\vec{c}$ for the i-th 64-bit block of the input vector $\vec{x}$, producing the i-th 8-bit block of the result vector $\vec{y}$.

divides w and thus N is an integer. This notion can be extended to a *packed list*: an array of M subwords can be packed into $\lceil M\tau/w \rceil$ consecutive words with constant-time subword access. For example, we can store T in a packed list of $\lceil n\lceil \lg \sigma \rceil / w \rceil$, where each subword is a character encoded using $\lceil \lg \sigma \rceil$ bits.

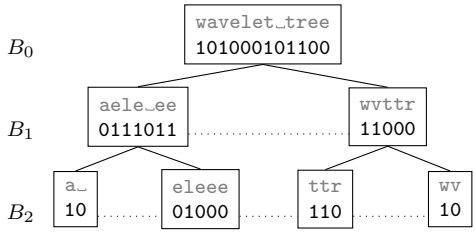
Advanced CPU Instructions. In the following, we introduce advanced instructions featured by modern CPUs that we use in this work. For a technical documentation, we refer to [12]. Given a word x, let x_i denote the i-most significant bit of x for $i \in [1, w]$. We use the following commonly available instructions on words:

- *Population Count* (`popcount`): Return $\text{rank}_1(x) = \sum_{i=1}^{w} x_i$.
- *Bit Extract* (`pext`): Filters the bits from x marked by a selection bitmask b and stores them consecutively in the result word y such that

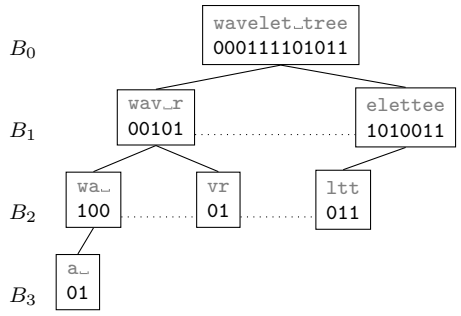
$$y_i := \begin{cases} x_{\text{select}_1(b,i)} & \text{if } \text{rank}_1(b) \geq i \\ 0 & \text{otherwise} \end{cases}, \text{ for } i \in [1, w].$$

We can also treat a word as a *vector* $\vec{x} = (x_1, x_2, \ldots, x_N)$ of packed τ-bit subwords x_1 to x_N and use the following vector instructions from the AVX-512 instruction sets:

- *Parallel Compare* (`vpcmp`): Given two vectors $\vec{x} = (x_1, \ldots, x_N)$ and $\vec{y} = (y_1, \ldots, y_N)$ and a binary comparison operator $\preceq$, computes $\vec{z} \in \{0,1\}^N$ with $z_i = 1$ iff $x_i \preceq y_i$.
- *Compress* (`vpcompress`): Filters subwords from $\vec{x} = (x_1, \ldots, x_N)$ according to the selection mask $\vec{b'} \in \{0,1\}^N$, similar to `pext`, but for subwords instead of bits.
- *Bit Shuffle* (`vpshufbit`): Gathers $w/8$ bits from $\vec{x}$. We partition $\vec{x}$ into 64-bit blocks such that $\vec{x} = (\vec{x}_1, \ldots, \vec{x}_{w/64})$ with $\vec{x}_i \in \{0,1\}^{64}$ for $i \in [1, w/64]$. The selection vectors $\vec{c}_i \in [1, 64]^8$ state 8 positions of bits to be gathered, in that order, from $\vec{x}_i$ into the result block $\vec{y}_i$. Fig. 1 shows an example. This is done in parallel for all $w/64$ blocks and the result is packed into the output vector $\vec{y} \in \{0,1\}^{w/8}$.
- *Permute* (`pshufb` / `vperm`): Permutes the subwords of $\vec{x} = (x_1, \ldots, x_N)$ according to the vector $\vec{s} = (s_1, \cdots, s_N) \in \{1, 64\}^N$ of bytes to vector $\vec{y} := (x_{s_1}, \ldots, x_{s_N})$.



(a) Binary wavelet tree of height $\lceil \lg \sigma \rceil = 3$. Each level stores exactly $n = 12$ bits.



(b) Huffman-shaped wavelet tree, with gaps (missing nodes) occurring only on the right.

(c) Listing of codes used for the wavelet trees.

