Theoretical Computer Science 812 (2020) 123–136

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Parallel computation of the Burrows Wheeler Transform in compact space [☆,☆☆]

José Fuentes-Sepúlveda [a,*], Gonzalo Navarro [a], Yakov Nekrich [b]

[a] Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
[b] Department of Computer Science, Michigan Technological University, USA

ARTICLE INFO

Article history:
Received 18 January 2019
Received in revised form 16 September 2019
Accepted 18 September 2019
Available online 24 September 2019

Keywords:
Burrows-Wheeler Transform
Suffix arrays
Parallel construction
Compact data structures

ABSTRACT

The Burrows-Wheeler Transform (BWT) has become since its introduction a key tool for representing large text collections in compressed space while supporting indexed searching: on a text of length n over an alphabet of size σ, it requires $O(n \lg \sigma)$ bits of space, instead of the $O(n \lg n)$ bits required by classical indexes. A challenge for its adoption is to build it within $O(n \lg \sigma)$ bits as well. There are some sequential algorithms building it within this space, but no such a parallel algorithm. In this paper we present a PRAM CREW algorithm to build the BWT using $O(n\sqrt{\lg n})$ work, $O(\lg^3 n / \lg \sigma)$ depth, and $O(n \lg \sigma)$ bits.
© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Since its introduction, the *Burrows-Wheeler Transform* (BWT) [2] has become the core of several data structures for text compression and indexing, with applications to bioinformatics and text searching [3–7]. For example, it is the kernel component of widely used bioinformatics software like `BowTie`[1] and `BWA`.[2]

A classical index supporting various searches on a sequence $S[1..n]$ is the suffix array $SA[1..n]$ [8], which stores the pointers to the n suffixes $S[i..n]$ in ascending lexicographic order. The BWT $B[1..n]$ of S is then defined as $B[i] = S[SA[i]-1]$, assuming $S[0] = S[n]$. Note that, on an alphabet of size σ, both S and B can be stored in $n \lg \sigma$ bits each (lg is the logarithm with base 2 by default), whereas SA requires $n \lg n$ bits in plain form. Compact text indexes use B as a replacement of both S and SA, thus significantly reducing space requirements. This permits, in particular, using the BWT to index large text collections in main memory, in situations where the SA would require resorting to the much slower secondary storage.

It is easy to compute the BWT from its definition by first building SA, which can be done in linear time and space, and efficiently in practice (e.g., [9]). However, this requires at least $n \lg n$ bits of intermediate space to end up with a smaller data structure, whose main advantage is, precisely, that it can fit in main memory whenever the $n \lg n$ bits of SA do not. In this article we aim at *compact space* constructions, that is, using working space proportional to the final BWT, $O(n \lg \sigma)$ bits.

☆ The results of Section 7 appeared in the Data Compression Conference (DCC 2019) [1].
☆☆ The first author received funding from FONDECYT grant 3170534. The first and second authors received funding from Basal Funds FB0001, CONICYT, Chile.
* Corresponding author.
E-mail addresses: jfuentess@dcc.uchile.cl (J. Fuentes-Sepúlveda), gnavarro@dcc.uchile.cl (G. Navarro), yakov.nekrich@googlemail.com (Y. Nekrich).
[1] bowtie-bio.sourceforge.net.
[2] bio-bwa.sourceforge.net.

https://doi.org/10.1016/j.tcs.2019.09.030
0304-3975/© 2019 Elsevier B.V. All rights reserved.

Several algorithms exist for the direct computation of the BWT, without building SA as an intermediate structure. Kärkkäinen [10] showed how to build it in $O(n\lg n+\sqrt{v}n)$ average time ($O(n\lg n+vn)$ in the worst case) and $O(n\lg n/\sqrt{v})$ bits of space in addition to S, for any $v\in[3,n^{2/3}]$. Hon et al. [11] built it in $O(n\lg n)$ time and within $O(n\lg\sigma)$ bits of space; the time was later improved to $O(n\lg\lg\sigma)$ [12]. Okanohara and Sadakane [13] obtained linear time $O(n)$, yet using slightly more space, $O(n\lg\sigma\lg\lg_\sigma n)$ bits. Belazzougui [14] obtained $O(n)$ randomized time within $O(n\lg\sigma)$ bits of space, making the time linear worst-case in the extended version [15]. The recent work of Munro et al. [16] also achieves linear $O(n)$ worst-case time within $O(n\lg\sigma)$ bits of space.

In the parallel computation scenario, however, there are no space-efficient algorithms for the computation of the BWT. Edwards and Vishkin [17] introduced a parallel algorithm based on the computation of the suffix tree. Using the parallel suffix tree algorithm of Sahinalp and Vishkin [18], the algorithm of Edwards and Vishkin is a CRCW PRAM algorithm with $O(n+\sigma^{2\lg^* n})$ work, $O(\lg^2 n)$ depth, and $O(n^2+\sigma^{2\lg^* n})$ bits of working space. If instead, the algorithm of Hariharan [19] is used to compute the suffix tree, the algorithm of Edwards and Vishkin is a CREW PRAM algorithm with $O(n\lg\sigma)$ work, $O(\lg^4 n)$ depth, and $O(n\lg n)$ bits of working space. Hayashi and Taura [20] presented an algorithm with $O(n\lg^3 n)$ work, $O(\lg\sigma\lg^5 n)$ depth and $O(n\lg\sigma+n\lg\lg n)$ bits of working space in the CREW PRAM model. Alternatively, parallel algorithms for the computation of the SA spending $O(n\lg n)$ bits can be used, and then compute the BWT in $O(n)$ further work and $O(1)$ depth. In this context, the CREW PRAM algorithm of Labeit et al. [21] computes the SA in $O(n)$ work and $O(\sigma^2\lg n+\sigma\lg^2 n)$ depth, for $\sigma\le\sqrt{n/\lg n}$. The algorithm of Kärkkäinen et al. [9] can compute the SA in $O(n\sqrt{\lg n})$ work and $O(\lg^2 n)$ depth under the EREW PRAM model, and in $O(n)$ randomized work and $O(\lg^2 n)$ depth under the priority CRCW PRAM model for constant-size alphabets.

The design of space-efficient parallel algorithms for the construction of compact data structures is challenging. Common design patterns for parallel algorithms must store much temporary data simultaneously, which increases the working space. In general, reducing such temporary space implies reducing the degree of parallelism. Further, accessing several items compacted in a single machine word outrules EREW models of computation. In this work, we show that the space-efficient BWT construction algorithm of Munro et al. [16] is amenable to parallelization. Our main contribution is summarized as follows.

Theorem 1. *There exists a CREW PRAM algorithm to compute the* BWT *of a sequence* $S[1..n]$ *over alphabet* $[1..\sigma]$ *with* $O(n\sqrt{\lg n})$ *work and* $O(\lg^3 n/\lg\sigma)$ *depth, using* $O(n\lg\sigma)$ *bits of working space.*

We note that $O(n\sqrt{\lg n})$ is the best known work of a parallel sorting algorithm [22].[3] Since sorting (with $\sigma=n$) can be reduced to a BWT computation, improving the work of our solution requires an improvement in parallel sorting. It is, however, possible to do less work for smaller values of σ. For example, the algorithm of Edwards and Vishkin [17] (using Hariharan [19] to compute the suffix tree so as to avoid using quadratic space), does $O(n\lg\sigma)$ work. Our solution, however, has lower depth and, what is the point in this article, does not require $\Theta(n\lg n)$ bits of working space.

The algorithm of Hayashi and Taura [20] is the only previous one that uses $o(n\lg n)$ bits when $\lg\sigma=o(\lg n)$. Still, it uses more than $O(n\lg\sigma)$ bits on very small alphabets, $\lg\sigma=o(\lg\lg n)$, and its work and depth are much higher than those of our algorithm.

Notice that our algorithm is interesting for the case $\lg\sigma=o(\lg n)$. Otherwise, no algorithm can use $o(n\lg n)$ bits of space, and in this scenario the EREW SA construction algorithm of Kärkkäinen et al. [9] (as we modify it in this article) is a better alternative. This algorithm does the same $O(n\sqrt{\lg n})$ work as ours, but its depth is just $O(\lg^2 n)$. Our depth, $O(\lg^3 n/\lg\sigma)$, smoothly approaches $O(\lg^2 n)$ as σ increases and reaches n^ε for a constant ε.

