Journal of Discrete Algorithms ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Parallel lightweight wavelet tree, suffix array and FM-index construction [☆]

Julian Labeit [a,*], Julian Shun [b], Guy E. Blelloch [c]

[a] *Karlsruhe Institute of Technology, Department of Informatics, Am Fasanengarten 5, 76131 Karlsruhe, Germany*
[b] *University of California, Berkeley, CA 94720, United States*
[c] *Computer Science Department, Carnegie Mellon University, PA 15213-3890, Pittsburgh, United States*

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Wavelet tree
Suffix array
FM index
Parallel
Induced sorting
Lightweight

ABSTRACT

We present parallel lightweight algorithms to construct wavelet trees, rank and select structures, and suffix arrays in a shared-memory setting. The work and depth of our first parallel wavelet tree algorithm match those of the best existing parallel algorithm while requiring asymptotically less memory and our second algorithm achieves the same asymptotic bounds for small alphabet sizes. Our experiments show that they are both faster and more memory-efficient than existing parallel algorithms. We also present an experimental evaluation of the parallel construction of rank and select structures, which are used in wavelet trees. Next, we design the first parallel suffix array algorithm based on induced copying. Our induced copying requires linear work and polylogarithmic depth for constant alphabets sizes. When combined with a parallel prefix doubling algorithm, it is more efficient in practice both in terms of running time and memory usage compared to existing parallel implementations. As an application, we combine our algorithms to build the FM-index in parallel.

1. Introduction

In recent years, compressed full-text indexes [32] have become popular as they provide an elegant way of compressing data while at the same time supporting queries on the compressed data efficiently. The most popular indexes all rely on three basic concepts: succinct rank and select on bit-vectors, wavelet trees, and suffix arrays. Modern applications need algorithms for constructing these data structures that are fast, scalable, and memory-efficient. The Succinct Data Structure Library (SDSL) [13] is a state-of-the-art library for constructing these data structures sequentially. Additionally, in recent years wavelet tree construction [10,37] and linear-work suffix array construction [19] have been successfully parallelized.

☆ We acknowledge support from the Miller Institute for Basic Research in Science at UC Berkeley and from the National Science Foundation under grant number CCF-1314590. Additionally we acknowledge the support of the Baden-Württemberg Stiftung and interACT making the student exchange of Julian Labeit possible.
* Corresponding author.
E-mail addresses: julianlabeit@gmail.com (J. Labeit), jshun@eecs.berkeley.edu (J. Shun), guyb@cs.cmu.edu (G.E. Blelloch).

http://dx.doi.org/10.1016/j.jda.2017.04.001
1570-8667/© 2017 Published by Elsevier B.V.

However, so far most parallel implementations are not memory-efficient. The goal of this work is to develop parallel algorithms for constructing these data structures that are memory-efficient while at the same time being fast and scalable.

For wavelet tree construction, we reduce the space usage of the algorithm by Shun [37] from $O(n \log n)$ to $n \log \sigma + o(n)$ bits of additional space beyond the input and output for an input size n and alphabet size σ. Our algorithm requires $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth. Our experiments on 32 cores show that our modified algorithm achieves a speedup of up to 7x over the original algorithm of [37], and achieves a speedup of up to 28x over the fastest sequential algorithm. Additionally, we propose a variation of the domain-decomposition algorithm by Fuentes et al. [10], which requires the same work, depth and space as our first algorithm when $\sigma / \log \sigma \in O(\log n)$. We also present an experimental evaluation of the parallel construction of rank and select structures, which are used in wavelet trees, and show self-relative speedups of 10–38x on 32 cores.

For suffix array construction there have been three main classes of algorithms described in the literature [36]: prefix doubling, recursive, and induced copying (sorting) algorithms. While prefix doubling [24] and recursive algorithms [19] have been parallelized in the past, the sequential algorithms that are the fastest and most memory-efficient in practice all use induced copying. Induced copying algorithms are hard to parallelize because they use sequential loops with non-trivial data dependences. In this work, we develop a parallel algorithm using induced copying. We first use parallel rank and select on bit-vectors to develop a more memory-efficient version of the parallel implementation of prefix doubling from the Problem Based Benchmark Suite (PBBS) [39]. Then we show how to parallelize an iteration of induced copying for constant-sized alphabets in polylogarithmic depth. Finally we combine both techniques to generate a parallel version of a two-stage algorithm introduced in [17]. Our experiments show that our algorithm uses only slightly more space than the most memory-efficient sequential algorithm, and among parallel algorithms it is the most memory-efficient and usually the fastest. On 32 cores, our algorithm is up to 1.7x faster and uses 1.8x less memory than the fastest existing parallel algorithm, and achieves a speedup of 5–7x over the fastest sequential algorithm for non-repetitive inputs.

Finally, we use our algorithms to construct FM-indexes [7], a compressed full-text index, in parallel and integrate our implementations into the SDSL. Using the algorithms as part of the SDSL for FM-index construction, on 32 cores we achieve self-relative speedups of up to 18x and absolute speedups of up to 6x over the sequential SDSL implementation on a variety of real-world inputs.

2. Preliminaries

For cost analysis we use the *work-depth* model, where *work* W is the number of total operations required and *depth* D is the number of time steps required. Using Brent's scheduling theorem [18] we can bound the running time by $O(W/P + D)$ using P processors on a PRAM. We allow for concurrent reading from shared memory locations but no concurrent writing.

We make use of the parallel primitives prefix sum, filter, and split. *Prefix sum* takes an array A of n elements, an associative operator $\oplus$ and an identity element $\perp$ with $\perp \oplus x = x$ for all x, and returns the array $\{\perp, A[0], A[0] \oplus A[1], \ldots, A[0] \oplus A[1] \oplus \ldots \oplus A[n-2]\}$ as well as the overall sum $A[0] \oplus A[1] \oplus \ldots \oplus A[n-1]$. Prefix sum can be implemented with $O(n)$ work and $O(\log n)$ depth [18]. *Filter* and *split* both take an array A of n elements and a predicate function f with $f(A[i]) \in \{0, 1\}$. Split returns two arrays A_0 and A_1 where A_k holds all elements with $f(A[i]) = k$. Filter only returns A_1. Both filter and split preserve the relative order between the elements and can be implemented using prefix sums in $O(n)$ work and $O(\log n)$ depth [18]. By dividing the input into groups of $\log n$ elements and processing each group sequentially and in parallel across all groups, filter and split can be implemented with n bits of space in addition to the input and output.

A string S is a sequence of characters from a finite ordered set $\Sigma = [0, \ldots, \sigma - 1]$, called the *alphabet*, where $\sigma = |\Sigma|$. $|S| = n$ denotes the length, $S[i]$ denotes the i'th character (zero-based) and $S[i, \ldots, j]$ denotes the substring from the i'th to the j'th position of S (inclusive). If $j = |S| - 1$ then $S[i, \ldots, j]$ is the i'th *suffix* of S. The ordering of Σ induces a lexicographical ordering for strings. The *suffix array (SA)* of a string is an array storing the starting positions of all suffixes of S in lexicographic order. As all suffixes of a string are unique, SA is a permutation and the inverse permutation is called the *inverse suffix array (ISA)*. The *Burrows–Wheeler transform (BWT)* of a string S is the permutation BWT of S with $BWT[i] = S[SA[i] - 1]$ for $i \in \{0, \ldots, n-1\}$ where $S[-1] = S[n-1]$.

We define the three following queries on a string S: $S[i]$ accesses the i'th element, $rank_c(S, i)$ counts the occurrences of character c in $S[0, \ldots, i-1]$ and $select_c(S, i)$ calculates the position of the i'th occurrence of c in S. For bit-vectors ($\sigma = 2$), there are rank and select structures using $n + o(n)$ bits of space and supporting the queries in $O(1)$ work [12]. *Wavelet trees (WT)* generalize this to larger alphabets [14]. A WT of the string S over the alphabet $[a, \ldots, b] \subseteq [0, \ldots, \sigma - 1]$ has root node v. If $a = b$ then v is a leaf node labeled with a. Otherwise v has a bit-vector B_v where $B_v[i] = 1$ if $S[i] > (a+b)/2$ and $B_v[i] = 0$ otherwise. Let S_k be the string of all $S[i]$ with $B_v[i] = k$. The left child of v is the WT of the string S_0 over the alphabet $[a, \ldots, \lfloor (a+b)/2 \rfloor]$ and the right child is the WT of the string S_1 over the alphabet $[\lfloor (a+b)/2 \rfloor + 1, \ldots, b]$. An example WT is shown in Fig. 1. The WT can support the three queries above in $O(\log \sigma)$ work. By changing the definition of B_v, the shape of the WT can be altered from a balanced binary tree to, for example, a Huffman-shaped tree [27]. We refer the reader to [31] for numerous applications of WTs.

The *FM-index* is a compressed full-text index using the BWT of a text [7]. By supporting efficient rank and select queries on the BWT of a text S (for example, with a wavelet tree), the FM-index can efficiently count the number of occurrences of a pattern P in the text S. By additionally sampling the SA of S, the exact locations of the occurrences in S can be

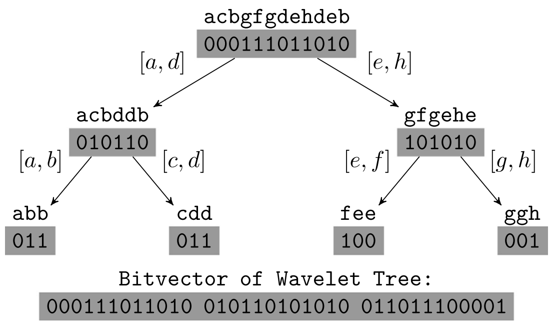


Fig. 1. Example of a balanced binary wavelet tree over the string acbgfgdehdeb.

calculated. In this work we refer to the FM-index of S as the wavelet tree over the BWT of S augmented with rank and select structures.