Character	␣	a	e	l	r	t	v	w
Binary code (3 bits)	000	001	010	011	100	101	110	111
Inverse canonical Huffman code	0001	0000	11	100	011	101	010	001

Figure 2: Wavelet trees for example text $T =$ `wavelet␣tree` over alphabet $\Sigma = \{␣, \mathtt{a}, \mathtt{e}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{v}, \mathtt{w}\}$ ($\sigma = 8$). The texts are not stored and shown merely for orientation. A character's code can be restored by following the corresponding root-to-leaf path. The levelwise representations (bit vectors B) are obtained by concatenating the bits on each level from left to right along the dotted lines.

Wavelet Trees. The *wavelet tree* [10] is a representation of T that also serves as a rank and select index on T. It is a binary tree defined recursively as follows. The root v on level $\ell = 0$ consists of the bit vector $B_v \in \{0,1\}^{|T|}$ that contains the $(\ell+1)$-most significant bit of each character in T, that is, $B_v[i] := \mathrm{enc}(T[i])[\ell]$ for every $i \in [0, |T|)$ and a bit encoding $\mathrm{enc} : \Sigma \to \{0,1\}^*$ of the characters of Σ. Let T_0 be the subsequence of T constructed by filtering only the characters at positions i with $B_v[i] = 0$. The left subtree of v (on level $\ell+1$) is defined recursively for T_0, or empty if T_0 contains only a single distinct character. Analogously, we define T_1 and the right subtree of v.

Let $a \in \Sigma$. By construction, the length-ℓ prefix of $\mathrm{enc}(a)$ encodes the path from the root to the node representing a on level ℓ (a 0- or 1-bit for a left or right edge, respectively). We only store the bit vectors B_v at each node. Since $\Sigma = [0, \sigma)$, we can encode each character using its $\lceil \lg \sigma \rceil$-bit binary code. The wavelet tree then has height $\lceil \lg \sigma \rceil$. On every level, the number of stored bits sums up to n, such that the wavelet tree stores $n\lceil \lg \sigma \rceil$ bits in total. Fig. 2a shows an example.

In the *levelwise* wavelet tree, we concatenate the bit vectors on each level $\ell \in [0, \lceil \lg \sigma \rceil)$ to form a consecutive bit vector B_ℓ of exactly n bits. The tree's structure is retained implicitly, and we can save $\mathcal{O}(\sigma)$ words of memory that we would require for pointers between nodes and their children (at the cost of a negligible time and space overhead for answering queries, which we do not consider here). The *borders* array K_ℓ for level ℓ is an important tool for constructing directly the levelwise wavelet tree (used, e.g., in [4]). It holds the positions in B_ℓ of the first bits labeling the 2^ℓ nodes on that level. As an example, the borders array for level 2 in Fig. 2a is $K_2 = [0, 2, 7, 10]$.

The wavelet tree can be constructed in time $\mathcal{O}\left(n \lg \sigma / \sqrt{\lg n}\right)$ [1, 14] through use of

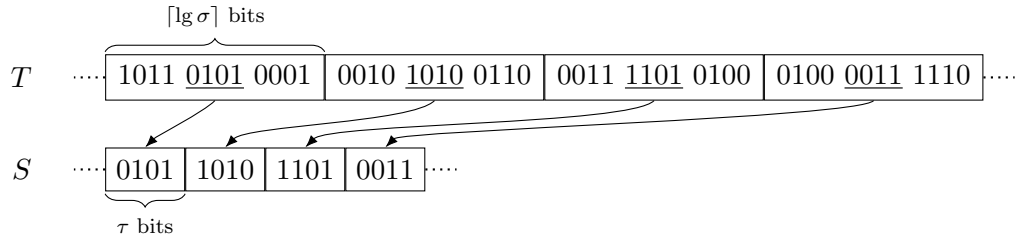


Figure 3: Extracting τ-bit blocks (underlined) from T to S for $\alpha = 1$ and $\tau = 4$.

packed lists. In practice, this can be implemented efficiently by making use of vector instructions [13]. We give a detailed description of our approach in Sect. 3.