Finally, we show that our construction can be practical. We implement a simplified version of our algorithm in the dynamic multithreading model [24], and evaluate it experimentally on a dataset of randomly generated sequences over alphabets of various sizes, comparing it with other implemented parallel and sequential BWT construction algorithms. The results show that our implementation is indeed the one using the least space, while being competitive in time.

2. Preliminaries

Let $S=S[1..n]$ be a sequence of length $|S|=n$ over an alphabet $\Sigma=[1..\sigma]$, $\sigma\le n$. We use the notation $S[1],S[2],\ldots,S[n]$ to represent the symbols of S; $S[i..j]$ defines the subsequence $S[i],S[i+1],\ldots,S[j]$. This is a prefix of S if $i=1$ and a suffix if $j=n$. We use the same notation for arrays.

Given a sequence S, operation $rank_c(S,i)$ reports the number of times symbol c occurs in the prefix $S[1..i]$, $select_c(S,i)$ gives the position of the ith occurrence of c in S, and $insert_c(S,i)$ moves all the symbols in the suffix $S[i..n]$ one position to the right and inserts c at $S[i]$.

The suffix array $\mathrm{SA}[1..n]$ of a sequence $S[1..n]$ is a permutation of $[1..n]$ so that, for all $1\le i<n$, $S[\mathrm{SA}[i]..n]<S[\mathrm{SA}[i+1]..n]$ in lexicographic order [8]. To make lexicographic comparison of suffixes well defined, it is assumed that S ends with a special terminator character $S[n]=\$$ that is smaller than all the others. The BWT $B[1..n]$ of S is a reordering of the symbols of S defined as $B[i]=S[\mathrm{SA}[i]-1]$, taking $S[0]=S[n]$.

[3] It is possible to sort with $O(n)$ work, but then the depth is $O(n^\varepsilon)$ for some constant $0<\varepsilon<1$ [23, Thm. 6.1]. We are interested in polylogarithmic depths in this work.

For the analysis of our algorithm, we use the RAM model of computation, with a machine word of $\Theta(\lg n)$ bits supporting the typical operations.

Our parallel algorithm is analyzed in the *work-depth model* [25], where *work* W is the total number of operations of the parallel computation, or equivalently, the total running time using only one processor, and *depth* D is the number of time steps needed to complete the parallel computation. The measure of depth can be interpreted as the best running time we can obtain, no matter the number of available processors. Using an optimal scheduling algorithm, such as the algorithm of Brent [26], the running time using P processors on a PRAM machine can be bounded by $O(W/P + D)$. In this work we assume a shared memory model with concurrent reads and exclusive writes (PRAM CREW).

3. The sequential algorithm

Our parallel algorithm corresponds to a parallelization of the sequential space-efficient algorithm of Munro et al. [16]. The sequential algorithm divides the input sequence S into subsequences of size $\Delta = \lceil \lg_\sigma n \rceil$, constructing the BWT B in Δ steps. They add enough \$ symbols to make n divisible by Δ. For the first and second steps, the algorithm concatenates the sequences S_1 and S_2, where S_j is obtained by rotating S j symbols to the right. The concatenation $S_1 \circ S_2$ is represented as a sequence of $2n/\Delta$ meta-symbols over a meta-alphabet of size σ^Δ, by grouping Δ consecutive symbols of $S_1 \circ S_2$. Then, the algorithm computes the suffix array of $S_1 \circ S_2$ in linear time and space using the algorithm of Kärkkäinen et al. [9]. The resulting suffix array is equivalent to the suffix array of the suffixes of S starting at positions $i\Delta$ and $i\Delta - 1$, with $1 \le i \le n/\Delta$. Considering the ordering in the suffix array, the symbols at positions $i\Delta - 1$ and $i\Delta - 2$ are inserted in B. Two additional arrays, W and *Acc*, are needed to store some tracking information. The array W stores the position of the suffixes $i\Delta - 1$ in B ($i\Delta - j + 1$, in the jth step) and $Acc[a]$ stores the number of occurrences of symbols $c < a$ in B. For the remaining steps $j = 3, 4, \ldots, \Delta$ of the algorithm, the symbols $S[i\Delta - j]$ are inserted at positions $p_i = Acc[a] + rank_a(B, W[i]) + c_i$, where $a = S[i\Delta - j + 1]$ and c_i corresponds to the number of S_1 suffixes appearing before the suffix $S[i\Delta - j + 1..n]$ in the suffix array of $S_1 \circ S_j$. On each step, n/Δ new symbols are inserted in B. In order to obtain linear time, the batches of n/Δ *rank* and *insert* operations are done in $O(n/\Delta)$ time, spending $O(1)$ amortized time per operation. For $\sigma < \lg^4 n$, this is achieved easily by using the dynamic multiary wavelet tree of Navarro and Nekrich [27], with arity $\Theta(\lg^\varepsilon n)$, $0 < \varepsilon < 1$. For $\sigma \ge \lg^4 n$, a more sophisticated solution is used (see Section 6). At the end of each step, the arrays W and *Acc* are updated considering the new inserted symbols. Once the Δth step is completed, array B is the BWT of S.

Fig. 1 shows the execution of the algorithm for the DNA sequence $S = GATCAATGAGGTGGACACCAGAGGCGGTG$. In this particular example, $n = 30$, $\sigma = 5$ (including the final \$ symbol) and $\Delta = \lceil \lg_5 30 \rceil = 3$. In the figure, the indexes of the suffixes of S_2 and S_3 are underlined in $\mathrm{SA}(S_1 \circ S_2)$ and $\mathrm{SA}(S_1 \circ S_3)$, respectively.

4. The parallel algorithm

We use the following notation in the description of our parallel algorithm. Given a meta-symbol M, operations $\textsc{getLm}(M)$ and $\textsc{getRm}(M)$ obtain the leftmost and rightmost character of M, respectively. These operations can be computed in constant time using standard bit-wise operations. For example, for the meta-symbol GAC, we have $\textsc{getLm}(GAC) = G$ and $\textsc{getRm}(GAC) = C$.

Algorithm 1 shows the general idea of our parallel algorithm. Like the sequential algorithm, the parallel algorithm proceeds in $\Delta = \lceil \lg_\sigma n \rceil$ steps, and in each step it inserts $n_{ms} = \lceil n/\Delta \rceil$ new symbols to the BWT B. The algorithm allocates two arrays, $W[1..n_{ms}]$ and $Acc[1..\sigma]$ (lines 3 and 4). W is an array of pairs, where the first and second components are referred to as $W[i].f$ and $W[i].s$, respectively. Given the ith meta-symbol M_i at step $j \ge 3$, $W[i].f$ stores the position where $\textsc{getRm}(M_i)$ was inserted in B in step $j - 1$, and $W[i].s = i$. During the execution of our algorithm, W will be sorted by the first component in order to perform the batches of *rank* and *insert* operations, and by the second component to restore the order among the meta-symbols. $Acc[a]$ stores the number of occurrences of symbols $c < a$ in B. At the beginning, we allocate the $n \lg \sigma$ bits of the output BWT B (line 5). To simplify the notation, we use $S_1 \circ S_j$ to represent the concatenation of the rotations S_1 and S_j, yet such rotations are not stored explicitly. Instead, any rotation of S can be simulated, and all the extra space needed is that to prepend the last Δ symbols of S at its beginning, that is, $\Delta \lg \sigma$ extra bits. Thus, at any step j, we can recover the ith meta-symbol of S_j by reading Δ consecutive symbols of S starting at position $(i-1)\Delta - j$, with $1 \le i \le \frac{n}{\Delta}$ and $1 \le j \le \Delta$.