3. Related work

Fuentes et al. [10] describe two $O(n \log \sigma)$ work and $O(n)$ depth algorithms to construct WTs. The first uses the observation that the levels of the WT can be built independently; the second splits the input among processors, then builds WTs over each part sequentially and finally merges the WTs. Shun [37] introduces the first polylogarithmic-depth WT construction algorithms and also describes how to build the rank and select structures on the bit-vectors in parallel. The algorithm performing best in practice constructs the WT level-by-level. Each level is computed from the previous level in $O(\log n)$ depth. Recently, Ferres et al. [8] describe how to construct range min–max trees in parallel. Their structure can support rank/select queries on bit-vectors in $O(\log n)$. Recently, sequential algorithms with $O(n \log \sigma / \sqrt{\log n})$ work have been described [1,30] and have recently been parallelized [38], but none of these algorithms have been implemented.

Suffix arrays were first introduced by Manber and Myers [28] as a space-efficient alternative to suffix trees. Since then, many different suffix array construction algorithms have been developed, including the difference cover (DC3) algorithm [19] and the induced sorting algorithm (SA-IS) [33]. DC3 was one of the first linear-work suffix array algorithms, and it can be efficiently parallelized in various computational models. There are parallel implementations of DC3 available for shared memory [39], distributed memory [9,21], and GPUs [5,35,42]. SA-IS is a lightweight linear-work algorithm and one of the fastest in practice. Unfortunately, it is hard to parallelize as the SA-IS algorithm consists of multiple sequential scans with non-trivial data dependences. Recently, Kärkkäinen et al. [20] introduced a new divide and conquer algorithm *Scan*. The algorithm first constructs SAs over chunks of the input text using an existing algorithm and then merges the partial SAs in parallel.

Many bioinformatics applications use compressed SAs, and thus there have been many frameworks with parallel SA implementations optimized for DNA inputs [16,25]. For example, PASQUAL [25] has a fast implementation using a combination of prefix-doubling and string sorting algorithms. For certain applications, only the BWT is needed so there has been significant work on constructing the BWT in parallel [15,26].

4. Parallel wavelet tree construction

In this section, we develop space-efficient parallel algorithms for WT construction. In addition to the tree structure and bit-vectors per node, each node of the WT also requires a rank/select structure. We describe our parallel implementation of rank/select structure construction at the end of this section.

Recursive WT. The levelWT algorithm proposed by Shun [37] uses prefix sums over the bit-vectors of a level of the WT to calculate the bit-vectors for the next level, allocating two integer arrays each of length n. As a result the algorithm has a memory requirement of $O(n \log n)$ bits. We reduce the memory requirement by substituting the prefix sums with the parallel split operation, reducing the memory down to $n \log \sigma$ bits of additional space excluding the input and output. A further optimization is to implement the algorithm recursively instead of strictly level-by-level as done in [37]. In particular, the two children of a node are constructed via two recursive calls in parallel. This approach is more cache-friendly and avoids explicitly computing the node boundaries per level, which requires $O(\sigma \log n)$ bits of space, and instead each processor computes the boundaries when it launches the two recursive calls, requiring $O(\log n \log \sigma)$ bits of stack space per processor (one pointer for each level of the tree). We refer to this algorithm as *recursiveWT*, and the pseudocode is shown below. Note that Lines 8 and 9 can proceed in parallel, and is implemented with fork-join. This algorithm has $O(n \log \sigma)$ work and $O(\log n \log \sigma)$ depth, matching that of the original levelWT algorithm.

```
1  recursiveWT ($S$, $\Sigma = [a, b]$):
2      if $a = b$: return leaf labeled with $a$
3      $v$ := root node
```

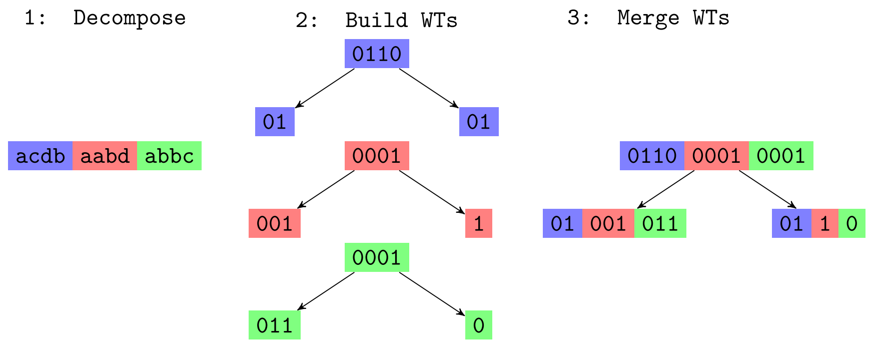


Fig. 2. Example execution of the *ddWT* algorithm with $k = 3$. Displayed is the decomposition of the input $S = acdbaabdabbc$ into chunks S_0, S_1 and S_2 (line 2), the constructed wavelet trees B_0, B_1 and B_2 (line 4) and the final result B (line 14). The bits of the wavelet trees are marked with the corresponding colors S_0 (purple), S_1 (red) and S_2 (green) of the input chunks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
4     $v.bitvector :=$ bitvector of size $|S|$
5     **parfor** $i := 0$ to $|S| - 1$:
6         $v.bitvector[i] = \begin{cases} 0, \text{if } S[i] \le (a+b)/2, \\ 1, \text{else} \end{cases}$
7     $(S_0, S_1) = \text{parallelSplit}(S, v.bitvector)$
8     pardo:
9         $v.leftChild = \text{recursiveWT}(S_0, [a, \lfloor \frac{a+b}{2} \rfloor])$
10        $v.rightChild = \text{recursiveWT}(S_1, [\lfloor \frac{a+b}{2} \rfloor + 1, b])$
11    **return** $v$
```

Domain decomposition WT. Our second algorithm for WT construction (*ddWT*) is a modified version of the domain decomposition algorithm introduced by Fuentes et al. [10]. The first part of our algorithm is the same as the original algorithm: the input string is split into k chunks, and a WT is constructed for each chunk independently in parallel. The second part of the algorithm involves merging the WTs. Fig. 2 illustrates how in general the *ddWT* algorithms works with an example. In the original algorithm of [10], the bit-vectors of each level are merged sequentially, leading to linear depth. We observe that merging the WTs essentially requires reordering the bit-vectors. Initially, the bit-vectors of the k WTs are stored consecutively in the bitmaps B_k. By calculating the prefix sum over the lengths of all bit-vectors belonging to each node in the final WT, we obtain the corresponding positions in the final bit-vector for each of the bit-vectors from the first part of the algorithm. Then the final bit-vector can be constructed by copying the input bit-vectors to it in parallel. We refer to this algorithm as *ddWT*, and the pseudocode is shown below. The first part of the algorithm requires $O(n \log \sigma)$ work and $O(n \log(\sigma)/k)$ depth, and the second part of the algorithm requires $O(k\sigma)$ work and $O(\log(k\sigma))$ depth giving an overall work of $O(k\sigma + n \log \sigma)$ and depth of $O(\log(k\sigma) + n \log(\sigma)/k)$. By choosing $k \in \Theta(n/\log n)$ the overall work is $O(\sigma n/\log n + n \log \sigma)$ and the depth is $O(\log n \log \sigma)$. If $\sigma/\log \sigma \in O(\log n)$ then our algorithm requires $O(n \log \sigma)$ work, $O(\log n \log \sigma)$ depth and uses $O(n \log \sigma)$ bits of additional space. Subsequent to the initial publication of this algorithm [22, 23], Fuentes et al. [11] published a parallel domain decomposition algorithm very similar to ours.

```
1   ddWT (S, $\Sigma$, k = $\Theta(n/\log n)$):
2       decompose S into k strings $S_i$
3       **parfor** i := 0 to k−1:
4           ($B_i$, Nodes[i]) = serialWT($S_i$)
5       offsets := array of size $2 \cdot |\Sigma| \cdot k$
6       **parfor** n := 0 to $2 \cdot |\Sigma| - 1$:
7           **parfor** i := 0 to k−1:
8               offsets[$n \cdot k + i$] = Nodes[i][n].length
9       perform prefix sum on offsets
10      **parfor** i := 0 to k−1
11          **parfor** n := 0 to $2 \cdot |\Sigma| - 1$:
12              // destination: bv and offset, source: bv and offset, number of bits to copy
13              copy(B, offsets[$n \cdot k + i$], $B_i$, Nodes[i][n].start, Nodes[i][n].length)
14      **return** $B$
```

Both recursiveWT and ddWT can easily be adapted to construct different shapes of WTs, such as Huffman-shaped WTs [27]. For recursiveWT only the case distinction in Line 6 has to be changed. For ddWT on Line 4, the sequential algorithm for the desired shape is used. As part of a parallel version of the SDSL, we provide implementations of recursiveWT for constructing various shapes of WTs. For Huffman-shaped WTs, we assume that the shape of the Huffman tree is given as input. Note that the Huffman tree can be computed in parallel using the algorithm of [6]. Changing the shape of the wavelet tree has impact on the runtime analysis. Let h be the maximum height of the wavelet tree (previously $\log \sigma$) and