Huffman-shaped Wavelet Trees. When using Huffman codes [11] to encode characters, the wavelet tree assumes the shape of the Huffman tree for T. The number of total bits stored is then $n\lceil\mathcal{H}_0(T)\rceil$, with $\mathcal{H}_0(T) = \sum_{\alpha\in\Sigma} \frac{n_\alpha}{n} \lg \frac{n}{n_\alpha}$ denoting the zeroth-order entropy of T. The depth of a subtree varies depending on the code lengths, such that there may be gaps (empty nodes) in the wavelet tree. As a result, bit vectors are no longer consecutive in the levelwise representation, which complicates navigation in it. We can use canonical Huffman codes [15] to establish that root-to-leaf paths are arranged in non-decreasing order of their lengths from left to right. By inverting these codes, we force gaps to occur only on the right, such that the bit vectors in the levelwise representation remain consecutive. Fig. 2b shows an example.

3 Levelwise Wavelet Tree Construction

In this section, we describe our algorithm for constructing the levelwise wavelet tree of T based on the $\lceil\lg\sigma\rceil$-bit binary encoding of characters. Our strategy follows that of [1, 14], aiming at a total construction time of $\mathcal{O}\left(n\lg\sigma/\sqrt{\lg n}\right)$. We partition the wavelet tree into $\lceil\lceil\lg\sigma\rceil/\tau\rceil$ *clusters* of $\tau \in [1, \lceil\lg\sigma\rceil]$ consecutive levels. To that end, cluster α covers levels $\alpha\tau$ to $\alpha\tau + \tau - 1$ of the wavelet tree, for $\alpha \in [0, \lceil\lg\sigma\rceil/\tau)$. For simplicity, we assume that τ divides $\lceil\lg\sigma\rceil$.

The general idea is to construct, for each cluster α, the τ corresponding bit vectors $B_{\alpha\tau}$ to $B_{\alpha\tau+\tau-1}$ of the levelwise wavelet tree in time $\mathcal{O}(n)$. Our total construction time is then $\mathcal{O}(n\lg\sigma/\tau)$, and choosing $\tau := \sqrt{\lg n}$ meets our target time. To achieve this, we use packed lists and vector operations. We first extract from each character of T the τ bits relevant for the cluster's levels in time $\mathcal{O}(n)$ and store them in a packed list S of $n\tau/w$ words. In τ passes, we then extract from S the relevant bits for the τ bit vectors. Because S is packed, each pass takes time $\mathcal{O}(n\tau/w)$, and thus we require time $\mathcal{O}(n\tau^2/w) = \mathcal{O}(n)$ for all $\tau = \sqrt{\lg n}$ passes. The greatest challenge lies in efficiently maintaining S in order such that the relevant bits can be extracted in one scan per pass. Further, in order to process the next cluster in a similar fashion, we need to account for the reordering of the characters of T that occurs by descending τ levels. In the following, we look at each step in more detail and consider suitable vector instructions.

Packed List Extraction. At the beginning of processing cluster $\alpha \in [0, \lceil \lg \sigma \rceil / \tau)$, we compute S by extracting the block of τ bits from $\alpha\tau$ to $\alpha\tau + \tau - 1$ from each character of T as depicted in Fig. 3. This can be done in one scan taking time $\mathcal{O}(n)$.

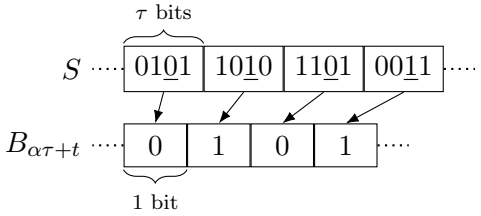
Bit Extraction. We now process S in τ passes assuming the following invariant: at the beginning of the t-th pass ($t \in [0, \tau - 1]$) of processing the α-th cluster, the τ-bit blocks in S are arranged so that $B_{\alpha\tau+t}$ of the levelwise wavelet tree can be computed in one scan, extracting the t-most significant bit from each block in S. An example is shown in Fig. 4a. Let $t \in [0, \tau - 1]$ be the current pass. Using the `pext` operation, with a selection mask where only the t-most significant bit is set per block of τ bits, we can extract w/τ bits at a time, taking claimed time $\mathcal{O}(n\tau/w)$. In practice, `pext` is limited to 64-bit words. For larger words, we set $\tau := 8$ and use `vpshufbit` with selection mask $\vec{c}_i = (t, 2t, \ldots, 8t)$ for every $i \in [1, w/64]$ to get the same result.