Steps 1 and 2 are processed together (lines 6–15). First we reduce the meta-alphabet of size σ^Δ to an alphabet of size $\frac{2n}{\Delta}$ (line 6). With that aim, we store a temporary array $R[1..\frac{2n}{\Delta}]$ of $O(\frac{n}{\Delta} \lg \frac{n}{\Delta})$ bits, with $R[i] = i$, to represent the indexes of the concatenation. Instead of sorting an array of meta-symbols, we sort the indexes representing them in R, using the meta-symbols only for the comparisons. The sorting can be done in parallel with $O(W_{sort}(\frac{n}{\Delta}))$ work and $O(D_{sort}(\frac{n}{\Delta}))$ depth, taking as input the array R and the meta-symbols, and returning the sorted indexes of the array R. With the sorted indexes, we replace in parallel the meta-symbols by their new representations in the new alphabet, obtaining the sequence T_c. Each meta-symbol is replaced by the position of its index in the sorted array. In the case of indexes representing equal meta-symbols, we take the smallest position among those indexes. For that, we perform a left-to-right scanning over the array R propagating such positions through the range of indexes representing the same meta-symbol. The left-to-right scanning can be implemented in parallel using a parallel prefix sum algorithm [25] without extra space. Since the meta-symbols are not explicitly stored, we only use $O(\frac{n}{\Delta} \lg \frac{n}{\Delta})$ extra bits to store the array R and the output sequence T_c. Thus, the new alphabet

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$S=$	G	A	T	C	A	A	T	G	A	G	G	T	G	G	A	C	A	C	C	A	G	A	G	G	C	G	G	T	G	$

$\Sigma = \{\$, A, C, G, T\}$ $\quad\Delta = \lceil \lg_5 30 \rceil = 3$

Steps 1 and 2

	1	2	3	4	5	6	7	8	9	10
$S_1 =$	$GA	TCA	ATG	AGG	TGG	ACA	CCA	GAG	GCG	GTG
$S_2 =$	G$G	ATC	AAT	GAG	GTG	GAC	ACC	AGA	GGC	GGT

$\mathrm{SA}(S_1 \circ S_2) =$ 1, $\underline{13}$, 6, $\underline{17}$, $\underline{18}$, 4, $\underline{12}$, 3, 7, $\underline{11}$, $\underline{16}$, 8, $\underline{14}$, 9, $\underline{19}$, $\underline{20}$, 10, $\underline{15}$, 2, 5

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$B=$	G	C	G	C	C	G	G	A	A	T	G	A	T	G	A	C	G	G	A	G

$W[1] = 10$ $\quad W[2] = 7$ $\quad W[3] = 2$ $\quad W[4] = 13$ $\quad W[5] = 18$

$W[6] = 11$ $\quad W[7] = 4$ $\quad W[8] = 5$ $\quad W[9] = 15$ $\quad W[10] = 16$

$Acc[\$] = 0$ $\quad Acc[A] = 0$ $\quad Acc[C] = 5$ $\quad Acc[G] = 9$ $\quad Acc[T] = 18$

Step 3

	1	2	3	4	5	6	7	8	9	10
$S_3 =$	TG$	GAT	CAA	TGA	GGT	GGA	CAC	CAG	AGG	CGG

$\mathrm{SA}(S_1 \circ S_3) =$ 1, 6, $\underline{19}$, 4, 3, $\underline{13}$, $\underline{17}$, $\underline{18}$, 7, $\underline{20}$, 8, $\underline{12}$, 9, $\underline{16}$, $\underline{15}$, 10, 2, $\underline{11}$, $\underline{14}$, 5

$c_1 = 9,\ c_2 = 6,\ c_3 = 4,\ c_4 = 9,\ c_5 = 7,\ c_6 = 7,\ c_7 = 4,\ c_8 = 4,\ c_9 = 2,\ c_{10} = 5$

Insert $S[27] = G$ at position $Acc[T] + rank_T(B, W[1]) + c_1 = 18 + 1 + 9 = 28$
Insert $S[30] = \$$ at position $Acc[G] + rank_G(B, W[2]) + c_2 = 9 + 4 + 6 = 19$
Insert $S[3] = T$ at position $Acc[C] + rank_C(B, W[3]) + c_3 = 5 + 1 + 4 = 10$
Insert $S[6] = A$ at position $Acc[T] + rank_T(B, W[4]) + c_4 = 18 + 2 + 9 = 29$
Insert $S[9] = A$ at position $Acc[G] + rank_G(B, W[5]) + c_5 = 9 + 8 + 7 = 24$
Insert $S[12] = T$ at position $Acc[G] + rank_G(B, W[6]) + c_6 = 9 + 5 + 7 = 21$
Insert $S[15] = A$ at position $Acc[C] + rank_C(B, W[7]) + c_7 = 5 + 2 + 4 = 11$
Insert $S[18] = C$ at position $Acc[C] + rank_C(B, W[8]) + c_8 = 5 + 3 + 4 = 12$
Insert $S[21] = G$ at position $Acc[A] + rank_A(B, W[9]) + c_9 = 0 + 4 + 2 = 6$
Insert $S[24] = G$ at position $Acc[C] + rank_C(B, W[10]) + c_{10} = 5 + 4 + 5 = 14$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$B=$	G	C	G	C	C	G	G	G	A	T	A	C	A	G	T	G	A	T	$	G	T	A	C	A	G	G	A	G	A	G

$W[1] = 28$ $\quad W[2] = 19$ $\quad W[3] = 10$ $\quad W[4] = 29$ $\quad W[5] = 24$

$W[6] = 21$ $\quad W[7] = 11$ $\quad W[8] = 12$ $\quad W[9] = 6$ $\quad W[10] = 14$

$Acc[\$] = 0$ $\quad Acc[A] = 1$ $\quad Acc[C] = 9$ $\quad Acc[G] = 14$ $\quad Acc[T] = 26$

Fig. 1. Execution of the algorithm of Munro et al. [16] on the sequence $S = GATCAATGAGGTGGACACCAGAGGCGGTG$.

can be computed in $O(W_{sort}(\frac{n}{\Delta}))$ work and $O(D_{sort}(\frac{n}{\Delta}))$ depth, since the parallel sorting is the most costly operation. We then compute the suffix array of T_c (line 7) and insert the first $2n_{ms}$ symbols in B by extracting the rightmost symbol of each meta-symbol. More precisely, the rightmost symbol of the meta-symbol $T_c[\mathrm{SA}[i] - 1]$ is written in $B[i]$, $1 \le i \le 2n_{ms}$ (line 11). Special cases are managed in line 10. The position of the symbols coming from S_2 are stored in W (lines 12–14). Note that the meta-symbols of T_c can be processed independently. The array Acc is filled with the accumulated frequency of the symbols (line 15). This can be done in parallel by performing a parallel sorting over a temporary copy of B and then scanning the sorted array, using a parallel prefix sum algorithm, to find the ranges of equal symbols. Thus, steps 1 and 2 take $O(W_{sort}(\frac{n}{\Delta}) + W_{SA}(\frac{n}{\Delta}))$ work and $O(D_{sort}(\frac{n}{\Delta}) + D_{SA}(\frac{n}{\Delta}))$ depth, where $W_{SA}(\frac{n}{\Delta})$ and $D_{SA}(\frac{n}{\Delta})$ correspond to the work and depth of the parallel suffix array algorithm used in line 7.