```
parallelRange $(S,n)$:
    $SA$, $ISA$ := integer arrays of size $n$
    parfor $i$ := 0 to $n-1$:
        $ISA[i] = S[i]$
        $SA[i] = i$
    $ranges := \{(0, n-1)\}$
    $offset := 0$
    while $ranges$ not empty:
        $nranges := \{\}$
        parfor $(s,e)$ in $ranges$:
            sort all $i \in SA[s,\dots,e]$ by the values at $ISA[SA[i] + offset]$
        parfor $(s,e)$ in $ranges$:
            scan $SA[s,\dots,e]$ in parallel, update $ISA$ and add equal ranges to $nranges$
        $ranges = nranges$
        $offset = \max(1, 2 \cdot offset)$
    return $SA$
```

Fig. 3. Parallel range algorithm for parallel suffix-array construction.

m the total wavelet tree size in bits (previously $n \log \sigma$). For the Huffman shaped wavelet trees we can bound $h \in O(\log n)$ using the results of Buro [3] and $m \in O(nH_0(s))$. The *recursiveWT*, without the construction of the Huffman tree, then has $O(\log^2 n)$ depth and $O(nH_0(s))$ work.

In both algorithms *recursiveWT* and *ddWT* we write to the bits of shared words in parallel. These writes can be executed sequentally in $\log n$ depth. As both algorithms already need $O(\log n)$ depth per level this does not change the runtime analysis. In practice we use the compare-and-swap instruction to write to shared words in parallel.

Parallel rank. We now describe our parallel implementation of constructing rank and select structures on bit-vectors based on the ideas of [37]. The answers to the rank queries are pre-computed and stored in first and second-level lookup tables. In theory the first level dictionary stores the answer for every $\log^2 n$ query in $O(\frac{n}{\log^2 n} \log n)$ bits. The second level stores relative answers to every $\frac{\log n}{2}$ query in $O(\frac{n}{\log n} \log \log n)$ bits. With the first and second level lookup tables every $\frac{\log n}{2}$ query can be answered. For all other queries an additional relative rank has to be computed in a block of length $\frac{\log n}{2}$. These are pre-computed and stored in an additional lookup table. As there are only $2^{\frac{\log n}{2}} = \sqrt{n}$ different types of blocks this table also has sublinear space.

In practice, we chose to parallelize the broadword implementation in SDSL [40]. The rank structure uses 25% additional space for the bit-vector and has a cache-friendly memory layout. The construction can be parallelized using prefix sums.

Parallel select. For select, we parallelize the SDSL implementation of a variant of the structure by Clark [4]. Similar to the rank structure, the first level stores the answers to every $\log^2 n$ query in $O(\frac{n}{\log^2 n} \log n)$ bits. If two pre-computed query results differ by more than $\log^4 n$ all queries in-between are also precomputed and stored explicitly. We call this block of queries a *long* block, all other blocks are called *short* blocks. There can be at-most $\frac{n}{\log^4 n}$ long blocks, thus all answers of these blocks can be stored in $O(\frac{n}{\log^4 n} \log^3 n)$ bits. For the short blocks the relative answer to every $\log^2 n$ query is stored in $O(\frac{n}{\log^2 n} \log \log n)$ bits. If two pre-computed querys differ by more than $\frac{\log n}{2}$ the relative answers of the queries in between are stored explicitly. This can be done in $O(\frac{n}{\log n} \log \log n)$ bits. The relative answers to the remaining queries are part of blocks of size $\frac{\log n}{2}$. As there are only $2^{\frac{\log n}{2}} = \sqrt{n}$ different type of blocks, these answers can be stored in a look-up table. For a more detailed analysis, see [4].

Prefix sums can be used to categorize the blocks into long and short blocks. After categorizing the blocks, they can be initialized independently. As short blocks are only of polylogarithmic size, they can be initialized sequentially in polylogarithmic depth. Long blocks are also initialized with prefix sums.

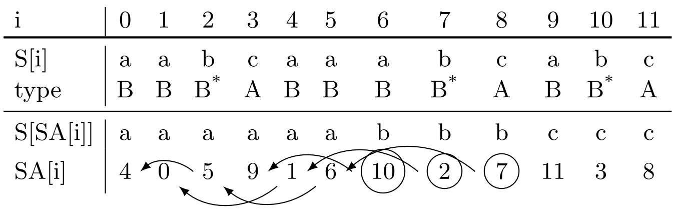
5. Parallel suffix array construction

Previous parallel SA algorithms either use the recursive [19] or prefix doubling [24] approach. However the fastest and most space-efficient sequential SA algorithms use induced copying [36], so our goal here is to parallelize such an algorithm. We first describe a simple parallel prefix doubling algorithm, which in practice needs little more than $n \log n$ bits of memory in addition to the input and output. We then introduce a parallel algorithm which uses induced copying, and uses the prefix doubling algorithm as a subroutine. The algorithm uses $n + o(n)$ additional bits of space.

Parallel range. Parallel Range algorithm (shown in Fig. 3) from the PBBS [39] is a parallel version of the suffix array construction algorithm described by Larsson and Sadakane [24]. The algorithms starts by approximating the ISA array with the

```
1  $bucketA$ := Starting positions of the $A$ buckets
2  $bucketB$ := Ending position of the $B$ buckets
3  for $i := n-1$ to 0:
4      if $SA[i]$ has been initialized and $SA[i]-1$ is a $B$−type suffix:
5          $SA[bucketB[S[SA[i]-1]]] = SA[i]-1$
6          $bucketB[S[SA[i]-1]] = bucketB[S[SA[i]-1]]-1$
7  for $i := 0$ to $n-1$:
8      if $SA[i]-1$ is an $A$−type suffix:
9          $SA[bucketA[S[SA[i]-1]]] = SA[i]-1$
10         $bucketA[S[SA[i]-1]] = bucketA[S[SA[i]-1]]+1$
```

Fig. 4. Induced sorting of all A and B-type suffixes by using the already sorted B^*-type suffixes.



i	0	1	2	3	4	5	6	7	8	9	10	11
S[i]	a	a	b	c	a	a	a	b	c	a	b	c
type	B	B	B*	A	B	B	B	B*	A	B	B*	A
S[SA[i]]	a	a	a	a	a	a	b	b	b	c	c	c
SA[i]	4	0	5	9	1	6	10	2	7	11	3	8

Fig. 5. Example of induced sorting of B suffixes using B^* suffixes. The SA entries corresponding to the B^* suffixes are circled, and the order in which SA is filled in is denoted by the arrows. Each chain of arrows corresponds to a run of B suffixes ending with a B^* suffix.

```
1  $bucketB$ := Ending position of the $B$ buckets
2  for $\alpha := \sigma-1$ to 0:
3      $[s,e]$ := interval in $SA$ of all suffixes starting with $\alpha$
4      $bucketSums$ := array of arrays to count number of suffixes put into buckets
5      parfor $i := e$ to $s$:
6          if $SA[i]$ has been initialized and $SA[i]-1$ is a $B$−type suffix:
7              $bucketSums[S[SA[i]-1]][i] = bucketSums[S[SA[i]-1]][i]+1$
8      parfor $\alpha := 0$ to $\sigma-1$:
9          perform prefix sum on $bucketSums[\alpha]$
10     parfor $i := e$ to $s$:
11         if $SA[i]$ has been initialized and $SA[i]-1$ is a $B$−type suffix:
12             $b := S[SA[i]-1]$
13             $SA[bucketB[b]-bucketSums[b][i]] = SA[i]-1$
```

Fig. 6. Parallel induced sorting of all B-type suffixes for inputs with no repetitions of characters.