Borders. We need to rearrange S to maintain the invariant for computing the next level $\alpha\tau + t + 1$ in the next pass. For this, we need the borders array $K_{\alpha\tau+t+1}$. We can compute on the fly by counting the number of 0-bits written to $B_{\alpha\tau+t}$ within each range defined by the current borders array $K_{\alpha\tau+t}$ (initially, $K_0 = [0]$). To stay within our time bound, we use the population count instruction to count w/τ bits simultaneously.

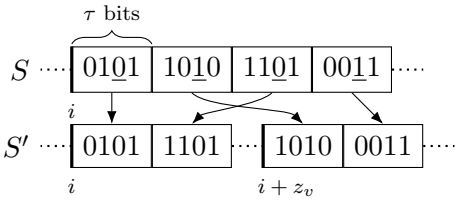
List Splitting. Consider the v-th node on level $\alpha\tau + t$ of the wavelet tree and let $B_{\alpha\tau+t}[i..j]$ (with $0 \le i \le j < n$) be the range of bits that label v. We can read $i = K_{\alpha\tau+t}[v]$ from the borders array. By construction, the range $[i, j]$ also labels the two children of v on the next level. In particular, if $z_v = \mathrm{rank}_0(B_{\alpha\tau+t}[i..j])$, then the left child of v is labeled by $B_{\alpha\tau+t+1}[i..i + z_v - 1]$ and the right child by $B_{\alpha\tau+t+1}[i + z_v..j]$. Observe how $z_v = K_{\alpha\tau+t+1}[2v + 1] - K_{\alpha\tau+t+1}[2v]$ can be computed directly from the next level's borders array that we already have available. To maintain the invariant stated earlier, we stably sort the blocks of S into the two child ranges according to their t-th bits. We refer to this process as *list splitting*. Fig. 4b shows an example.

Kaneta [13] considered using `pshufb` and `pext` for list splitting. The use of `pshufb` requires, for efficiently computing the corresponding selection vectors, a lookup table of size $2^{w/\tau}$, which is infeasible for large words of up to 512 bits. Furthermore, `pext` is limited to 64-bit words. For large words and $\tau := 8$, we use `vpcompress` with a side benefit: the selection mask for the right child is given directly the bits written to $B_{\alpha\tau+t}$ and does not need to be computed separately. For the left child, it is simply inverted.

Using these vector operations, we split w/τ blocks simultaneously. Each operation needs to be executed twice: once for the left child of v and a second time for the right. This takes time $\mathcal{O}(n\tau/w)$. However, it is crucial to see that we process ranges pertaining to *nodes* of the wavelet tree. The number of blocks labeling a node may not be a multiple of w, and if this is the case, we have to process one additional incomplete block (in constant time). Since there are most σ nodes on any level of the wavelet tree, this requires additional time $\mathcal{O}(\sigma)$. The total time required for a list splitting is thus $\mathcal{O}(\sigma + n\tau/w) = \mathcal{O}(n\tau/w)$ [1, 13].



(a) Bit extraction, writing the t-th bit (underlined) from each block of S to $B_{\alpha\tau+t}$.



(b) List splitting, stably sorting the blocks of S according to their t-th bits (underlined) to target positions i (0-bits) and $i + z_v$ (1-bits) in S'.

Figure 4: Bit extraction and list splitting for level $\alpha\tau + t$ with $\tau = 4$ and $t = 2$.

Text Reshuffling. In preparation to process the next cluster, to maintain our invariant, we need to rearrange the characters of T to account for descending τ levels in the wavelet tree. Recall how the alphabet at a node v on level ℓ is defined by the encoding path(v). Recursing into the two children of v by appending a bit to path(v) corresponds to stably sorting the characters of T by the ℓ-th bit in their binary encoding (list splitting). When descending τ levels at once, this corresponds to stably sorting the characters according to the relevant blocks of τ bits. In our case, advancing to the next cluster means we descend from level $\alpha\tau$ to $\alpha\tau + \tau$, and thus we stably sort T according to bits $\alpha\tau$ to $\alpha\tau + \tau - 1$ of each character.

This is similar to the list splitting step, with the difference that we now split into 2^τ child ranges instead than just two. The boundaries are given by the borders array $K_{\alpha\tau+\tau}$ that we already computed. Further, the τ bits from $\alpha\tau$ to $\alpha\tau + \tau - 1$ of each character address its corresponding child range.