In the remaining $\Delta - 3$ steps, the algorithm computes the position in B of n_{ms} new symbols (lines 16–28). Let j be the current step and $S_j[k]$ be the kth meta-symbol of the rotation S_j. The position of the symbol $\textsc{getRm}(S_j[k-1])$ is $Acc[a] + rank_a(B, W[k]) + c_k$, where $a = \textsc{getLm}(S_j[k])$ and c_k is the number of suffixes of S_1 appearing before the suffix starting at $S_j[k]$ in the suffix array of $S_1 \circ S_j$. $Acc[a]$ is already stored. The values $rank_a(B, W[k])$ and c_k, for all $1 \le k \le n_{ms}$, are computed in lines 18 and 22–24, respectively. For the computation of the *rank* operations, we first sort them by their indexes (line 17), and then we use algorithm BatchRank from Sections 5 and 6 to answer the batch of $\frac{n}{\Delta}$ rank operations (line 18). To save space, the answer of each *rank* is stored in the first component of the array W. In lines 19–21, the outputs of the *rank* queries are added to the corresponding values of Acc. To compute the c_k values, the corresponding alphabet of $S_1 \circ S_j$ and the equivalent sequence T_c are computed, in the same way as for steps 1 and 2 (line 22). Then, in line 23, the suffix array of T_c is computed. Finally, a parallel prefix sum is performed over the suffix array counting only the suffixes of S_1 (line 24). The outputs are added to the output of the *rank* queries using the first component of W. Once the first positions of W contain the final positions of the new symbols, W is sorted again by first position $W[\cdot].f$ (line 25) and then inserted in B using algorithm BatchInsert from Sections 5 and 6 (line 26). The frequency array Acc is updated after all the symbols are inserted (line 27), by performing a parallel sorting over the new symbols, as

Algorithm 1: Parallel computation of the BWT.

```
   Input   : Sequence S of length n and alphabet size σ
   Output  : Burrows-Wheeler Transform B of S
 1 Δ ← lg_σ n
 2 n_ms ← ⌈n/Δ⌉
 3 W[1..n_ms] is an array of pairs
 4 Acc[1..σ] is an array of integers
 5 B[1..n] is the output BWT
   // Steps 1 and 2
 6 T_c ← contAlphabet(S_1 ∘ S_2)
 7 SA ← SA(T_c)
 8 parfor i ← 1 to 2n_ms do
 9 |   j ← SA[i] − 1
   |   // To avoid special cases, assume that T_c behaves as a circular sequence
10 |   if SA[i] = 1 or SA[i] = n_ms + 1 then  j ← j + n_ms
11 |   B[i] ← getRm(T_c[j])
12 |   if SA[i] > n_ms then
13 |   |   W[SA[i] − n_ms].f ← i
14 |   |   W[SA[i] − n_ms].s ← SA[i] − n_ms
15 countSymbols(B, Acc)
   // Steps 3 until Δ
16 for j ← 3 to Δ do
17 |   pSort1(W)
18 |   batchRank(B, W) // Writes the output in W[·].f
19 |   parfor i ← 1 to n_ms do
20 |   |   s ← getLm(S_j[W[i].s])
21 |   |   W[i].f ← W[i].f + Acc[s]
22 |   T_c ← contAlphabet(S_1 ∘ S_j)
23 |   SA ← SA(T_c)
24 |   countS_1(SA, W)
25 |   pSort1(W)
26 |   batchInsert(B, S_j, W)
27 |   updateAcc(Acc)
28 |   pSort2(W)
29 return B
```

done in the steps 1 and 2 of the algorithm. Finally, the new symbols and their positions are restored to their original positions, according to the ordering of the meta-symbols in S_j. To do that, the algorithm sorts the entries of W by their second component (line 28). This last sorting is necessary to maintain the relative order of the meta-symbols across all the steps. Thus, each of the remaining $\Delta - 2$ steps takes $O(W_{sort}(\frac{n}{\Delta}) + W_{rank}(\frac{n}{\Delta}) + W_{SA}(\frac{n}{\Delta}) + W_{ins}(\frac{n}{\Delta}) + \frac{n}{\Delta})$ work and $O(D_{sort}(\frac{n}{\Delta}) + D_{rank}(\frac{n}{\Delta}) + D_{SA}(\frac{n}{\Delta}) + D_{ins}(\frac{n}{\Delta}) + \lg \frac{n}{\Delta})$ depth, where $W_{rank}(\frac{n}{\Delta})$ and $D_{rank}(\frac{n}{\Delta})$ are the work and depth of the algorithm BatchRank, and $W_{ins}(\frac{n}{\Delta})$ and $D_{ins}(\frac{n}{\Delta})$ are the work and depth of the algorithm BatchInsert. The $\frac{n}{\Delta}$ and $\lg \frac{n}{\Delta}$ terms of the work and depth corresponds to the work and depth of the algorithm countS$_1$.

Analysis. Our parallel algorithm has $O(\Delta(W_{sort}(\frac{n}{\Delta}) + W_{rank}(\frac{n}{\Delta}) + W_{SA}(\frac{n}{\Delta}) + W_{ins}(\frac{n}{\Delta})) + n)$ work and $O(\Delta(D_{sort}(\frac{n}{\Delta}) + D_{rank}(\frac{n}{\Delta}) + D_{SA}(\frac{n}{\Delta}) + D_{ins}(\frac{n}{\Delta})) + \Delta \lg \frac{n}{\Delta})$ depth. For the parallel sorting we can use the EREW PRAM algorithm of Han and Shen [22], which yields $W_{sort}(\frac{n}{\Delta}) = O((n/\Delta)\sqrt{\lg n})$ and $D_{sort}(\frac{n}{\Delta}) = O(\lg n)$. Over the Δ steps, we spend $O(n\sqrt{\lg n})$ work and $O(\lg^2 n/\lg \sigma)$ depth. For the computation of the suffix array, we can use the algorithm of Kärkkäinen et al. [9]. The algorithm runs in $O((n/\Delta)\sqrt{\lg n})$ work and $O(\lg^2 n)$ depth in EREW PRAM, if we use the sorting algorithm of Han and Shen [22] as an internal subroutine. Over the Δ steps, the SA constructions take $O(n\sqrt{\lg n})$ work and $O(\lg^3 n/\lg \sigma)$ depth. In Sections 5 and 6, we present data structures to support batches of rank and insert operations, for all the $\Delta - 2$ steps, in $O(n\sqrt{\lg n})$ work and $O(\lg^2 n)$ depth. All the solutions of Sections 5 and 6 are CREW PRAM.

Our algorithm reuses the space during its Δ steps, so its memory usage is the peak of memory consumption in one step. This peak occurs in the line 23 of Algorithm 1, where the arrays W, Acc and T_c are present, and the memory peak of the SA construction occurs at the same time. The size of W is $\frac{n}{\Delta} \lg n = O(n \lg \sigma)$ bits, the size of Acc is $O(\sigma \lg n) \subseteq O(n \lg \sigma)$ bits, the size of T_c is $\frac{2n}{\Delta} \lg n = O(n \lg \sigma)$ bits, and the memory peak of the SA construction of Kärkkäinen et al. [9] is $O(n \lg \sigma)$ bits, since in the worst case it performs $\lg_{3/2} n$ recursive calls, allocating two arrays of $2(\frac{2}{3})^i \frac{n}{\Delta} \lg n$ bits in the ith recursive call. The sorting algorithm of Han and Shen, used as a subroutine of the SA algorithm and in lines 15 and 27 of Algorithm 1, assumes keys of $\Theta(\lg n)$ bits. When the sorting keys are symbols of $\lg \sigma$ bits, we can expand each symbol to use $\lg n$ bits, and still have a peak of memory consumption of $O(n \lg \sigma)$ bits, since at most $\frac{n}{\Delta}$ values are sorted.

The complexities of our algorithm are summarized in Theorem 1. The difficult part is the implementation of the operations BatchRank and BatchInsert, which is considered in the next sections.

5. Parallel processing of batches of rank/insert operations

In this section we show how to construct a data structure to support batches of $m = \Theta(n/\Delta)$ rank and insert operations using $O(n \lg \sigma)$ bits, on a sequence B of length n and alphabet $[1..\sigma]$. The structure is first built in steps 1 and 2 of Algorithm 1, over $2n/\Delta$ symbols. The batches of rank and insert operations are then used in the remaining $\Delta - 2$ steps. In this section we show how to handle the case of small alphabets, that is, $\sigma < 2^{\sqrt{\lg n}}$. The solution for the case $\sigma \geq 2^{\sqrt{\lg n}}$ is more complex and is deferred to Section 6. In all cases, the work for the whole BWT construction is $O(n\sqrt{\lg n})$, the depth is $O(\lg^2 n)$, and the space usage is $O(n \lg \sigma)$ bits. Our solution for small alphabets corresponds to a wavelet tree [28] where each operation, such as rank, takes $O(\lg \sigma)$ time. Thus, this solution is suitable only for $\sigma = 2^{O(\sqrt{\lg n})}$, so that the $O(n)$ rank operations of Algorithm 1 require work $O(n \lg \sigma) = O(n\sqrt{\lg n})$, as desired. On the other hand, our solution for large alphabets obtains $O(n \lg_\sigma n)$ work, which is the desired $O(n\sqrt{\lg n})$ when the alphabet size is $\sigma = 2^{\Omega(\sqrt{\lg n})}$.