```
1  $[e',e]$ := interval of $SA$ values that already are initialized
2  $\alpha$ := first character of all the suffixes in $SA[s,e]$
3  while $[e',e]$ not empty:
4      $m := |\{i \in [e',e] \mid S[SA[i]-1]=\alpha\}|$
5      $SA[e'-m-1, e'-1] = \{x \in SA[e',e] \mid S[SA[x]-1]=\alpha\}$        //using filter
6      $e := e'-1$
7      $e' := e'-m-1$
8      parfor $i := e'$ to $e$:
9          $SA[i] = SA[i]-1$
```

Fig. 7. Subroutine of parallel induced sorting of B-type suffixes for inputs with repetitions (insert after Line 3 of the pseudocode in Fig. 6).

characters of the text, here the one-to-one correspondence of the alphabet with integers smaller σ is used (Lines 3–5 in Fig. 3). A series of at most $\log n$ refinement steps are then applied, where after step d, the suffixes are sorted by their first 2^d characters and the ISA array is updated to reflect this order. Afters step d the ISA array holds the ranks of the suffixes first 2^d characters. Note that ISA array is not a valid permutation until all suffixes have been sorted correctly. More precisely on step d groups of suffixes with the same ISA value from the previous step are sorted by their first 2^d characters by accessing $ISA[SA[i]+2^{d-1}]$ for suffix i (Lines 10–11 in Fig. 3). Then the groups and ISA values are updated (Lines 12–13). Each group of SA elements that compared equal during the sorting forms a new group. The ISA values of a new group (s,e) are all set to s. Singleton elements are not stored as group (s,s), however the ISA value is set to $ISA[SA[s]]=s$. In the original implementation, two integer arrays (one for reading and one for writing) are used to keep track of groups of same ISA value, occupying an additional $2n\log n$ bits. To reduce memory consumption, we mark the boundaries of groups with a bit-vector. Using a parallel select structure allows us to iterate over all groups of the same ISA value efficiently. Thus the algorithm only needs $n\log n$ additional bits for the ISA array, $2n+o(n)$ bits for two bit-vectors with select structures, and the space for

Table 1
Collection statistics: number of characters, alphabet size and average LCP value.

Input	n	σ	Average LCP
non-repetitive real inputs			
sources	210,866,603	229	371.8
pitches	55,814,376	132	262.4
proteins	1,184,051,855	27	1,421.6
dna	403,927,746	16	2,420.7
english.1024MB	1,073,741,816	236	36,886.5
dblp.xml	296,135,874	97	44.9
repetitive real inputs			
Escherichia_Coli	112,689,515	15	11,322.4
cere	461,286,644	5	7,080.1
coreutils	205,281,760	235	149,625.9
einstein.de.txt	92,758,441	117	35,248.1
influenza	154,808,555	15	774.7
kernel	257,961,544	159	173,308.1
para	429,265,758	5	3,275.2
world_leaders	46,968,181	89	8,837.2
fib41	267,914,296	2	70,711,161.3
rs.13	216,747,218	2	44,038,468.6
tm29	268,435,456	2	32,156,331.2
dblp.xml.00001.1	104,857,600	89	94,981.5
dblp.xml.00001.2	104,857,600	89	95,781.4
dblp.xml.0001.1	104,857,600	89	9,844.8
dblp.xml.0001.2	104,857,600	89	9,865.6
dna.001.1	104,857,600	5	993.0
english.001.2	104,857,600	106	987.3
proteins.001.1	104,857,600	21	991.4
sources.001.2	104,857,600	98	992.2
artificial inputs			
aaa	104,857,600	1	52,428,799.5
abab	104,857,600	2	52,428,798.5
aabbaabb	104,857,600	2	51,388,088.8
rnd-8	104,857,600	2^8	
rnd-12	104,857,600	2^{12}	
rnd-16	104,857,600	2^{16}	
rnd-20	104,857,600	2^{20}	

the sorting routine. Using a parallel integer sorting algorithm with $O(n)$ work and $O(n^\epsilon)$ depth for $0 < \epsilon < 1$ [41], Parallel Range has $O(n \log n)$ work and $O(n^\epsilon \log n)$ depth.

Parallel DivSufSort. Using Parallel Range, we can parallelize the *DivSufSort* implementation [29] by Mori of the two-stage algorithm [17].

In the first step the suffixes are categorized into A, B, and B^* suffixes. A suffix $S[i, \dots, n-1]$ is of type A if $S[i+1, \dots, n-1] < S[i, \dots, n-1]$ and of type B otherwise. B^* suffixes are all B-type suffixes that are followed by an A-type suffix. This step can be parallelized using two parallel loops and prefix sums in $O(n)$ work and $O(\log n)$ depth. In the first parallel for loop the boundaries of continuous ranges of either category type are computed. With the prefix sums these positions are propagated across the suffix array. With the second parallel loop the actual categorization takes place.

The second step lexicographically sorts all of the B^* substrings. B^* substrings are all substrings formed by the characters between two consecutive B^* suffixes. Then each B^* substring can be replaced by its rank among the B^* substrings, forming a reduced text. Note that there are very efficient parallel string sorting algorithms available [2]. Our implementation, however, only parallelizes an initial bucket sort and uses a sequential multikey quicksort for the resulting buckets.

The third step constructs the SA of the reduced text. As the text size has been reduced by at least half, since there can be at most $n/2$ B^* suffixes, the unused part of the SA can be used for the ISA. Thus Parallel Range can be applied with only $n + o(n)$ bits additional space, plus the space needed for the sorting routine.

In the final step, the sorted order of the B^* suffixes is used to induce the sorting of the remaining suffixes. We describe induced sorting next, and introduce a linear-work polylogarithmic-depth algorithm for induced sorting on constant-sized alphabets.

Induced sorting. The sequential algorithm for induced sorting (shown in Fig. 4) consists of two sequential loops, one sorting the B suffixes and one sorting the A suffixes. At the beginning, the SA entries corresponding to B^* suffixes are initialized. The order in which the SA is traversed is crucial to guarantee a correct sorting. B-type suffixes are defined such that if a B-type suffix is directly preceded (in text order) by another B-type suffix, then the preceding suffix has to be lexicographically smaller. Fig. 5 shows an example of how induced sorting inserts B-type suffixes into the SA. The arrows indicate the

Table 2
Running times (seconds) sequential, parallel and self-relative speedup of WT construction algorithms on 32 cores. The fastest parallel running times are marked in bold.

Input	levelWT			recWT			ddWT			serWT	sdslWT
	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_1
non-repetitive real inputs											
sources	8.97	1.02	8.76	4.75	**0.15**	31.87	6.78	0.28	24.54	3.88	5.80
pitches	2.63	0.27	9.56	1.70	**0.05**	33.35	2.22	0.09	24.13	1.26	1.96
proteins	36.28	3.32	10.92	26.96	**0.72**	37.42	33.93	1.12	30.32	18.77	36.60
dna	9.49	0.85	11.11	7.43	**0.20**	36.44	8.97	0.33	26.92	4.95	12.50
english.1024MB	47.54	5.30	8.97	26.73	**0.83**	32.24	53.55	1.46	36.75	21.61	33.40
dblp.xml	10.84	1.23	8.83	5.34	**0.17**	31.49	7.92	0.33	24.20	4.61	6.91
repetitive real inputs											
Escherichia_Coli	2.12	0.23	9.02	1.19	**0.04**	32.00	1.65	0.09	18.38	0.96	2.46
cere	5.52	0.65	8.53	2.69	**0.09**	28.84	4.11	0.21	19.65	2.34	8.84
coreutils	8.12	0.99	8.17	3.60	**0.12**	28.84	5.57	0.24	23.40	3.32	4.11
einstein.de.txt	3.12	0.39	8.08	1.35	**0.05**	27.88	2.12	0.10	20.41	1.26	1.76
influenza	2.68	0.32	8.30	1.21	**0.04**	27.91	1.82	0.12	14.75	1.07	2.97
kernel	10.06	1.25	8.07	4.34	**0.15**	28.25	6.57	0.30	21.75	3.94	5.01
para	5.17	0.60	8.59	2.57	**0.09**	29.19	3.84	0.19	19.82	2.21	8.33
world_leaders	1.57	0.20	8.01	0.69	**0.03**	27.18	1.06	0.05	19.63	0.63	0.91
fib41	0.34	0.01	30.06	0.34	**0.01**	31.04	0.62	0.03	18.88	0.32	4.99
rs.13	0.31	0.01	31.47	0.30	**0.01**	32.79	0.56	0.03	20.20	0.29	4.04
tm29	0.39	0.01	32.36	0.38	**0.01**	33.86	0.70	0.05	15.40	0.36	4.99
