Parallel Lightweight Wavelet Tree, Suffix Array and FM-Index Construction

Julian Labeit
Karlsruhe Institute of Technology
julianlabeit@gmail.com

Julian Shun
UC Berkeley
jshun@eecs.berkeley.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

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 parallel wavelet tree algorithm matches that of the best existing algorithm while requiring asymptotically less memory. Our experiments show that it is 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. The induced copying requires linear work and polylogarithmic depth for constant alphabets. 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.*

Introduction. In recent years, compressed full-text indexes [23] have become popular as they provide an elegant way of compressing data while at the same time supporting queries on it 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) [7] is a state-of-the-art library for constructing these data structures sequentially. Additionally, in recent years wavelet tree construction [6, 28] and linear-work suffix array construction [13] have been successfully parallelized. 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 [28] 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 64 cores show that our modified algorithm achieves a speedup of 2–4x over the original algorithm of [28], and achieves a speedup of 23–38x over the fastest sequential algorithm. Additionally, we propose a variation of the domain-decomposition algorithm by Fuentes et al. [6], which requires the same work and space as our first algorithm when $\sigma / \log \sigma \in O(\log n)$. We also present an experimental evaluation of constructing rank and select structures, which are used in wavelet trees, in parallel, and show speedups of 11–38x on 64 cores.

For suffix array construction there are three main classes of algorithms described in the literature [27]: prefix doubling, recursive, and induced copying (sorting) algorithms. While prefix doubling [17] and recursive algorithms [13] 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

This work is a short version of Julian Labeit's bachelor's thesis [16]. 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.

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) [29]. 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 [11]. 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 64 cores, our algorithm is up to 1.6x faster and uses 1.8x less memory than the fastest existing parallel algorithm, and achieves a speedup of 4–9x over the fastest sequential algorithm.

Finally, we use our algorithms to construct FM-indexes [4], 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 64 cores we achieve self-relative speedups of 17–33x and absolute speedups of 6–9x over the sequential SDSL implementation on a variety of real-world inputs.

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 [12] we can bound the running time by $O(W/P + D)$ using P processors on a PRAM. We allow for concurrent reading and writing to shared memory locations. 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 0 with $0 \oplus x = x$ for all x, and returns the array $\{0, A[0], A[0] \oplus A[1], \ldots, A[0] \oplus A[1] \ldots \oplus A[n-2]\}$ as well as the overall sum $A[0] \oplus A[1] \ldots \oplus A[n-1]$. Prefix sum can be implemented with $O(n)$ work and $O(\log n)$ depth [12]. *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. By dividing the input in 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 appearances of character c in $S[0, \ldots, i - 1]$ and $select_c(S, i)$ calculates the position of the i'th appearance 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.

Wavelet trees (WT) generalize this to larger alphabets [8]. 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 $S[i] \leq (a + b)/2$ 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 $[\lfloor (a + b)/2 \rfloor + 1, \ldots, b]$. 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. We refer the reader to [22] for numerous applications of WTs.

The *FM-index* is a compressed full-text index using the BWT of a text [4]. 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 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.

Related Work. Fuentes et al. [6] 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 [28] 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. [5] describe how to construct range min-max trees in parallel. Their structure can support rank/select queries on bit-vectors in $O(\log n)$.

Suffix arrays were first introduced by Manber and Myers [21] as a space-efficient alternative to suffix trees. Since then, many different suffix array algorithms have been developed, including the difference cover (DC3) algorithm [13] and the induced sorting algorithm (SA-IS) [24]. 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 [29], distributed memory [15], and GPUs [3, 26, 31]. SA-IS is a lightweight linear-work algorithm and one of the fastest in practice. Unfortunately, it is hard to parallelize as induced sorting consists of multiple sequential scans with non-trivial data dependences.

Many bioinformatics applications use compressed SAs, and thus there have been many frameworks with parallel SA implementations optimized for DNA inputs [10, 18]. For example, PASQUAL [18] 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 [9, 19].

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.