Therefore, all that is left to do is scan over T and move each character to their bucket by extracting the relevant bits in constant time each using word operations. This is mostly similar to the packed list extraction and can be done in time $\mathcal{O}(n)$.

Working Space. Our algorithm requires $n\lceil \lg \sigma \rceil$ bits of working space for the text T, $n\tau$ bits for the packed list S and $\sigma\lceil \lg n \rceil$ bits for the borders array $K_{\alpha\tau+t}$. For each, we need two buffers: one for reading the current state and the other for writing the next. In total, we require $2n(\lceil \lg \sigma \rceil + \tau) + 2\sigma\lceil \lg n \rceil$ bits of memory.

4 Huffman-Shaped Wavelet Tree Construction

We adapt our algorithm to construct the levelwise Huffman-shaped wavelet tree. In a preliminary step, we process T and compute the inverse canonical Huffman codes (see Sect. 2) for all characters of Σ. Since the code lengths are bounded by $\log_\phi n$ bits (with $\phi = (1 + \sqrt{5})/2$ the golden ratio) [2], we can afford storing a table of size $\sigma(\lceil \log_\phi n \rceil + \lceil \lg \log_\phi n \rceil) = \mathcal{O}(\sigma \lg n)$ bits that maps characters to their codes.

The procedure is now vastly analogous to Sect. 3, extracting τ-bit blocks from the Huffman codes of characters for processing a cluster. A complication arises from the

fact that when computing level $\alpha\tau + t$ of the wavelet tree, some codes may already have ended. Using inverse canonical Huffman codes already establishes that this can only occur on the right side of the tree. What remains is to *filter* from S codes that are of length at least $\alpha\tau + t + 1$ bits, and discard codes ending after $\alpha\tau + t$ bits. We do this just before the list splitting step.

Because S only contains τ-bit blocks of the characters' codes, neither the full codes nor their lengths are known. We therefore introduce an additional packed list L of $n\tau$ bits, storing at $L[i]$ the *remaining* length in the current cluster of the Huffman code corresponding to $S[i]$ (for all $i \in [0, n)$). We reduce each length by one, such that length 0 means one remaining bit. To ensure that τ bits suffice to store a length, we also limit them to τ: we only do τ passes in the current cluster and do not care about how much they extend beyond. Each length in L thus requires $\lceil \lg(\tau + 1) \rceil \leq \tau$ bits.

By construction of L, the code of a character ends at level $\alpha\tau + t$ if its remaining length equals t. With this, we can do a parallel less-or-equal comparison of w/τ lengths against t before performing the list splitting on S. This can be done using operations similar to those used in fusion nodes, or using the parallel compare instruction if τ is a multiple of 8 (we refer to [5] for details). The result is a bitmask indicating values in L that are $\leq t$, which can be used to filter from S using `pext` or `vperm` as fit.

At the beginning of a cluster, we can construct L from T with negligible overhead during the extraction of S using our code table. For subsequent levels, we split the entries of L according to the t-th bit in the code fraction in the same way that we split S. When reshuffling T for preparing the processing of the next cluster, we discard characters whose code lengths ended in the current cluster.

5 Experiments

We implemented our algorithm in C++ and provide the source code under an open source license at `https://github.com/jptrn/mawt`. We conduct an experimental evaluation on a system with an Intel Core i9-11900KF CPU, running at 3.5 GHz (turbo disabled) and featuring a majority of the AVX-512 instruction set for 512-bit words, including those mentioned in Sect. 2. The available RAM is 128 GiB; the size of the L1 data cache is 384 KiB. Our evaluation includes the following algorithms:

- `ext`: Using `pext` for bit extraction and list splitting. We set $\tau := 4$ and fix the word size to 64 bits, which `pext` is limited to.
- `shuf` w: We fix $\tau := 8$ and vary the word size w between 64, 128, 256 and 512 bits. For bit extraction, we use `pext` if $w = 64$ and switch to `vpshufbit` otherwise. For list splitting (and filtering for Huffman-shaped wavelet trees), we use `pshufb` for $w \leq 128$, and switch to `vpcompress` for larger words.
- `lut`: Baseline using precomputed lookup tables rather than vector instructions as originally proposed by [1, 14]. We set $\tau = 4$, such that the tables are reasonably small to remain in the CPU's cache at all times, and use 64-bit words.
- `pc` and `pc-ss`: The prefix counting algorithm due to [4] and its single scan variant, the fastest purely sequential practical algorithms known to us.