dblp.xml.00001.1	3.60	0.43	8.33	1.56	**0.05**	28.79	2.49	0.12	20.14	1.48	1.99
dblp.xml.00001.2	3.60	0.43	8.30	1.57	**0.06**	28.45	2.49	0.12	20.60	1.48	1.99
dblp.xml.0001.1	4.18	0.44	9.61	1.57	**0.05**	28.69	2.49	0.12	20.34	1.48	1.99
dblp.xml.0001.2	3.60	0.44	8.27	1.57	**0.05**	28.65	2.49	0.12	20.27	1.48	2.00
dna.001.1	1.25	0.15	8.51	0.61	**0.02**	28.29	0.89	0.07	13.62	0.53	2.00
english.001.2	3.57	0.43	8.22	1.57	**0.06**	28.37	2.45	0.12	20.50	1.44	2.04
proteins.001.1	2.45	0.29	8.40	1.11	**0.04**	28.72	1.71	0.10	17.49	1.02	2.02
sources.001.2	3.58	0.43	8.27	1.59	**0.05**	29.19	2.50	0.12	20.13	1.48	2.04
artificial inputs											
rnd-8	7.98	0.81	9.80	8.18	**0.35**	23.63	9.25	0.44	21.26	4.92	215.00
rnd-12	12.08	1.23	9.86	12.48	**0.49**	25.46	13.90	0.62	22.40	7.43	226.00
rnd-16	15.94	1.46	10.89	16.68	**0.55**	30.15	18.58	0.85	21.75	9.92	237.00
rnd-20	19.85	1.69	11.76	21.17	**0.66**	31.91	23.31	1.83	12.74	12.40	252.00

insert operations in Line 5 of the induced sorting algorithm. Intuitively, induced sorting works because insert operations into a bucket are made in decreasing lexicographical order. For A-type suffixes the observation is analogous. For a full proof that this algorithm induces the correct SA, we refer the reader to [33]. Now we describe how to parallelize the induced sorting of the B-type suffixes. Sorting the A-type suffixes can be done analogously.

Parallel induced sorting. Inspecting Lines 3–6 of Fig. 4 reveals dependences in SA and $bucketB$ among iterations. We say that SA position $bucketB[S[SA[i]-1]$ is being initialized by position i. To perform the iteration i of the for-loop independently from the other iterations, we need to know the value of $SA[i]$ and of $bucketB[S[SA[i]-1]$ before the for-loop executes. Assuming an interval of SA values have already been initialized, the size of the buckets can be pre-computed using prefix sums, enabling the for-loop to be executed in parallel. If we make the simplifying assumption that consecutive characters are always different in the input string (an assumption which we will remove later), then $S[SA[i]-1] < S[SA[i]]$ holds on Line 5. Hence, no B-type suffix will be initialized by a suffix in the same bucket. Thus, once the loop has been executed for all B-type suffixes with lexicographical larger first characters than α, all B-type suffixes starting with character α have been initialized. This gives us a way to parallelize induced sorting for the case of no consecutive repeated characters in the input string by executing the for-loop σ times, each time processing all suffixes starting with a particular character in parallel. The pseudocode for this algorithm is shown in Fig. 6. Note that the intervals $[s,e]$ on Line 3 have already been computed as a byproduct of determining the B^* suffixes.

The complexity of the algorithm is dominated by the prefix sums (Line 9 of Fig. 6), leading to an $O(\sigma \log n)$ depth and $O(\sigma n)$ work algorithm. To make the algorithm linear-work, we compute $bucketSums[\alpha][i]$ only for every $\sigma \log n$'th position of i. We can easily compute $bucketSums[\alpha][i-1]$ from $bucketSums[\alpha][i]$ in constant work. So we can fill in the gaps by executing blocks of size $\sigma \log n$ of the loop in Line 10 sequentially, but in parallel across all blocks, thus resulting in an $O(\sigma^2 \log n)$ depth and $O(n)$ work algorithm. To store only every $\sigma \log n$'th value of $bucketSums$ we need n bits of space.

Dealing with repetitions in the input text. The previous algorithm assumes that there are no repeated consecutive characters. However, this does not hold in general. We now generalize the algorithm. We present a linear-work algorithm with depth $O(\sigma \log n \sum_{\alpha\in\Sigma} R_\alpha)$, where R_α is the longest run of the character α in T. For general inputs, the property $S[SA[i]-1] <$

Table 3
Memory consumption (bytes per input character) of WT construction on 32 cores. The algorithm with the lowest memory consumption among the parallel algorithms is marked in bold.

Input	levelWT	recWT	ddWT	serWT	sdslWT
non-repetitive real inputs					
sources	12.13	**4.14**	5.10	4.00	4.28
pitches	12.42	**4.43**	5.64	4.02	4.25
proteins	11.66	**3.66**	4.27	3.63	4.82
dna	11.57	**3.56**	4.05	3.50	4.34
english.1024MB	12.04	**4.04**	5.02	4.00	4.00
dblp.xml	11.97	**3.97**	4.82	3.88	4.82
repetitive real inputs					
Escherichia_Coli	11.70	**3.68**	4.13	3.51	4.22
cere	11.43	**3.43**	3.77	3.38	4.17
coreutils	12.13	**4.13**	5.08	4.00	4.33
einstein.de.txt	12.13	**4.17**	4.95	3.89	4.48
influenza	11.64	**3.64**	4.10	3.51	4.75
kernel	12.11	**4.11**	5.05	4.00	4.05
para	11.43	**3.44**	3.80	3.38	4.26
world_leaders	12.32	**4.33**	4.98	3.91	4.53
fib41	1.20	**1.19**	1.37	1.13	4.01
rs.13	1.21	**1.20**	1.35	1.13	4.25
tm29	**1.20**	1.20	1.34	1.13	5.01
dblp.xml.00001.1	12.11	**4.10**	4.88	3.89	4.30
dblp.xml.00001.2	12.14	**4.13**	4.90	3.89	4.30
dblp.xml.0001.1	12.10	**4.13**	4.86	3.89	4.31
dblp.xml.0001.2	12.11	**4.17**	4.86	3.89	4.32
dna.001.1	11.59	**3.55**	3.88	3.39	4.31
english.001.2	12.11	**4.13**	4.88	3.89	4.31
proteins.001.1	11.83	**3.88**	4.36	3.64	4.31
sources.001.2	12.11	**4.12**	4.89	3.89	4.30
artificial inputs					
rnd-8	47.43	39.37	**39.11**	39.17	54.86
rnd-12	49.69	41.70	**41.43**	41.49	57.19
rnd-16	51.90	**43.91**	47.91	43.69	69.95
rnd-20	54.11	**46.13**	138.06	45.92	72.19

Table 4
Sequential and parallel running times (seconds), and self-relative speedup of rank and select structure construction algorithms on 32 cores.

Input	parallelRank			SDSL-Rank	parallelSelect			SDSL-Select
	T_1	T_{64}	T_1/T_{64}	T_1	T_1	T_{64}	T_1/T_{64}	T_1
sources	0.93	**0.028**	33.2	0.5	2.5	**0.22**	11.4	4.4
pitches	0.24	**0.0077**	31.2	0.13	0.67	**0.061**	11.0	1.1
proteins	3.4	**0.09**	37.8	1.8	13	**0.88**	14.8	48
dna	0.88	**0.027**	32.6	0.49	3.9	**0.3**	13.0	24
english	4.9	**0.13**	37.7	2.7	13	**1.2**	10.8	28
dblp.xml	1.1	**0.034**	32.4	0.63	3.4	**0.3**	11.3	7.1

$S[SA[i]]$ is relaxed to $S[SA[i]-1] \leq S[SA[i]]$ when $SA[i]-1$ is a B-type suffix. This means that even after executing the loop for all B-type suffixes with lexicographical larger first character than α, not all SA values in the interval $[s, e]$ (refer to Fig. 6) have been initialized. In particular, all B-type suffixes with multiple repetitions of α have not been initialized. We observe that the B-type suffixes that begin with multiple repetitions of α are lexicographically smaller than those with only a single α. Thus $[s, e]$ can be divided into two contiguous parts $[s, e'-1]$ and $[e', e]$ where all SA values in $[e', e]$ have already been initialized and all values $[s, e'-1]$ still need to be initialized. The algorithm shown in Fig. 7 initializes in the k'th iteration of the while loop all suffixes that have $(k+1)$ repetitions of α, which would be done after Line 3 of the algorithm in Fig. 6. At most R_α iterations of the while loop in Line 3 of Fig. 7 are needed until all B-type suffix starting with α are initialized. Calculating m and the filter primitive require $O(\log n)$ depth. The overall depth of the algorithm is therefore $O(\sigma \log n \sum_{\alpha\in\Sigma} R_\alpha)$. $\sum_{\alpha\in\Sigma} R_\alpha$ has an upper bound of $O(n)$, but is much smaller for most real-world inputs. The overall work is $O(n)$, as constant work is spent for each $i \in [e', e]$ and all intervals $[e', e]$ are disjoint. As filter can be implemented using n bits of additional space the space requirement is unchanged.