The levelWT algorithm proposed by Shun [28] 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 [28]. In particular, the two children of a node are constructed via two recursive calls in parallel. This approach 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
4      $v.bitvector :=$ bitvector of size $|S|$
5      **parfor** $i := 0$ to $|S| - 1$:
6          $v.bitvector[i] = \begin{cases} 0, \text{if } S[i] \leq (a+b)/2, \\ 1, \text{else} \end{cases}$
7      $(S_0, S_1) =$ parallelSplit $(S, v.bitvector)$
8      $v.leftChild =$ recursiveWT $(S_0, [a, \lfloor \frac{a+b}{2} \rfloor])$ //asynchronous parallel call
9      $v.rightChild =$ recursiveWT $(S_1, [\lfloor \frac{a+b}{2} \rfloor + 1, b])$ //asynchronous parallel call
10     **return** $v$
```

Our second algorithm for WT construction (*ddWT*) is a modified version of the domain decomposition algorithm introduced by Fuentes et al. [6]. The first part of our algorithm is the same as the original algorithm: the input string is split into P chunks, and a WT is constructed for each chunk independently in parallel. The second part of the algorithm involves merging the WTs. In the original algorithm of [6], 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. If $\sigma / \log \sigma \in O(\log n)$ then our algorithm requires $O(n \log \sigma)$ work, $O(\log n \log \sigma)$ depth and in $O(n \log \sigma)$ bits of space. Due to limited space, we refer the reader to [16] for a detailed description and cost analysis.

Both recursiveWT and ddWT can easily be adapted to construct different shapes of WTs, such as Huffman-shaped WTs [20]. As part of a parallel version of the SDSL, we provide implementations of recursiveWT for constructing various shapes of WTs.

We now describe our parallel implementation of constructing rank and select structures on bit-vectors based on the ideas of [28]. For rank, we chose to parallelize the broadword implementation in SDSL [30]. The answers to the rank queries are pre-computed and stored in first and second-level lookup tables. 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. For select, we parallelize the SDSL implementation of a variant of the structure by Clark [2]. First, the location of every 4096'th 1-bit is stored. The regions between stored 1-bits define blocks. We refer to a block that spans more than $\log^4 n$ bits as a *long* block, and the remaining blocks as *short* blocks. For long blocks, the answers to all queries are stored explicitly. For short blocks, the location of every every 64'th 1-bit is stored, again defining sub-blocks. For sub-blocks of longer than $\log(n)/2$ bits, the answers are stored explicitly, and for the other blocks, a linear scan of the original bit-vector in the range of the sub-block is performed to answer the query.[1] For a more detailed analysis, see [2]. 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.

Parallel Suffix Array Construction. Previous parallel SA algorithms either use the recursive [13] or prefix doubling [17] approach. However the fastest and most space-efficient sequential SA algorithms use induced copying [27], so our goal here is to parallelize such an algorithm. We first describe a simple parallel prefix doubling algorithm, which in practice needs $n(2 + \log n) + o(n)$ 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.

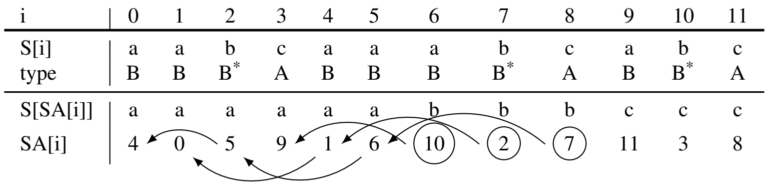
ParallelRange from the PBBS [29] is a parallel version of the algorithm described by Larsson and Sadakane [17]. It starts with an approximate SA and ISA, in which the suffixes are only sorted by their first character. A series of refinement steps are then applied, where on step d, groups of suffixes with the same ISA value from the previous step are sorted by their first 2^d characters by accessing $ISA[SA[i] + 2^d]$ for suffix i. 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 the sorting routine. Using a parallel integer sorting algorithm with $O(n)$ work and $O(n^\epsilon)$ depth for $0 < \epsilon < 1$ [12], parallelRange has $O(n \log n)$ work and $O(n^\epsilon \log n)$ depth. In practice, the work is usually much closer to linear.

Using parallelRange, we can parallelize the *DivSufSort* implementation of the two-stage algorithm [11] by Mori. In the first step the suffixes are categorized into A, B, and B^* suffixes. A suffix $S[i, \ldots, n-1]$ is of type A if $S[i+1, \ldots, n-1] < S[i, \ldots, n-1]$ 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. 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 [1]. Our implementation, however, only parallelizes an initial bucket sort and uses the sequential multikey quicksort for the resulting buckets, which we found to be sufficient in practice. The third step constructs the SA of the reduced text. As the text size has reduced by at least half, the unused part of the SA can be used for the ISA, and thus parallelRange 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

[1]In theory, a lookup table is used to store all possible answers to small sub-blocks, however a linear scan works better in practice.