Table 1: The input file corpus used for experiments, with n the number of characters, σ the number of distinct characters and $\mathcal{H}_0$ the zeroth-order entropy.

File	Description	Source	n	σ	$\mathcal{H}_0$
dblp.xml	Structured text (XML)	[8]	282.42 Mi	97	5.26
dna	DNA sequences	[8]	385.22 Mi	16	1.98
english	English texts	[8]	2.06 Gi	239	4.53
pitches	MIDI pitch values	[8]	53.25 Mi	133	5.63
proteins	Protein sequences	[8]	1.10 Gi	27	4.21
sources	Source program code	[8]	210.10 Mi	230	5.46
cc.16gib	Web crawl (text only)	[4]	16.00 Gi	243	6.20
dna.16gib	Raw DNA sequences	[4]	16.00 Gi	4	2.00
wiki.16gib	Wikipedia dump (text only)	[4]	16.00 Gi	212	5.38
ru.8gib	Word-based text	[4]	2.00 Gi	28,760,289	14.49

Table 2: Throughputs (MiBit/s) of the binary wavelet tree construction experiment. Underlined throughputs mark the fastest for the respective input file.

File	`lut`	`ext`	`shuf64`	`shuf128`	`shuf256`	`shuf512`	`pc`	`pc-ss`
dblp.xml	433.44	722.21	614.24	834.92	1,197.80	1,477.77	608.43	752.48
dna	529.32	883.00	563.11	668.93	862.49	1,011.45	594.02	745.68
english	456.91	770.55	677.96	906.42	1,304.80	1,642.69	623.08	704.90
pitches	448.02	749.24	686.88	886.62	1,276.36	1,584.19	578.70	328.47
proteins	375.73	575.99	565.63	707.23	985.35	1,178.02	633.58	761.41
sources	451.24	757.75	650.22	882.45	1,296.80	1,632.85	594.22	754.72
cc.16gib	453.97	729.58	653.25	875.61	1,265.27	1,604.84	628.46	752.97
dna.16gib	436.89	644.08	483.45	451.33	537.36	593.96	669.70	650.33
wiki.16gib	447.95	714.42	634.91	871.14	1,267.69	1,604.39	591.01	753.05
ru.8gib	317.20	642.51	506.04	660.23	938.68	1,121.03	346.96	170.44

We construct the wavelet trees for input files used in recent related work [4, 13], listed in Table 1, and measure the median throughput over five iterations. Similar to [4], the throughput is defined as the number of *output* bits produced per time unit to account for the different effective alphabet sizes of the different inputs.

Table 2 shows our results for constructing the binary wavelet tree. Our implementations `shuf256` and `shuf512` dominate even the fastest known sequential wavelet tree construction algorithms. In particular, `shuf512` is at least 35% (dna) and up to 2.7 times (pitches) as fast as the `pc` variants due to [4]. For a larger alphabet (ru.8gib), being 3.2 as fast as `pc`, we produce the best result in comparison. Generally, doubling the word width for the `shuf` implementation increases its throughput. The most remarkable increase occurs going from 128-bit to 256-bit registers, which increases the throughput by over 40% on many inputs. When fixing words to 64 bits, our implementation `ext` is faster than `shuf64` and competitive with `pc` and `pc-ss` on most inputs. Our implementation using lookup tables (`lut`) is never competitive. We note that for small alphabets ($\lceil \lg \sigma \rceil \leq 8$), our `shuf` variants only process a single cluster since $\tau = 8$. When the input is large but the alphabet very small (dna.16gib), the

Table 3: Throughputs (MiBit/s) of the Huffman-shaped wavelet tree construction experiment. Underlined throughputs mark the fastest for the respective input file.