5.1. Batches of rank/insert operations on binary sequences

Our base case are bitvectors (i.e., $\sigma = 2$). We show how a structure of $o(\ell)$ bits on top of a bitvector $B[1..\ell]$ supports m rank operations in $O(m)$ work and $O(\lg \ell)$ depth. It can also support m insertions in $O(m + \ell/\lg(\ell + m))$ work and $O(\lg n)$ depth, where $\ell + m \leq n$ on a $\Theta(\lg n)$-bits word RAM machine.

Construction. Shun [29] showed how to build the described structure on top of a bitvector $B[1..\ell]$ in $O(\ell/\lg \ell)$ work and $O(\lg \ell)$ depth on a CREW PRAM.[4]

Batch of rank operations. The data structure supports rank operations in constant time. Thus, m rank operations can be processed independently in $O(m)$ total work and $O(1)$ depth, on a CREW PRAM.

Batch of insertions. To perform the batch of insertions $Q = \{insert_{b_1}(B, i_1), \ldots, insert_{b_m}(B, i_m)\}$ in $B[1..\ell]$ we allocate space for the resulting sequence, $B'[1..\ell + m]$. The insertions come sorted by increasing index i_k and refer to the final positions after the insertions are made. These indices define ranges of entries in B to be copied to B'. In particular, the range $B[i_m - m + 1..\ell]$ is copied to $B'[i_m + 1..\ell + m]$ and the ranges $B[i_{k-1} - k + 2..i_k - k]$ are copied to $B'[i_{k-1} + 1..i_k - 1]$, for $m \geq k \geq 2$. Since the ranges do not overlap, we can make all the copies independently. Notice that several ranges may be copied into the same machine word of B', however. To avoid copying conflicts, we detect such special cases beforehand. Thus, the copying process has two stages: First, all the machine words of B' without copying conflicts are filled. Then, machine words with copying conflicts are filled, using one processor per word, so as to avoid concurrent writing. Since we know in advance the final position of the ranges of B in B', the detection of the words with copying conflicts is straightforward. Thus, the copying process is done with $O(\ell/\lg n)$ work and $O(\lg n)$ depth, since each machine word has $\Theta(\lg n)$ bits. After copying, we set $B'[i_k] = b_k$, for $1 \leq k \leq m$, which takes $O(m)$ work and $O(1)$ depth. Finally, the rank data structure of Shun [29] is rebuilt from scratch over B' in $O((\ell + m)/\lg(\ell + m))$ work and $O(\lg(\ell + m))$ depth. The peak of memory consumption is $O(\ell + m)$ bits, dominated by the space of B'.

5.2. Batches of rank/insert operations for $\sigma < 2^{\sqrt{\lg n}}$

We represent B using a wavelet tree [28]. This structure supports m ranks in $O(m \lg \sigma) \subseteq O(m\sqrt{\lg n})$ work and $O(\lg \sigma) \subseteq O(\sqrt{\lg n})$ depth, and m insertions in $O(m \lg \sigma) \subseteq O(m\sqrt{\lg n})$ work and $O(\lg n \lg \sigma) \subseteq O(\lg^{3/2} n)$ depth. The total impact in our BWT construction is $O(n\sqrt{\lg n})$ work and $O(\lg^2 n)$ depth.

Construction. We represent B with a static binary wavelet tree [28]. The root of this tree stores a bitvector B_{root} with the highest bits of the symbols of B. Its left(right) child, $B_{left}(B_{right})$ represents the subsequence of B formed by the symbols whose highest bits is a 0(1). Each child is, recursively, the root of a wavelet tree representing that subsequence, which is in turn partitioned according to the next highest bit, and so on. The wavelet tree then has height $\lg \sigma$, and it constitutes a representation of B. Accessing any symbol $B[i]$ or computing a rank on B translates into $\lg \sigma$ accesses and ranks over the bitvectors B_v associated with the wavelet tree nodes, one per level. For example, to compute $rank_c(B, i)$, we check the highest bit of c. If it is a 0, then we continue on B_{left} with position $i \leftarrow rank_0(B_{root}, i)$, otherwise we continue of B_{right} with position $i \leftarrow rank_1(B_{root}, i)$. When we reach a leaf, the answer is the value of i.

We represent the bitvectors B_v as described in Section 5.1. Shun [29] showed how to construct the whole wavelet tree of $B[1..\ell]$ with $O(W_{sort}(\ell)\lceil \lg \sigma/\sqrt{\lg \ell}\rceil)$ work, $O(D_{sort}(\ell)\lceil \lg \sigma/\sqrt{\lg \ell}\rceil + \lg \ell \lg \sigma)$ depth, and $O(\ell \lg n)$ bits, where $W_{sort}(\ell)$

[4] The algorithm needs concurrent reads because it uses lookup tables.

and $D_{sort}(\ell)$ are the work and depth of sorting ℓ integers, respectively. The algorithm uses $O(\ell \lg n)$ bits since the sorting algorithms needs keys of the size of a machine word, $\Theta(\lg n)$ bits. Thus, each symbol of $\lg\sigma$ bits must be expanded to a machine word of $\lg n$ bits. Using the $O(\ell\sqrt{\lg \ell})$-work and $O(\lg \ell)$-depth EREW PRAM integer sorting algorithm of Han and Shen [22], we obtain an $O(\ell \lg \sigma)$ work and $O(\lg \ell \lg \sigma)$ depth CREW PRAM construction algorithm for the wavelet tree of a sequence of length ℓ.

For technical reasons, we need that all the bitvectors B_v at the same depth d are stored as a single concatenated bitvector $B_d[1..\ell]$, into which the nodes v point to identify the start position O_v of their corresponding subsequence B_v. An operation $rank_b(B_v, i)$ then translates into the subtraction of two rank operations, $rank_b(B_d, i + O_v - 1) - rank_b(B_d, O_v - 1)$, which preserves the constant time. It is not hard to modify the construction of Shun [29] to build such a concatenated sequence, but it is simpler to explain how we can postprocess his original algorithm. At each wavelet tree level, a prefix sum over the lengths of the bitvectors B_v yields the starting positions O_v in B_d. Then all the bitvectors B_v are copied onto $B_d[O_v..O_v + |B_v| - 1]$, analogously as the way we support m insertions in Section 5.1. Finally, the structure for binary sequences of Section 5.1 is rebuilt on B_d. Since there may be $O(\sigma)$ wavelet tree nodes in a level (especially in the deeper ones), this requires $O(\sigma + \ell/\lg n) \subseteq O(\ell)$ work and $O(\lg \ell)$ depth per level, which preserves the total construction work and depth we already have.

We first construct the wavelet tree on the $2n/\Delta$ symbols of steps 1 and 2 of Algorithm 1. For $\sigma < 2^{\sqrt{\lg n}}$, this takes $O(\frac{n}{\Delta}\lg\sigma) = O(n\frac{\lg^2\sigma}{\lg n}) \subseteq O(n)$ work, $O(\lg\frac{n}{\Delta}\lg\sigma) \subseteq O(\lg^{3/2} n)$ depth, and $O(\frac{n}{\Delta}\lg n) = O(n\lg\sigma)$ bits of space.

Batch of rank operations. Each rank operation in the wavelet tree takes $O(\lg\sigma)$ time, as it reduces to performing $\lg\sigma$ constant-time rank operations on bitvectors. Thus, processing independently and in parallel each rank operation, we obtain $O(m\lg\sigma) \subseteq O(m\sqrt{\lg n})$ work and $O(\lg\sigma) \subseteq O(\sqrt{\lg n})$ depth to process a batch of m rank operations. Added over the $\Delta - 2$ steps of the BWT construction, this yields a total work of $O(n\sqrt{\lg n})$ and a depth of $O(\lg n)$.