```
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         $bucketB[S[SA[i]-1]]--$
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        $bucketA[S[SA[i]-1]]++$
```

Figure 1: 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

Figure 2: 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.

polylogarithmic-depth algorithm for induced sorting on constant-sized alphabets.

The sequential algorithm for induced sorting (shown in Figure 1) 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 lexicographical smaller. Figure 2 shows an example of how induced sorting inserts B-type suffixes into the SA. The arrows indicate the 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 [24]. Now we describe how to parallelize the induced sorting of the B-type suffixes. Sorting the A-type suffixes can be done analogously.

Inspecting Lines 3–6 of Figure 1 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, 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 Figure 3. Note that the intervals

```
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              $bucketSums[S[SA[i] - 1]][i] + +$
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             $SA[bucketB[b] - bucketSums[b][i]] = SA[i] - 1$
```

Figure 3: Parallel induced sorting of all B-type suffixes for inputs with no repetitions of characters.

$[s, e]$ 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, 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. Then we only need n bits to store *bucketSums*. We can easily compute $bucketSums[\alpha][i-1]$ from $bucketSums[\alpha][i]$ in constant time. 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.

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 an 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 $S[SA[i]-1] < S[SA[i]]$ 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 Figure 3) 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 Figure 4 initializes in the k'th iteration of the while loop all suffixes which have $(k+1)$ repetitions of α, which would be done after Line 3 of the algorithm in Figure 3. At most R_α iterations of the while loop in Line 3 of Figure 4 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 then $O(\sigma \log n \sum_{\alpha \in \Sigma} R_\alpha)$. $\sum_{\alpha \in \Sigma} R_\alpha$ 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.

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, this requires $O(\log n)$ rounds, each with $O(\log n)$ depth, leading to the $O(\sigma \log^2 n)$ term. Due to space limitations, we describe the details of this approach in [16]. For constant alphabet size, this results in a polylogarithmic-

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

Figure 4: Subroutine of parallel induced sorting of B-type suffixes for inputs with repetitions (insert after Line 3 of the pseudocode in Figure 3).

			levelWT			recursiveWT			ddWT			serWT
Input	Input Size (MB)	σ	T_1	T_{64}	T_1/T_{64}	T_1	T_{64}	T_1/T_{64}	T_1	T_{64}	T_1/T_{64}	T_1
sources	211	230	19	0.88	21.6	12	**0.26**	46.2	16	0.45	35.5	9.2
pitches	56	133	4.7	0.23	20.4	2.8	**0.068**	41.2	3.7	0.1	37.0	2.2
proteins	1184	27	72	2.9	24.8	56	**1.1**	50.9	69	1.7	40.6	39
dna	404	16	16	0.72	22.2	12	**0.24**	50.0	15	0.47	31.9	8.7
english	1024	239	99	4.4	22.5	65	**1.3**	50.0	110	2.2	50.0	49
dblp.xml	295	97	24	1.1	21.8	16	**0.34**	47.0	20	0.57	35.1	12
rnd-2^8	381	2^8	10	0.58	17.2	9.9	**0.27**	36.7	13	0.45	28.9	6.4
rnd-2^{12}	381	2^{12}	14	0.84	16.7	15	**0.38**	39.5	19	0.61	31.1	9.6
rnd-2^{16}	381	2^{16}	20	1.0	20.0	20	**0.48**	41.7	25	0.81	30.9	13
rnd-2^{20}	381	2^{20}	24	1.2	20.0	27	**0.61**	44.3	32	1.4	22.9	16

Table 1: Running times (seconds) sequential, parallel and self-relative speedup of WT construction algorithms on 64 cores.

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 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.

Parallel FM-Index Construction. By combining our parallel algorithms for WTs, rank and select structures, and SA construction, we can construct FM-indexes [4] in parallel. The BWT required by the FM-index is computed from the SA in the naive way. 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.