Polylogarithmic depth. Theoretically, we can reduce the depth to $O(\sigma^2 \log n + \sigma \log^2 n)$ while maintaining linear work by processing suffixes with between 2^k and 2^{k+1} repetitions in parallel, for each value of $k \in \{0, \ldots, \log R_\alpha\}$. For each character,

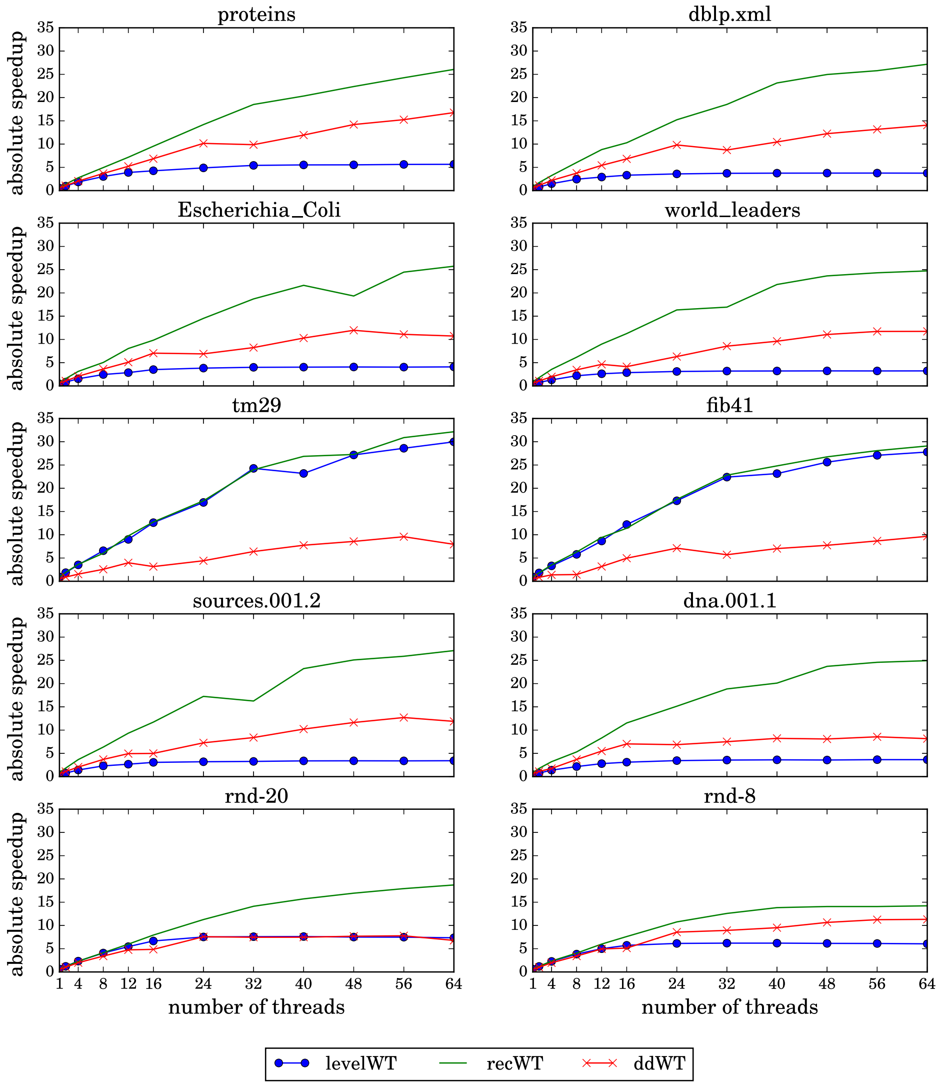


Fig. 8. Absolute speedup of wavelet tree construction compared to *serWT* as function of number of threads.

this requires $O(\log n)$ rounds, each with $O(\log n)$ depth, leading to the $O(\sigma \log^2 n)$ term. First, we calculate and store the number of repetitions of the character α for each suffix. This can be done in linear work and $O(\log n)$ depth by marking the end of runs in a bit-vector and constructing the rank/select structures. Then each suffix simply looks up this information by performing some arithmetic in $O(1)$ work. Now we initialize all suffixes in $[s, e]$ which have $[2^k, \dots, 2^{k+1}]$ repetitions of the character α, for increasing $k = 0, \dots, \log M$, where M is the maximum number of repetitions. For each value of k, the suffixes which have at least p repetitions of α can be initialized independently by filtering out the suffixes with less than p repetitions from the suffixes with $2^{\lfloor \log_2 p \rfloor}$ repetitions. Note that to use the filter primitive in linear work, the number of repetitions of the character α has to be pre-calculated. A suffix with p repetitions of α will contribute $O(p)$ to the overall work because it will be involved in at most $2p$ filter calls. However, a suffix with p repetitions also contributes to p suffixes in $[s, e]$. Thus the B-type suffixes starting with a character α can be initialized in $O(\log^2 n)$ depth and constant work per suffix.

For a constant alphabet size, this results in a polylogarithmic-depth and linear-work parallelization of the induced sorting approach used by *DivSufSort* or by the *SA-IS* algorithm (note that for polylogarithmic depth, this approach is only used for the first iteration, as σ may increase afterward). We did try an implementation of this theoretically-efficient algorithm but found that the simpler $O(\sigma \log n \sum_{\alpha \in \Sigma} R_\alpha)$ depth algorithm was much faster in practice due to $\sum_{\alpha \in \Sigma} R_\alpha$ being small and large overheads in the theoretically-efficient version. We report experimental results for the simpler version.

Table 5
Sequential and parallel running times (seconds), and speedup of SA construction on 32 cores.

Input	KS			Range			parDss			Scan			serDss
	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1
non-repetitive real inputs													
sources	99.92	6.45	15.48	68.87	4.38	15.74	44.46	**2.79**	15.94	19.92	20.89	0.95	19.17
pitches	20.69	1.28	16.15	16.64	1.03	16.07	8.36	**0.58**	14.46	4.19	17.52	0.24	3.76
proteins	608.03	38.32	15.87	512.01	26.24	19.51	452.75	**23.23**	19.49	181.77	38.17	4.76	178.08
dna	182.15	12.30	14.81	90.22	**6.65**	13.57	85.37	9.23	9.25	50.55	23.75	2.13	49.72
english.1024MB	566.10	35.60	15.90	849.91	41.04	20.71	549.60	**26.99**	20.36	149.55	36.74	4.07	151.42
dblp.xml	134.43	8.87	15.15	85.61	6.61	12.95	61.30	**4.53**	13.53	29.27	21.70	1.35	28.83
repetitive real inputs													
Escherichia_Coli	49.30	3.06	16.10	80.67	3.90	20.66	40.23	**3.01**	13.37	12.68	19.21	0.66	12.43
cere	204.47	**13.63**	15.00	320.33	18.28	17.52	188.72	16.25	11.61	52.71	24.74	2.13	52.77
coreutils	95.34	6.13	15.56	221.76	11.78	18.83	100.86	**5.58**	18.09	21.24	21.17	1.00	20.89
einstein.de.txt	39.94	2.48	16.12	74.02	4.61	16.05	37.03	**2.31**	16.06	9.07	20.30	0.45	8.75
influenza	65.24	**4.19**	15.58	76.28	4.49	17.00	47.44	4.28	11.09	16.27	20.70	0.79	16.10
kernel	121.94	7.97	15.30	254.53	15.67	16.24	127.85	**7.45**	17.16	27.76	21.99	1.26	26.68
para	192.48	**12.87**	14.96	284.89	15.04	18.94	158.99	14.87	10.69	51.30	22.89	2.24	50.79
world_leaders	17.54	1.16	15.12	26.04	1.94	13.42	4.65	**0.53**	8.72	2.10	18.71	0.11	1.92
fib41	75.49	**5.74**	13.16	367.91	28.91	12.73	235.78	20.69	11.39	45.41	28.57	1.59	44.55
rs.13	66.56	**4.64**	14.33	299.86	21.76	13.78	194.69	17.07	11.41	35.29	26.51	1.33	35.29
tm29	78.85	**6.07**	13.00	365.04	25.88	14.10	238.87	18.83	12.69	53.92	24.53	2.20	53.04
dblp.xml.00001.1	44.59	2.85	15.67	67.40	5.96	11.31	35.07	**2.49**	14.10	10.26	21.23	0.48	9.77
dblp.xml.00001.2	43.74	2.82	15.53	91.47	6.44	14.20	43.44	**2.71**	16.04	10.06	20.80	0.48	9.70
dblp.xml.0001.1	44.47	2.83	15.71	55.82	4.69	11.90	29.21	**2.13**	13.71	9.90	19.02	0.52	9.67
dblp.xml.0001.2	43.79	2.79	15.67	79.76	5.19	15.38	38.37	**2.39**	16.03	9.82	19.96	0.49	9.61
dna.001.1	42.61	**2.64**	16.12	33.99	2.68	12.66	25.71	2.67	9.63	10.77	19.15	0.56	10.49
english.001.2	44.77	2.79	16.06	64.63	3.77	17.15	39.14	**2.44**	16.03	11.53	19.10	0.60	11.10
proteins.001.1	46.40	2.78	16.68	37.65	2.87	13.10	31.74	**1.80**	17.60	12.22	18.50	0.66	11.89
sources.001.2	43.83	2.83	15.47	62.09	3.79	16.37	31.45	**2.52**	12.50	9.46	19.18	0.49	9.12
artificial inputs													
aaa	11.17	1.08	10.36	96.13	9.59	10.02	0.34	**0.54**	0.64	0.85	20.85	0.04	0.48
abab	12.40	**1.17**	10.59	102.34	11.49	8.91	38.11	4.46	8.55	2.28	20.31	0.11	2.12
aabbaabb	17.29	**1.28**	13.50	81.80	8.17	10.02	1.44	1.86	0.77	1.55	22.57	0.07	1.34

6. Parallel FM-index construction

By combining our parallel algorithms for WTs, rank and select structures, and SA construction, we can construct FM-indexes [7] in parallel. The BWT required by the FM-index is computed from the SA in the naive way, using $BWT[i] = S[SA[i] - 1]$. Our parallelization of induced sorting can also be applied to algorithms computing the BWT without first computing the SA. To compute the number of occurrences of a pattern in the text, only the WT of the BWT is needed. To compute the actual position of matches in the text, a sample of the SA is generated in parallel at the beginning and stored.

7. Experiments

We present experimental results of our implementations of the parallel algorithms described in this paper. We use a 32-core machine with two 16 core Intel(R) Xeon(R) E5-2683 v4 @ 2.10 GHz CPUs with enabled hyper-threading, and 512 GB main memory. We use Cilk Plus to express parallelism, and the code is compiled with the gcc 5.2.0 passing the -O2 flag. We use the Pizza&Chili corpus http://pizzachili.dcc.uchile.cl, random integer sequences with alphabet sizes 2^k (rnd-2^k) and the generated files *aaa*, *abab* and *aabbaabb* for testing. The file *aaa* contains the string $a^{104857600}$, *abab* the string $(ab)^{52428800}$ and *aabbaabb* the string $(a^{524288}b^{524288})^{100}$. The file sizes, alphabet sizes and average LCP value of the input files are listed in Table 1, grouped according to their characteristics. For example, some inputs are highly repetitive while others are not. We report both running times and memory consumption of the algorithms. Memory consumption includes the input and the output. The fastest or most memory-efficient *parallel* implementations are marked in bold in the tables. In addition, we provide plots of absolute speedup versus thread count.

Wavelet tree. Table 2 compares the 32-core running time of our algorithms *ddWT* and *recursiveWT* to the parallel *levelWT* implementation, the serial *serWT* implementation from [37] and the SDSL implementation *sdslWT*, and Table 3 compares their memory consumption. The reported times do not include the construction times of the rank/select structures on the bit-vectors of the wavelet tree. *ddWT* and *recursiveWT* clearly outperform *levelWT* on byte alphabets. On the non-repetitive