File	`ext`	`shuf64`	`shuf128`	`shuf256`	`shuf512`	`pc`	`pc-ss`
dblp.xml	293.12	224.92	328.22	405.36	<u>467.22</u>	140.49	174.34
dna	223.31	156.60	192.67	222.97	<u>243.47</u>	152.19	145.95
english	236.20	214.51	295.57	370.08	<u>428.04</u>	132.16	129.48
pitches	308.94	237.26	372.00	488.96	<u>584.30</u>	157.42	173.36
proteins	237.44	209.68	300.20	379.16	<u>435.13</u>	158.95	169.76
sources	283.54	222.68	330.00	418.48	<u>486.50</u>	149.74	158.28
cc.16gib	287.72	210.78	301.71	370.68	<u>427.44</u>	148.24	166.16
dna.16gib	244.32	172.70	211.35	236.72	<u>258.73</u>	182.21	177.10
wiki.16gib	251.96	207.71	291.73	359.70	<u>414.72</u>	144.61	158.65

vectoring approach does not appear to have benefits over `pc`.

The performance gain of our algorithm comes at a memory trade-off: our implementations use roughly 4.5 times as much memory as `pc`. An improvement can be achieved by recycling the memory allocated for $T_{\alpha+1}$ for the packed lists $S_{\alpha\tau+t}$ while processing a cluster. When $\lceil \lg \sigma \rceil \geq 2\tau$, no additional memory needs to be allocated for $S_{\alpha\tau+t}$ at all. This reduces the space usage to 2.5 times that of `pc`.

The implementation of Kaneta [13] is closed source and not available. We expect our implementations `ext` and `shuf64` to be analogous to theirs. To get an idea about how we compare against them, we use that they also compare against `pc` for the Pizza & Chili corpus. By rule of three on their reported results for `pc`, we project their throughputs to get an estimate for our setup. By this projection, on average, our implementation of `ext` is 10% faster than theirs, while our `shuf64` is 3% slower. We therefore conclude that our implementations `ext` and `shuf64` are indeed analogous to Kaneta's. By supporting wider registers of up to 512 bits in `shuf512`, however, we achieve speedups of 2 to 2.5 compared to their results.

Table 3 shows our results for constructing Huffman-shaped wavelet trees. Here, we did not consider lookup tables, because we would need a different lookup table per cluster, and we did not expect it to be competitive given the results of the binary wavelet tree experiments. The computation of Huffman codes is included in the measured throughputs and based on the same (open source) code due to [4]. Our implementation `shuf512` outperforms `pc` in all instances, finishing 1.4 (dna.16gib) to 2.6 (wiki.16gib) times as fast. None of the algorithms finished within an acceptable time frame for the large alphabet input (ru.8gib), taking too long to compute its Huffman tree.

Further Extensions. Our algorithm can easily be modified to construct instead the (binary or Huffman-shaped) wavelet matrix. The experimental results showed no notable differences to our wavelet tree results and are therefore omitted for brevity.

Finally, we experimented with parallelizing the algorithm using domain decomposition, which gave the best results for [4]. However, this did not scale to our satisfaction; using 16 threads only resulted in a speedup of 6. This can be explained by the fact that Intel's Rocket Lake CPUs (which we are using) downclock when using AVX-512

registers in multicore situations [6]. Given this behaviour, a scalable shared-memory implementation using ultrawise registers seems infeasible.

Acknowledgements. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500), as well as from the Deutsche Forschungsgemeinschaft (DFG) under the Research Grants programme (project No. 501086801).

References

[1] Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *SODA*, pages 572–591. SIAM, 2015.

[2] Michael Buro. On the maximum length of huffman codes. *Inform. Process. Lett.*, 45(5):219–223, 1993.

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

[4] Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Practical wavelet tree construction. *ACM J. Exp. Algorithmics*, 26:1.8:1–1.8:67, 2021.

[5] Patrick Dinklage, Johannes Fischer, and Alexander Herlez. Engineering predecessor data structures for dynamic integer sets. In *SEA*, pages 7:1–7:19. Dagstuhl, 2021.

[6] Travis Downs. Ice Lake AVX-512 downclocking. `https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html`, accessed Oct 2022.

[7] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE, 2000.

[8] Paolo Ferragina and Gonzalo Navarro. Pizza & chili corpus – compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl/texts.html`, accessed Oct 2022.

[9] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[10] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. SIAM, 2003.

[11] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[12] Intel Corporation. *Intel (R) 64 and IA-32 Architectures – Software Developer's Manual – Volume 2: Instruction Set Reference, A-Z*, September 2016.

[13] Yusaku Kaneta. Fast wavelet tree construction in practice. In *SPIRE*, pages 218–232. Springer, 2018.

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

[15] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Commun. ACM*, 7(3):166–169, 1964.