Experiments. We present experimental results of our implementations of the parallel algorithms described in this paper. We use a 64-core machine with four 2.4 GHz 16-core AMD Opteron 6278 processors, and 188GB main memory. We use Cilk Plus to express parallelism, and the code is compiled with the gcc 4.8.0 with the -O2 flag. We use the Pizza&Chili corpus `http://pizzachili.dcc.uchile.cl` and random integer sequences with alphabet sizes 2^k for testing. The file sizes and alphabet sizes are listed in Table 1. 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 following tables. Plots of speedup versus thread count can be found in [16].

Table 1 compares the 64-core running time of our algorithms *ddWT* and *recursiveWT* to the parallel *levelWT* implementation and the serial *serWT* implementation from [28], and

Input	levelWT	recursiveWT	ddWT	serWT	Input	levelWT	recursiveWT	ddWT	serWT
sources	11.5	**3.9**	4.8	3.8	dblp.xml	11.4	**3.7**	4.5	3.7
pitches	11.6	**4.0**	4.9	3.8	rnd-2^8	30.0	**26.0**	29.6	25.9
proteins	11.1	**3.5**	4.0	3.5	rnd-2^{12}	29.6	**28.2**	31.8	25.7
dna	11.0	**3.4**	3.8	3.3	rnd-2^{16}	32.2	**28.4**	34.1	28.3
english	11,5	**3.8**	4.7	3.8	rnd-2^{20}	34.8	**31.0**	81.9	30.9

Table 2: Memory consumption (bytes per input character) of WT construction on 64 cores.

	parallelRank			SDSL-Rank	parallelSelect			SDSL-Select
Input	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

Table 3: Sequential and parallel running times (seconds), and self-relative speedup of rank and select structure construction algorithms on 64 cores.

Table 2 compares their memory consumption. *ddWT* and *recursiveWT* clearly outperform *levelWT* on byte alphabets. *recursiveWT* also scales very well on larger alphabets, while *ddWT* is slower on larger alphabets. On byte alphabets *recursiveWT* is around 3x faster than *levelWT* while using around 2.8x less memory. Both *recursiveWT* and *ddWT* achieve good self-relative speedups on 64 cores, and compared to *serWT*, they are 23–28x and 11–22x faster, respectively.

Table 3 compares the construction times of our parallel implementation of rank/select structures on 64 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 11–14x for select. Compared to the sequential SDSL times, we are 16–20x faster for rank and 18–81x 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.

Table 4 and Table 5 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 [17] from the PBBS (without the changes described in this work), *KS* is an implementation of the DC3 algorithm [13] from the PBBS, *Scan* is the in-memory version of the recently published algorithm by Karkkainen et al. [14], and *serDss* is the original DivSufSort implementation. On 64 cores, *parDSS* achieves a self-relative speedup of 15–32x and outperforms *Scan*, *Range*, and *KS* on almost all inputs, achieving up to 1.6x speedup over the fastest existing parallel implementation. Compared to *serDSS*, *parDSS* is 4–9x faster on 64 cores. Additionally, *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*.

We also compare with results reported in the literature for other algorithms. *parDSS* takes 7.9 seconds to build the SA for the chr1 file[2] on 16 threads, which is 14x faster than the times reported for computing the BWT in [9], 8x faster than the times reported for computing the SA with mkESA [10], and 2.8x faster than the *bwtrev* algorithm [25]. *parDSS* takes 236 seconds to build the SA for the complete HG19 file[2] on 12 threads, which is 4x faster than the 12-thread running time reported for the *ParaBWT* algorithm [19].

Table 6 shows that by plugging our algorithms into the SDSL we can get parallel speedups of 17–33x constructing FM-indexes and reduce the absolute construction time

[2]Downloaded from `https://genome.ucsc.edu/`

	KS			Range			parDss			Scan			serDss
Input	T_1	T_{64}	T_1/T_{64}	T_1	T_{64}	T_1/T_{64}	T_1	T_{64}	T_1/T_{64}	T_1	T_{64}	T_1/T_{64}	T_1
sources	220	11	20.0	190	7.9	24.1	99	**4.8**	20.6	35	8.7	4.0	28
pitches	42	2.5	16.8	50	2.1	23.8	23	**1.4**	16.4	7.3	2.5	2.9	6.2
proteins	1900	65	29.2	1800	49	36.7	1100	**34**	32.4	380	53	7.2	300
dna	480	20	24.0	280	**10**	28.0	190	13	14.6	94	17	5.0	80
english	1900	62	30.6	2700	78	34.6	1300	**41**	31.7	320	47	6.7	230