Batch of insertions. To support a batch of m insertions $Q = \{insert_{c_1}(B, i_1), \ldots, insert_{c_m}(B, i_m)\}$ in the wavelet tree of B, we first extract the highest bits $b_1, \ldots, b_m$ of $c_1, \ldots, c_m$ in $O(m)$ work and $O(1)$ depth, and execute $Q_{root} = \{insert_{b_1}(B_{root}, i_1), \ldots, insert_{b_m}(B_{root}, i_m)\}$ in $O(m + \ell/\lg(m+\ell))$ work and $O(\lg n)$ depth, using the result of Section 5.1. Note that, since $m = \Theta(n/\Delta)$ and $\ell \le n$, the total work is $O(m)$.

We then distribute the operations of Q among the two children of the root, so that the left(right) child will handle the set $Q_{left}(Q_{right})$ of operations where $b_k = 0(1)$, preserving their original relative order. The separation into Q_{left} and Q_{right} is easily done with two parallel prefix sums, in $O(m)$ work and $O(\lg m)$ depth. The new insertion positions in either child are computed with $i_k \leftarrow rank_{b_k}(B_{root}, i_k)$, all of which are also computed using the result of Section 5.1 with $O(m)$ work and $O(1)$ depth.

Thus, after $O(m)$ work and $O(\lg n)$ depth, we have updated B_{root} and distributed the operations to be performed recursively on the children of the root. This continues in the same way to deeper levels. All the nodes at each level are solved independently in parallel.

To preserve the complexity in deeper levels, however, we require that all the m insertions in each level are performed together, on the concatenated bitvector $B_d[1..\ell]$. This is achieved by shifting the set of insertion positions i_k in the set Q_v corresponding to each node v in the level d by the offset $O_v - 1$, and then concatenating all the sets Q_v. This requires a parallel prefix sum on the sizes $|Q_v|$, and takes in total $O(m)$ work and $O(\lg m)$ depth. The concatenated set of insertions Q' can then be applied on B_d using the method of Section 5.1 with $O(m + \ell/\lg(m+\ell)) = O(m)$ work and $O(\lg n)$ depth.

At each level of the wavelet tree, the total work adds up to $O(m)$ and the depth is $O(\lg n)$, so the total work is $O(m\lg\sigma)$ and the depth is $O(\lg n \lg\sigma)$. Our BWT construction repeats this process $\Delta - 2$ times, which yields $O(n\lg\sigma) \subseteq O(n\sqrt{\lg n})$ total work and $O(\lg^2 n)$ depth.

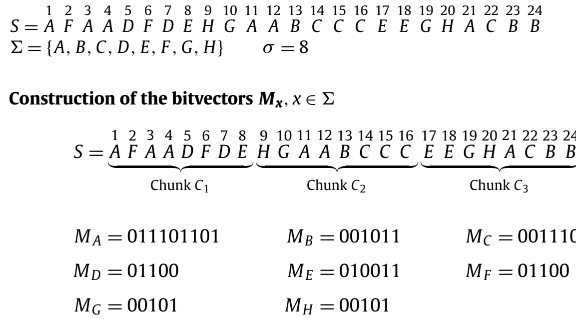
6. Batches of rank/insert operations on large alphabets

We now show how to handle batch operations when $\sigma \ge 2^{\sqrt{\lg n}}$, obtaining total work $O(n\sqrt{\lg n})$ and depth $O(\lg^2 n/\lg\sigma)$. We consider only values $\sigma \le n^{1/2}$ because, as explained, by the time we reach $\lg\sigma = \Theta(\lg n)$, compact constructions become indistinguishable from classical ones using $O(n\lg n)$ bits.

We build on a parallelization of a sequential data structure [16, App. A.3]. We first simplify the structure and then parallelize its construction and usage.

6.1. The data structure

We distribute $S[1..n]$ into a linked list of $\lceil n/\sigma\rceil$ chunks C_g of length σ. For every symbol $1 \le c \le \sigma$, a bitvector $M_c = 01^{d_1}01^{d_2}0\ldots1^{d_{\lceil n/\sigma\rceil}}$ with select support is stored, where d_g is the number of occurrences of the symbol c in the chunk C_g (cf. [30]). The sequences M_c add up to $n + \lceil n/\sigma\rceil\sigma < 2n + \sigma$ bits, and enable rank and insert operations at the chunk granularity.



Construction of the R sequences of each chunk

Chunk C_1

$$R = \underbrace{(A,1,1)(A,3,2)(A,4,3)}_{R_A}\underbrace{(D,5,1)(D,7,2)}_{R_D}\underbrace{(E,8,1)}_{R_E}\underbrace{(F,2,1)(F,6,2)}_{R_F}$$

Chunk C_2

$$R = \underbrace{(A,3,1)(A,4,2)}_{R_A}\underbrace{(B,5,1)}_{R_B}\underbrace{(C,6,1)(C,7,2)(C,8,3)}_{R_C}\underbrace{(G,2,1)}_{R_G}\underbrace{(H,1,1)}_{R_H}$$

Chunk C_3

$$R = \underbrace{(A,5,1)}_{R_A}\underbrace{(B,7,1)(B,8,2)}_{R_B}\underbrace{(C,6,1)}_{R_C}\underbrace{(E,1,1)(E,2,2)}_{R_E}\underbrace{(G,3,1)}_{R_G}\underbrace{(H,4,1)}_{R_H}$$

Fig. 2. Instance of the data structure of Section 6 for the sequence $S = AFAADFDEHGAABCCCEEGHACBB$.

To support rank/insert operations inside each chunk C, its symbols $C[i] = c$ are represented as a sequence R of triples $(c, i, rank_c(C, i))$, requiring $O(\lg\sigma)$ bits per symbol of C. The triples are sorted by symbol in R, breaking ties with their position i. Let us call R_c the range of R of the triples with symbol c.

Fig. 2 shows an instance of the data structure for the sequence $S = AFAADFDEHGAABCCCEEGHACBB$, with $\sigma = 8$.

6.2. Construction

To build the structure on a chunk $C[1..\sigma]$, we represent it as the sequence R of triples $(C[i], i, 0)$. We then sort R, first by symbol and then by position, in $O(W_{sort}(\sigma))$ work and $O(D_{sort}(\sigma))$ depth. We then find the start positions r_c of the ranges R_c, by looking where two consecutive triples of R differ in their symbols and using a parallel prefix sum, in $O(\sigma)$ work and $O(\lg\sigma)$ depth. Finally, we fill the third component of each triple $R[j] = (c, i, 0)$ with $j - r_c + 1$.

Therefore, we construct the internal data structures of a chunk of $\Theta(\sigma)$ symbols in $O(W_{sort}(\sigma) + \sigma)$ work and $O(D_{sort}(\sigma) + \lg\sigma)$ depth. For the whole sequence of length ℓ, this adds up to $O(\frac{\ell}{\sigma} W_{sort}(\sigma) + \ell)$ work and $O(D_{sort}(\sigma) + \lg\sigma)$ depth, since all the chunks can be processed at the same time. Using the integer sorting algorithm of Han and Shen [22], we obtain a CREW PRAM algorithm with $O(\ell\sqrt{\lg\sigma})$ work and $O(\lg\sigma)$ depth. Thus, the internal data structures of the chunks for the first $2n/\Delta$ symbols (after steps 1 and 2 of Algorithm 1) can be constructed in $O(\frac{n}{\Delta}\sqrt{\lg\sigma})$ work and $O(\lg\sigma)$ depth.

After the construction of the internal data structures, we build the global bitvectors M_c, $1 \le c \le \sigma$. We allocate arrays $P_c[1..\lceil\ell/\sigma\rceil]$, where we initially write in $P_c[g]$ the frequency of c in the chunk g, $r_{c+1} - r_c$. We then apply a parallel prefix sum over each array P_c simultaneously, and allocate $P_c[\lceil\ell/\sigma\rceil] + \lceil\ell/\sigma\rceil$ bits for M_c, initially filled with 1s. Finally, we set $M_c[j] = 0$, for $j = 1$ and $j = P_c[g] + g$, $1 \le g < \lceil\ell/\sigma\rceil$. This takes, overall $O(\ell)$ work and $O(\lg(\ell/\sigma))$ depth. The select data structure for all bitvectors M_c is constructed in $O(\ell)$ work and $O(\lg\lceil\ell/\sigma\rceil)$ depth [29].