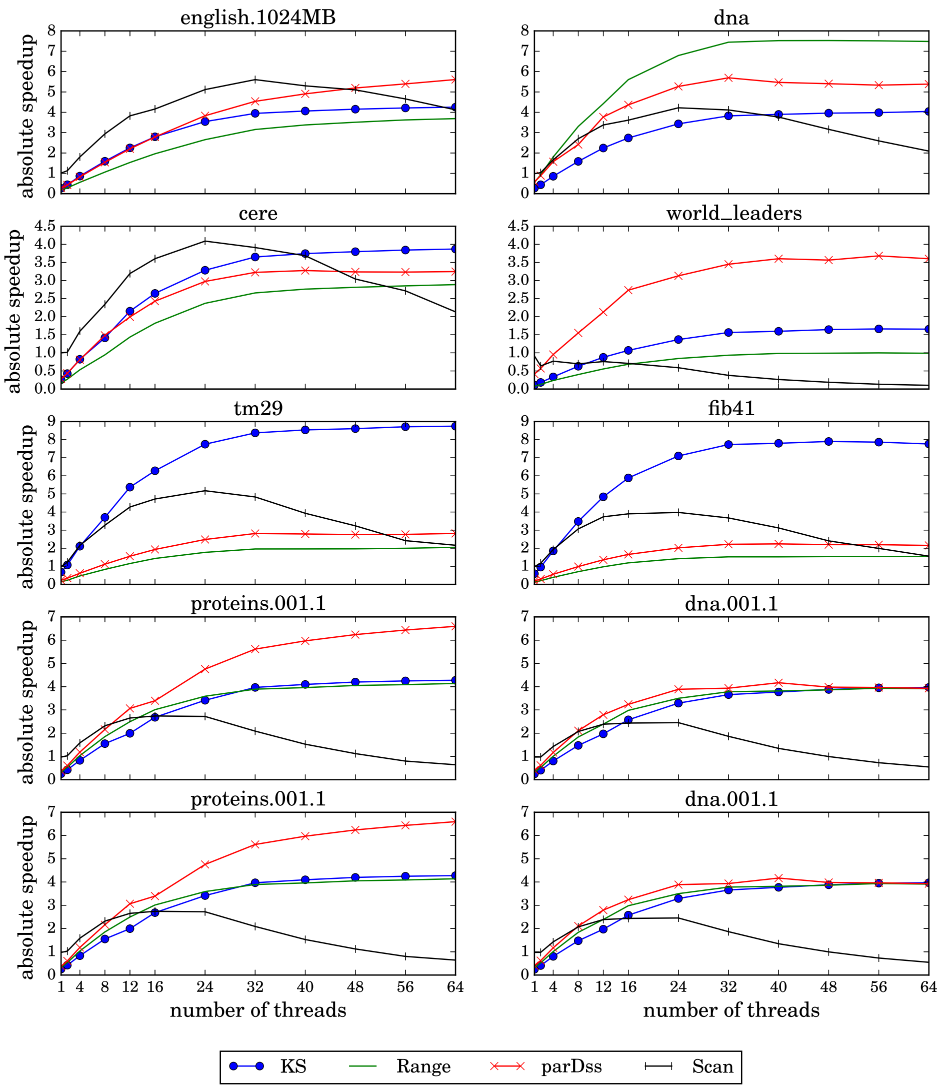


Fig. 9. Absolute speedup of suffix array construction compared to *serDss* as a function of number of threads.

real inputs *recursiveWT* is around 4–7x faster than *levelWT* while using around 3x less memory. Both *recursiveWT* and *ddWT* achieve good self-relative speedups on 32 cores, and compared to *serWT*, they are 25–27x and 13–16x faster, respectively on these inputs. *RecursiveWT* outperforms the other algorithms for all input files. Fig. 8 compares the speedup relative to *serWT* of the parallel algorithms on selected input files as a function of thread count.

Rank and select. Table 4 compares the construction times of our parallel implementation of rank/select structures on 32 cores to the sequential times from the SDSL. As input the concatenated levels of the WTs of the input files were used. The self-relative speedup is 31–38x for rank and 10–15x for select. Compared to the sequential SDSL times, we are 16–21x faster for rank and 18–80x faster for select (note that our parallel select on a single thread outperforms the sequential SDSL implementation). We speculate that speedups for rank are higher than for select due to the simpler memory layout of the rank structure.

Suffix array. Tables 5 and 6 compare the running time and memory usage of our parallel DivSufSort algorithm (*parDss*) to several existing parallel algorithms. *Range* is an implementation of the prefix doubling algorithm [24] from the PBBS (without the changes described in this work), *KS* is an implementation of the DC3 algorithm [19] from the PBBS, *Scan* is the in-memory version of the recently published divide and conquer algorithm by Kärkkäinen et al. [20], and *serDss* is the original DivSufSort implementation of the two-stage algorithm [17] implemented by Mori [29]. On 32 cores and the non-repetitive inputs, *parDss* achieves a self-relative speedup of 9–20x and outperforms *Scan*, *Range*, and *KS* on almost all non-repetitive inputs. Compared to *serDss*, *parDss* is 5–7x faster on 32 cores for the non-repetitive inputs. Additionally,

Table 6
Memory consumption (bytes per input character) of SA construction on 32 cores.

Input	KS	Range	parDss	Scan	serDss
non-repetitive real inputs					
sources	21.57	28.94	**6.83**	13.06	5.01
pitches	21.76	28.82	**7.27**	20.36	5.03
proteins	21.42	28.01	**6.79**	10.82	5.00
dna	22.95	27.93	**6.74**	11.75	5.00
english.1024MB	23.26	29.12	**6.84**	10.96	5.00
dblp.xml	21.50	28.78	**7.07**	12.30	5.01
repetitive real inputs					
Escherichia_Coli	23.08	28.58	**6.96**	14.57	5.02
cere	22.61	28.96	**6.56**	11.57	5.00
coreutils	21.56	29.56	**6.87**	12.47	5.01
einstein.de.txt	21.69	28.85	**7.17**	15.76	5.02
influenza	21.59	28.18	**6.92**	12.94	5.01
kernel	25.10	29.07	**6.81**	12.20	5.01
para	21.47	28.48	**6.67**	11.65	5.00
world_leaders	21.86	33.15	**6.79**	19.56	5.02
fib41	21.74	39.06	**9.82**	12.03	5.01
rs.13	21.48	39.91	**9.93**	12.29	5.01
tm29	22.43	40.84	**9.00**	12.05	5.00
dblp.xml.00001.1	21.64	29.21	**7.27**	14.40	5.01
dblp.xml.00001.2	21.62	30.65	**7.30**	14.41	5.01
dblp.xml.0001.1	21.66	29.31	**7.21**	14.99	5.01
dblp.xml.0001.2	21.64	30.62	**7.46**	14.11	5.01
dna.001.1	21.66	27.84	**6.97**	13.91	5.01
english.001.2	21.68	29.51	**7.13**	14.40	5.01
proteins.001.1	20.36	27.79	**6.96**	13.84	5.01
sources.001.2	21.65	29.62	**6.99**	15.02	5.01
artificial inputs					
aaa	21.66	33.20	**5.49**	13.02	5.01
abab	23.03	37.82	**10.24**	13.10	5.01
aabbaabb	20.33	37.15	**5.37**	14.94	5.01

Input	T_1	T_{64}	$\frac{T_1}{T_{64}}$	T_1
non-repetitive real inputs				
sources	130.00	**7.29**	17.83	33.80
pitches	32.10	**1.76**	18.24	6.62
proteins	705.00	**40.50**	17.41	249.00
dna	171.00	**15.60**	10.96	73.90
english.1024MB	721.00	**44.40**	16.24	224.00
dblp.xml	176.00	**10.50**	16.76	47.90
repetitive real inputs				
Escherichia_Coli	62.20	**4.90**	12.69	17.70
cere	276.00	**23.20**	11.90	74.70
coreutils	182.00	**9.93**	18.33	30.40
einstein.de.txt	70.90	**4.15**	17.08	12.80
influenza	76.20	**6.71**	11.36	23.00
kernel	228.00	**12.80**	17.81	39.00
para	244.00	**21.20**	11.51	71.10
world_leaders	16.40	**1.38**	11.88	3.80
fib41	261.00	**24.70**	10.57	58.60
rs.13	219.00	**20.20**	10.84	46.70
tm29	268.00	**22.60**	11.86	65.90
dblp.xml.00001.1	74.30	**4.63**	16.05	15.20
dblp.xml.00001.2	81.80	**4.95**	16.53	15.60
dblp.xml.0001.1	69.00	**4.38**	15.75	15.20
dblp.xml.0001.2	77.10	**4.57**	16.87	14.80
dna.001.1	44.50	**4.31**	10.32	14.90
english.001.2	73.70	**4.51**	16.34	15.50
proteins.001.1	63.90	**3.82**	16.73	16.20
sources.001.2	71.80	**4.81**	14.93	13.40
artificial inputs				
aaa	3.33	**2.09**	1.59	3.47
abab	48.30	**5.57**	8.67	5.60
aabbaabb	11.30	**2.95**	3.83	5.33

Input	par	ser
non-repetitive real inputs		
sources	**8.11**	6.01
pitches	**8.36**	6.03
proteins	**7.84**	6.00
dna	**7.72**	6.00
english.1024MB	**7.84**	6.00
dblp.xml	**8.05**	6.01
repetitive real inputs		
Escherichia_Coli	**7.90**	6.01
cere	**7.66**	6.00
coreutils	**8.15**	6.01
einstein.de.txt	**8.37**	6.01
influenza	**7.84**	6.01
kernel	**8.08**	6.01
para	**7.73**	6.00
world_leaders	**7.64**	6.03
fib41	**10.72**	6.01
rs.13	**10.60**	6.01
tm29	**10.46**	6.00
dblp.xml.00001.1	**8.28**	6.01
dblp.xml.00001.2	**8.37**	6.02
dblp.xml.0001.1	**8.44**	6.01
dblp.xml.0001.2	**8.37**	6.01
dna.001.1	**8.05**	6.01
english.001.2	**8.35**	6.01
proteins.001.1	**8.37**	6.01
sources.001.2	**8.20**	6.01
artificial inputs		
aaa	**6.52**	6.01
abab	**11.41**	6.01
aabbaabb	**6.38**	6.01

Fig. 10. Left: Sequential and parallel running times (seconds), and self-relative speedup of FM-index construction on 32 cores. **Right:** Memory consumption (bytes per input character) of FM-index construction on 32 cores.

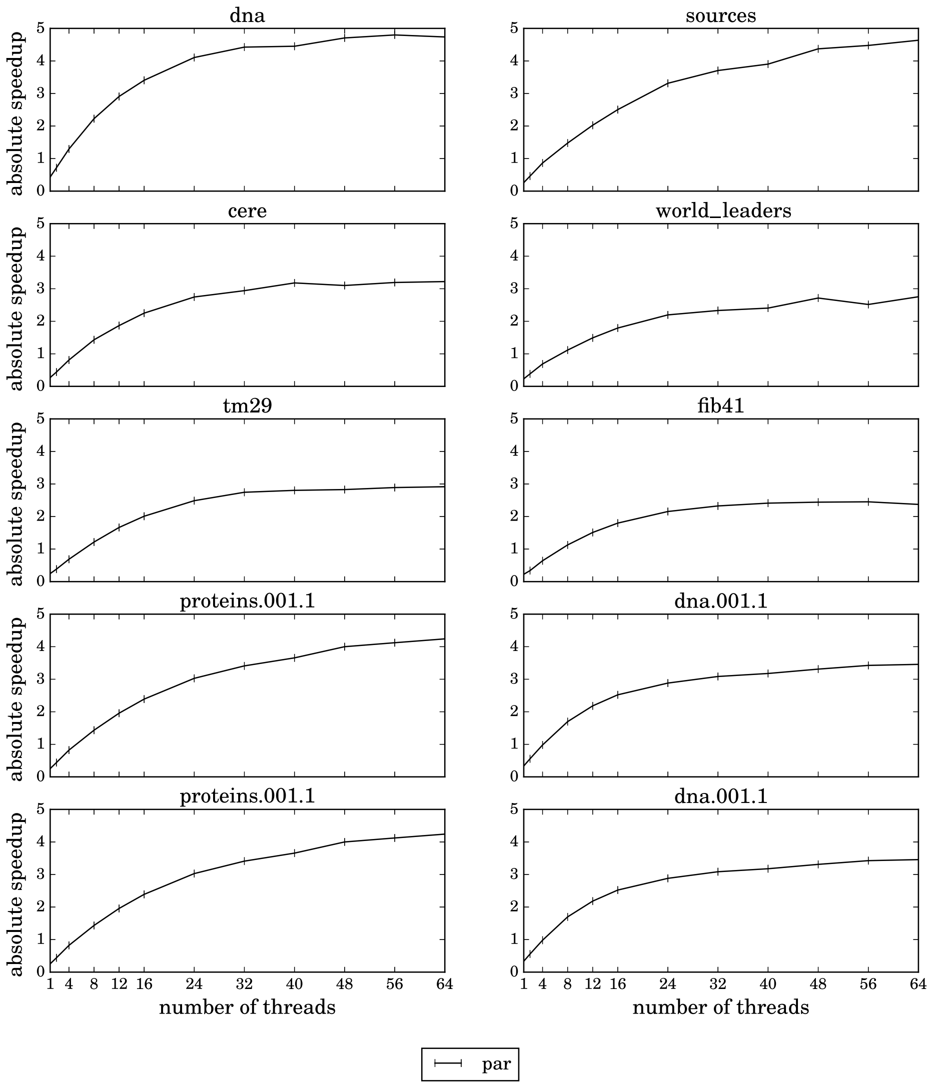


Fig. 11. Absolute speedup of FM-index construction compared to sequential *sdsl* construction as function of number of threads.

parDss is almost in-place, so it reduces memory consumption by around 4x compared to *KS*, 5x compared to *Range*, and 1.8x compared to *Scan*. For the other inputs *parDss* is one of the fastest algorithms for the inputs were the average LCP value is below one million. For inputs with very large average LCP values *KS* is the fastest algorithm. Fig. 9 compares the speedup relative to *serDss* of the parallel algorithms on selected input files as a function of thread count. *Scan* outperforms *parDss* for smaller thread counts. However, *Scan* does not scale to more than 20 threads.

We also compare with results reported in the literature for other algorithms on two files, namely chr1 and HG19, downloaded from the UCSC Genome Browser. chr1[1] is the first chromosome from the HG19 dataset in FASTA format. HG19[2] are all concatenated chromosomes from the HG19 dataset in FASTA format. *parDss* takes 7.9 seconds to build the SA for the chr1 file on 16 threads, which is 14x faster than the times reported for computing the BWT in [15], 8x faster than the times reported for computing the SA with mkESA [16], and 2.8x faster than the *bwtrev* algorithm [34]. *parDss* takes 236 seconds to build the SA for the complete HG19 file on 12 threads, which is 4x faster than the 12-thread running time reported for the *ParaBWT* algorithm [26]. Note that these numbers should only be used as a very rough performance estimation. The different machines and experimental settings used do not allow an accurate direct comparison.

FM-index. Fig. 10 shows that by plugging our algorithms into the SDSL we can get parallel speedups for almost all but the artificial inputs of 10–18x constructing FM-indexes and reduce the absolute construction time by up to 6x (using a Huffman-