dblp.xml	310	15	20.7	210	11	19.1	130	**7.0**	18.6	54	12	4.5	42

Table 4: Sequential and parallel running times (seconds), and speedup of SA construction on 64 cores.

Input	KS	Range	parDss	Scan	serDss
sources	21.4	28.5	**5.2**	10.0	5.0
pitches	21.3	28.1	**5.7**	10.0	5.0
proteins	20.1	26.3	**5.1**	10.0	5.0
dna	21.4	27.8	**5.5**	10.0	5.0
english	21.5	28.9	**5.2**	10.0	5.0
dblp.xml	21.5	28.7	**5.6**	10.0	5.0

Table 5: Memory consumption (bytes per input character) of SA construction on 64 cores.

Input	T_1	T_{64}	T_1/T_{64}	T_1
sources	180	**7.5**	24.0	45
pitches	42	**2.0**	21.0	11
proteins	1600	**49**	32.7	420
dna	290	**17**	17.1	110
english	1700	**53**	32.1	360
dblp.xml	240	**10**	24.0	72

Table 6: Sequential and parallel running times (seconds), and self-relative speedup of FM-index construction on 64 cores.

by 5.3–8.5x (using a Huffman-shaped WT). For constructing the FM-index, our parallel performance is comparable to the parallel PASQUAL framework [18], but PASQUAL is designed to handle only DNA inputs.

References.

[1] T. Bingmann et al. “Engineering Parallel String Sorting”. In *arXiv:1403.2056* (2014).
[2] D. Clark. “Compact Pat Trees”. PhD thesis. University of Waterloo, 1996.
[3] M. Deo and S. Keely. “Parallel suffix array and least common prefix for the GPU”. In *PPoPP*. 2013.
[4] P. Ferragina and G. Manzini. “Opportunistic data structures with applications”. In *FOCS*. 2000.
[5] L. Ferres et al. “Parallel Construction of Succinct Trees”. In *SEA*. 2015.
[6] J. Fuentes-Sepulveda et al. “Efficient Wavelet Tree Construction and Querying for Multicore Architectures”. In *SEA*. 2014.
[7] S. Gog et al. “From Theory to Practice: Plug and Play with Succinct Data Structures”. In *SEA*. 2014.
[8] R. Grossi et al. “High-order entropy-compressed text indexes”. In *SODA*. 2003.
[9] S. Hayashi and K. Taura. “Parallel and memory-efficient Burrows-Wheeler transform”. In *Big Data*. 2013.
[10] R. Homann et al. “mkESA: enhanced suffix array construction tool”. In *Bioinformatics* (2009).
[11] H. Itoh and H. Tanaka. “An efficient method for in memory construction of suffix arrays”. In *SPIRE*. 1999.
[12] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
[13] J. Kärkkäinen and P. Sanders. “Simple linear work suffix array construction”. In *ICALP*. 2003.
[14] J. Kärkkäinen et al. “Parallel External Memory Suffix Sorting”. In *CPM*. 2015.
[15] F. Kulla and P. Sanders. “Scalable parallel suffix array construction”. In *Parallel Computing* 33.9 (2007).
[16] J. Labeit. *Parallel Lightweight Wavelet Tree, Suffix Array and FM-Index Construction*. B.S. Thesis. Karlsruhe, Germany, 2015.
[17] N. J. Larsson and K. Sadakane. “Faster suffix sorting”. In *Theor. Comput. Sci.* (2007).
[18] X. Liu et al. “PASQUAL: Parallel techniques for next generation genome sequence assembly”. In *TPDS* (2013).
[19] Y. Liu et al. “Parallel and Space-efficient Construction of Burrows-Wheeler Transform and Suffix Array for Big Genome Data”. In *TCBB* (2015).
[20] V. Mäkinen and G. Navarro. “Succinct suffix arrays based on run-length encoding”. In *CPM*. 2005.
[21] U. Manber and G. Myers. “Suffix arrays: a new method for on-line string searches”. In *SIAM Journal on Computing* 22.5 (1993).
[22] G. Navarro. “Wavelet trees for all”. In *Journal of Discrete Algorithms* 25 (2014).
[23] G. Navarro and V. Mäkinen. “Compressed full-text indexes”. In *ACM CSUR* 39.1 (2007).
[24] G. Nong et al. “Linear suffix array construction by almost pure induced-sorting”. In *DCC*. 2009.
[25] E. Ohlebusch et al. “Computing the Burrows-Wheeler transform of a string and its reverse in parallel”. In *Journal of Discrete Algorithms* 25 (2014).
[26] V. Osipov. “Parallel suffix array construction for shared memory architectures”. In *SPIRE*. 2012.
[27] S. J. Puglisi et al. “A taxonomy of suffix array construction algorithms”. In *ACM CSUR* 39.2 (2007).
[28] J. Shun. “Parallel wavelet tree construction”. In *DCC*. 2015.
[29] J. Shun et al. “Brief announcement: the Problem Based Benchmark Suite”. In *SPAA*. 2012.
[30] S. Vigna. “Broadword implementation of rank/select queries”. In *WEA*. 2008.
[31] L. Wang et al. “Fast Parallel Suffix Array on the GPU”. In *Euro-Par*. 2015.