The construction of M_c uses $O(\ell\lg\ell)$ bits on a sequence of length ℓ, but it will be used on the $\ell = 2n/\Delta$ symbols of steps 1 and 2 of Algorithm 1, where it will take $O(n/\Delta)$ work, $O(\lg(n/(\Delta\sigma))$ depth, and $O(\frac{n}{\Delta}\lg\frac{n}{\Delta}) = O(n\lg\sigma)$ bits.

Once the BWT construction algorithm terminates, we obtain the desired sequence from the triples of R of each chunk, by sorting them by position in total work $O(\frac{n}{\sigma}W_{sort}(\sigma)) = O(n\sqrt{\lg\sigma})$ and depth $O(D_{sort}(\sigma)) = O(\lg\sigma)$ [22].

6.3. Batch of rank operations

Let $Q = \{rank_{c_1}(B, i_1), \ldots, rank_{c_m}(B, i_m)\}$ be the m rank queries to be solved, already sorted by index. The answer of each rank query has two components, $rank_{c_k}(B, i_k) = r_{k,1} + r_{k,2}$. The first counts the number of c_ks preceding i_k before its chunk, and the second the c_ks up to i_k within its chunk.

Batch of queries Q

$$Q = \{rank_A(S,3),\ rank_A(S,11),\ rank_C(S,14)\}$$

Assignation per chunk

$$Q_1 = \{(A,3,0)\} \qquad Q_2 = \{(A,3,0),\ (C,6,0)\}$$

Chunk C_1: Merge of the lists R and Q_1

$$merge(R,Q_1) = (A,1,1)(A,3,2)\underline{(A,3,0)}(A,4,3)(D,5,1)(D,7,2)(E,8,1)(F,2,1)(F,6,2)$$

Chunk C_2: Merge of the lists R and Q_2

$$merge(R,Q_2) = (A,3,1)\underline{(A,3,0)}(A,4,2)(B,5,1)(C,6,1)\underline{(C,6,0)}(C,7,2)(C,8,3)(G,2,1)(H,1,1)$$

Answer queries

$$rank_A(S,3) = r_{1,1} + r_{1,2} = select_0(M_A,1) - 1 + 2 = 1 - 1 + 2 = 2$$

$$rank_A(S,11) = r_{2,1} + r_{2,2} = select_0(M_A,2) - 2 + 1 = 5 - 2 + 1 = 4$$

$$rank_C(S,14) = r_{3,1} + r_{3,2} = select_0(M_C,2) - 2 + 1 = 2 - 2 + 1 = 1$$

Fig. 3. Execution of the batch of queries $Q = \{rank_A(S,3),\ rank_A(S,11),\ rank_C(S,14)\}$ on the sequence $S = AFAADFDEHGAABCCCEEGHACBB$.

We first discover the index g_k of the chunk where each index i_k belongs. With this aim, we fill the chunk sizes in an array $A[1..O(n/\sigma)]$, perform a parallel prefix sum on A, and then merge the indices i_k of Q with A. A new parallel prefix sum counting the number of elements of A that precede each index i_k gives the values $g_k - 1$. Note that the space of A is $O((n/\sigma)\lg n) = o(n)$ bits. Note also that $|Q| = m = \Theta(n/\Delta)$ and $|A| = \lceil n/\sigma \rceil = o(m)$. Parallel prefixes and merging then take $O(m + W_{merge}(m))$ work and $O(\lg m + D_{merge}(m))$ depth.

With the chunk index g_k of each position i_k we can already compute $r_{k,1} = select_0(M_{c_j}, g_k) - g_k$, in $O(m)$ work and $O(1)$ depth. The second value is computed inside the chunk corresponding to each index. The chunks are obtained from the list of chunks by list ranking, in $O(n/\sigma)$ work and $O(\lg(n/\sigma))$ depth.

Let Q_g be the batch of t rank queries to be answered inside the gth chunk C. We represent the queries by triples $(c_k, i_k, 0)$ and sort them by symbol and then position. The third component 0 is used to distinguish the triples of Q_g from those of R, using the fact that the third component of the triples of R is greater than 0. We then merge Q_g with the sequence R, with the same sorting keys. The sorting and merging can be done in $O(W_{sort}(t) + W_{merge}(t+\sigma))$ work and $O(D_{sort}(t) + D_{merge}(t+\sigma))$ depth. By merging the sorted triples of Q_g and the sorted triples of R we find, for each triple $(c_k, i_k, 0)$, the rightmost triple (c_j, i_j, r_j) of R with $i_j \le i_k$. Thus, we define $r_{k,2} = r_j$. A left-to-right parallel scanning of $O(t+\sigma)$ work and $O(\lg(t+\sigma))$ depth is performed to find the index j for each k. The r_j value of the triple (c_j, i_j, r_j) is propagated to the right until the next triple of R, $(c_{j+1}, i_{j+1}, r_{j+1})$, is found. During the scanning of the merged list of Q_g and R, if the l-th entry corresponds to the query $(c_k, i_k, 0)$, we can compute the index k as $l - j'$, where j' is the position in the merged list of the last visited triple of R.

Therefore, across all the chunks, we can support m rank queries in $O(m + W_{sort}(m) + W_{merge}(n))$ work and $O(\lg m + D_{sort}(m) + D_{merge}(m+\sigma))$ depth, using $O(\frac{n}{\sigma}\lg n) = o(n\lg\sigma)$ bits for the temporary array A. Using the merge algorithm of Kruskal [31] and the integer sorting algorithm of Han and Shen [22], we obtain work $O(n + m\sqrt{\lg m})$ and depth $O(\lg m)$. After the $\Delta - 2$ steps of the BWT construction algorithm, with $\Delta = \lg_\sigma n$, all the batched rank queries are done in $O(n\lg_\sigma n + n\sqrt{\lg m}) \subseteq O(n\sqrt{\lg n})$ work and $O(\lg^2 n/\lg\sigma)$ depth.

Fig. 3 shows the execution for the batch of queries $Q = \{rank_A(S,3), rank_A(S,11), rank_C(S,14)\}$ over the sequence S of Fig. 2. In the top part of the figure, the queries are assigned to the corresponding chunk. In particular, Q is separated into two lists $Q_1 = \{(A,3,0)\}$ and $Q_2 = \{(A,3,0),\ (C,6,0)\}$. Notice that the second component of each triple in Q_1 and Q_2 corresponds to the relative position of each query with respect to the beginning of the chunk. In the middle part of the figure, the lists Q_1 and Q_2 are merged with their respective lists R. The triples of Q_1 and Q_2 are underlined. Finally, in the bottom part of the figure, the rank queries are answered. In particular, the rightmost triple of R with respect to the query $(A,3,0)$ is $(A,3,2)$, for the query $(A,3,0)$ is $(A,3,1)$ and for the query $(C,6,0)$ is $(C,6,1)$.

6.4. Batch of insertions

The insertion of m new symbols is solved in two stages: first, the internal data structures of the updated chunks are modified, and then, the global binary sequences M_c are updated.

Let $Q = \{insert_{c_1}(B,i_1), \ldots, insert_{c_m}(B,i_m)\}$ be the m insert operations. These are already sorted by index and the indexes correspond to the final position of the new symbols. First, the insertions are assigned to the corresponding chunks g_k, as

done for the batches of rank operations, in $O(m + W_{merge}(m))$ work and $O(\lg m + D_{merge}(m))$ depth. The only difference is that we use the positions $i_k - k + 1$ to find the current chunks where the insertions belong.

Let us now consider the operations $Q_g = \{insert_{c_1}(C, i_1), \ldots, insert_{c_t}(C, i_t)\}$ to be made on the gth chunk, C, giving final positions in the chunk.

We first allocate a temporary bitvector $V[1..|C| + t]$ full of 1s, set all the positions i_k to 0, and build a select data structure on it, with $O(\sigma + t)$ work and $O(\lg(\sigma + t))$ depth [29]. All the triples (c, i, r) in R are then updated to $(c, i', 0)$, with $i' = select_1(V, i)$, in $O(\sigma)$ work and $O(1)$ depth. The total temporary space for V over all chunks is $O(n)$ bits.