[1] http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/chr1.fa.gz.
[2] http://hgdownload.soe.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz.

shaped WT). For constructing the FM-index, our parallel performance is comparable to the parallel PASQUAL framework [25], but PASQUAL is designed to handle only DNA inputs. Fig. 11 compares the speedup relative to the sequential SDSL implementation as a function of thread count. The figure shows that for all input and at least 8 available threads there is a clear performance benefit of using the parallel over the sequential implementation.

8. Conclusion

In this work, we showed how to parallelize a number of basic construction algorithms needed for compressed full-text indexes. We covered rank and select structures on bit-vectors, wavelet trees and suffix arrays. Additionally, we showed how to use the parallelized algorithms to construct FM-indexes in parallel. Our experiments show that our implementations are memory-efficient, achieve good parallel scalability, and outperform existing implementations for the same problem. Our work shows that a focus on memory efficiency can also improve the performance of algorithms. Implementations are available to the public as part of the Problem Based Benchmark Suite (PBBS) [39] and the Succinct Data Structure Library (SDSL) [13].[3]

References

[1] M.A. Babenko, P. Gawrychowski, T. Kociumaka, T.A. Starikovskaya, Wavelet trees meet suffix trees, in: Proc. of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2015, pp. 572–591.
[2] T. Bingmann, A. Eberle, P. Sanders, Engineering parallel string sorting, Algorithmica (2014) 1–52.
[3] M. Buro, On the maximum length of Huffman codes, Inf. Process. Lett. 45 (1993) 219–223.
[4] D. Clark, Compact Pat Trees, PhD thesis, University of Waterloo, 1996.
[5] M. Deo, S. Keely, Parallel suffix array and least common prefix for the GPU, in: SIGPLAN Notices, vol. 48, ACM, 2013, pp. 197–206.
[6] J.A. Edwards, U. Vishkin, Parallel algorithms for Burrows–Wheeler compression and decompression, Theor. Comput. Sci. 525 (2014) 10–22.
[7] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. of the 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 390–398.
[8] L. Ferres, J. Fuentes-Sepúlveda, M. He, N. Zeh, Parallel construction of succinct trees, in: Proc. of the International Symposium on Experimental Algorithms, Springer, 2015, pp. 3–14.
[9] P. Flick, S. Aluru, Parallel distributed memory construction of suffix and longest common prefix arrays, in: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 16.
[10] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, D. Seco, Efficient wavelet tree construction and querying for multicore architectures, in: Proc. of the International Symposium on Experimental Algorithms, 2014, pp. 150–161.
[11] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, D. Seco, Parallel construction of wavelet trees on multicore architectures, Knowl. Inf. Syst. (2016) 1–24.
[12] S. Gog, M. Petri, Optimized succinct data structures for massive data, Softw. Pract. Exp. 44 (11) (2014) 1287–1314.
[13] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, in: Proc. of the International Symposium on Experimental Algorithms, Springer, 2014, pp. 326–337.
[14] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2003, pp. 841–850.
[15] S. Hayashi, K. Taura, Parallel and memory-efficient Burrows–Wheeler transform, in: Proc. of the International Conference on Big Data, IEEE, 2013, pp. 43–50.
[16] R. Homann, D. Fleer, R. Giegerich, M. Rehmsmeier, mkESA: enhanced suffix array construction tool, Bioinformatics 25 (8) (2009) 1084–1085.
[17] H. Itoh, H. Tanaka, An efficient method for in memory construction of suffix arrays, in: Proc. of the String Processing and Information Retrieval Symposium, IEEE, 1999, pp. 81–88.
[18] J. JaJa, An Introduction to Parallel Algorithms, Addison–Wesley, 1992.
[19] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: International Colloquium on Automata, Languages, and Programming, Springer, 2003, pp. 943–955.
[20] J. Kärkkäinen, D. Kempa, S.J. Puglisi, Parallel external memory suffix sorting, in: Proc. of the Annual Symposium on Combinatorial Pattern Matching, Springer, 2015, pp. 329–342.
[21] F. Kulla, P. Sanders, Scalable parallel suffix array construction, Parallel Comput. 33 (9) (2007) 605–612.
[22] J. Labeit, Parallel lightweight wavelet tree, suffix array and FM-index construction, 2015.
[23] J. Labeit, J. Shun, G.E. Blelloch, Parallel lightweight wavelet tree, suffix array and FM-index construction, in: Data Compression Conference, IEEE, 2016, pp. 33–42.
[24] N.J. Larsson, K. Sadakane, Faster suffix sorting, Theor. Comput. Sci. 387 (3) (2007) 258–272.
[25] X. Liu, P. Pande, H. Meyerhenke, D.A. Bader, PASQUAL: parallel techniques for next generation genome sequence assembly, IEEE Trans. Parallel Distrib. Syst. 24 (5) (2013) 977–986.
[26] Y. Liu, T. Hankeln, B. Schmidt, Parallel and space-efficient construction of Burrows–Wheeler transform and suffix array for big genome data, IEEE/ACM Trans. Comput. Biol. Bioinform. 13 (3) (2015) 592–598.
[27] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, in: Proc. of the Annual Symposium on Combinatorial Pattern Matching, Springer, 2005, pp. 45–56.
[28] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[29] Y. Mori, Libdivsufsort: a lightweight suffix sorting library, https://github.com/y-256/libdivsufsort, version 2.0.2-1.
[30] J.I. Munro, Y. Nekrich, J.S. Vitter, Fast construction of wavelet trees, Theor. Comput. Sci. 638 (2016) 91–97.
[31] G. Navarro, Wavelet trees for all, J. Discret. Algorithms 25 (2014) 2–20.
[32] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007) 2.
[33] G. Nong, S. Zhang, W.H. Chan, Linear suffix array construction by almost pure induced-sorting, in: Proc. of the Data Compression Conference, IEEE, 2009, pp. 193–202.
[34] E. Ohlebusch, T. Beller, M.I. Abouelhoda, Computing the Burrows–Wheeler transform of a string and its reverse in parallel, J. Discret. Algorithms 25 (2014) 21–33.

[3] Source code can be found under www.cs.cmu.edu/~pbbs and https://github.com/jlabeit/sdsl-lite/tree/parallel_sdsl.

[35] V. Osipov, Parallel suffix array construction for shared memory architectures, in: Proc. of the International Symposium on String Processing and Information Retrieval, Springer, 2012, pp. 379–384.
[36] S.J. Puglisi, W.F. Smyth, A.H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2) (2007) 4.
[37] J. Shun, Parallel wavelet tree construction, in: Proc. of the Data Compression Conference, IEEE, 2015, pp. 63–72.
[38] J. Shun, Improved parallel construction of wavelet trees and rank/select structures, in: Proc. of the Data Compression Conference, IEEE, 2017, in press, arXiv:1610.03524.
[39] J. Shun, G.E. Blelloch, J.T. Fineman, P.B. Gibbons, A. Kyrola, H.V. Simhadri, K. Tangwongsan, Brief announcement: the problem based benchmark suite, in: Proc. of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2012, pp. 68–70.
[40] S. Vigna, Broadword implementation of rank/select queries, in: International Workshop on Experimental and Efficient Algorithms, Springer, 2008, pp. 154–168.
[41] U. Vishkin, Thinking in parallel: some basic data-parallel algorithms and techniques, 2010.
[42] L. Wang, S. Baxter, J.D. Owens, Fast parallel suffix array on the GPU, in: Proc. of the European Conference on Parallel Processing, Springer, 2015, pp. 573–587.