We then regard Q_g as a list of triples $(c_k, i_k, 0)$, sort it by c_k and then i_k, and merge it with R, with $O(W_{sort}(t) + W_{merge}(t + \sigma))$ work and $O(D_{sort}(t) + D_{merge}(t + \sigma))$ depth. We then recompute the indices r_c and the third components of the triples of R as done for the construction.

Summarizing, disregarding the possible overflows, updating the internal chunk structures upon m insertions requires $O(n + W_{sort}(m) + W_{merge}(n))$ work and $O(\lg n + D_{sort}(m) + D_{merge}(n))$ depth. Using the merge algorithm of Kruskal [31] and the integer sorting algorithm of Han and Shen [22], this is $O(n + m\sqrt{\lg m})$ work and $O(\lg n)$ depth. Added over the $\Delta - 2$ iterations of the BWTconstruction, this is $O(n \lg_\sigma n + n\sqrt{\lg m}) \subseteq O(n\sqrt{\lg n})$ work and $O(\lg^2 n / \lg \sigma)$ depth.

Finally, we update the global bitvectors M_c, $1 \le c \le \sigma$. With this aim, the bitvectors will be actually kept concatenated in M and we will know the starting position m_c of each M_c inside M. Therefore each operation $select_0(M_c, p)$ is replaced by $select_0(M, (c - 1)h + p) - m_c + 1$, where h is the number of chunks, since there are exactly $(c - 1)h$ 0s preceding M_c in M. With the numbers g_k of the chunk where each insertion belongs, we compute the positions $p_k = select_0(M, (c_k - 1)h + g_k)$ after which the bit 1 corresponding to the insertion (c_k, i_k) must be inserted in M. These do not yet refer, however, to final positions: several insertions may fall right after the same 0. We finally sort the positions p_k and execute the batched insertion $Q_M = \{insert_1(M, p_1 + 1), \ldots, insert_1(M, p_m + m)\}$ using the algorithm of Section 5.1.

The cost of updating the bitvectors M_c is then $O(m + n/\lg n + W_{sort}(m))$ work and $O(\lg n + D_{sort}(m))$ depth, where the $n/\lg n$ component of the work comes from the insertion algorithm of Section 5.1. Using the sorting algorithm of Han and Shen [22], we obtain $O(m\sqrt{\lg m})$ work and $O(\lg n)$ depth. Added over the $\Delta - 2$ iterations, this becomes $O(n\sqrt{\lg n})$ work and $O(\lg^2 n / \lg \sigma)$ depth.

Chunk overflows. If the t insertions in the chunk C make it longer than 2σ symbols, we split it into $\left\lceil \frac{|C|+t}{\sigma} \right\rceil$ new chunks of σ symbols (except the last one, which may be shorter). After generating the merged list R, but before recomputing its third component, we distribute it across the new chunks: the triple $(c, i, 0)$ in R is distributed to the newly created chunk number $f = \lceil i/\sigma \rceil$, and its new local offset is $i' = i - (f - 1)\sigma$. We then sort R stably by component f, which effectively distributes it into the new chunks, and finally compute the indexes r_c and third components inside each new chunk. The sorting is done in $O((t + \sigma)\sqrt{\lg(t + \sigma)}) \subseteq O((t + \sigma)\sqrt{\lg n})$ work and $O(\lg(t + \sigma))$ depth [22]. In an amortized analysis, we can charge $O(\sqrt{\lg n})$ work to each of the m inserted elements, since a block is created of size σ or less and it must receive at least other σ insertions to require a split and return to size σ or less. The total work incurred is then $O(m\sqrt{\lg n})$, which added over the $\Delta - 2$ iterations to build the BWT yields $O(n\sqrt{\lg n})$; the depth adds up to $O(\lg^2 n / \lg \sigma)$.

Note that the positions i of the triples $(c, i, 0)$ in R occupy $O(\lg(t + \sigma))$ bits before we divide the chunk. Let $t_g = |Q_g|$, then this adds up to $O(\sum_{g=1}^{\lceil n/\sigma \rceil} (t_g + \sigma) \lg(t_g + \sigma))$ bits, where $\sum_{g=1}^{\lceil n/\sigma \rceil} t_g = m$. This is in $O(\sum_{g=1}^{\lceil n/\sigma \rceil} (t_g \lg t_g + t_g \lg \sigma + \sigma \lg t_g + \sigma \lg \sigma)) \subseteq O(m \lg m + m \lg \sigma + n \lg(m\sigma/n) + n \lg \sigma) = O(n \lg \sigma)$, where we pessimistically maximized each term depending on its concavity.

After chunk C_g overflows, we must also insert some 0s in M. For each new chunk $1 \le g' < f$ we create (note we exclude the last), we use the starting positions r_c of the subsequences R_c: starting at the 0 that represents the chunk g in M_c, we must skip r_c 1s and insert a 0, consecutively for each new chunk $g - 1 + g'$. To do this in parallel, after distributing the sequence R of the overflowing chunk, we work in parallel on each symbol c, doing a parallel prefix sum on the $f - 1$ values r_c and collecting the sums in arrays $q_c[1..f - 1]$. We then generate $f - 1$ operations $insert_0(M, s + q_c[g'] + g')$, where $s = select_0(M, (c - 1)h + g)$, for all the new chunks g' and the σ symbols simultaneously. This takes $O((f - 1)\sigma)$ work and $O(\lg(f - 1))$ depth. The $(f - 1)\sigma$ insertions created on M are then sorted in $O(W_{sort}((f - 1)\sigma))$ work and $O(D_{sort}((f - 1)\sigma))$ depth, and carried out again using the algorithm of Section 5.1, in $O((f - 1)\sigma + n/\lg n)$ work and $O(\lg n)$ depth.

The total cost to update the bitvectors M_c upon overflows is then $O((f - 1)\sigma + W_{sort}((f - 1)\sigma) + n/\lg n)$ work and $O(D_{sort}((f - 1)\sigma) + \lg n)$ depth. These can be made $O((f - 1)\sigma\sqrt{\lg n} + n/\lg n)$ work and $O(\lg n)$ depth [22]. Since the sum of the $f - 1$ values across all the iterations is the final number of chunks, $O(n/\sigma)$, this adds up over the $\Delta - 2$ iterations to $O(n\sqrt{\lg n})$ work and $O(\lg^2 n / \lg \sigma)$ depth.

7. Implementation

In this section we validate the practicality of our construction by presenting an implementation of Algorithm 1 using multithreading, with p processors on a w-bit machine, and comparing it with other implemented parallel and sequential algorithms.

7.1. A practical version of the algorithm

In order to obtain a practical solution with $O(n \lg \sigma)$ bits of working space, we made the following modifications to the Algorithm 1:

Parallel sorting. We used a parallel implementation of radix sort [32] to sort W (lines 17, 25 and 28 of Algorithm 1). The smallest alphabet of size $\frac{2n}{\Delta}$ was computed with a parallel quicksort, since the cost of radix sort increases linearly with Δ. This corresponds to the traditional recursive algorithm, where the two recursive calls are executed concurrently. For the split step of quicksort, the array is divided into p equal-size segments sharing a global pivot. Then, on each segment a sequential split function is applied. Finally, the outputs of the sequential split functions are merged into one array by using prefix sums to compute the position of each entry. To compare meta-symbols of $\Delta \lg \sigma$ bits, the algorithm must read them by words of w bits, starting from the most significant bits. For large values of Δ, quicksort is faster than radix sort since it requires fewer memory accesses to decide which of two meta-symbols is smaller (reading the first word suffices most of the times). Radix sort, instead, would always execute $\Theta(\frac{\Delta}{\lg n})$ steps, so its time would not decrease with Δ. With p threads, quicksort takes $O((\frac{n}{p\Delta} + \lg p)\lg n)$ parallel average time and $2p \lg \frac{n}{\Delta}$ bits of space. The resulting average work is $O(\frac{n}{\Delta} \lg \frac{n}{\Delta})$ and the depth is $O(\lg^2 \frac{n}{\Delta})$. Notice that the parallel algorithm needs $O(\frac{n}{\Delta} \lg \sigma)$ bits to copy the elements at the end of the parallel split, but we can reuse one of the two arrays of the parallel suffix array algorithm (see next).

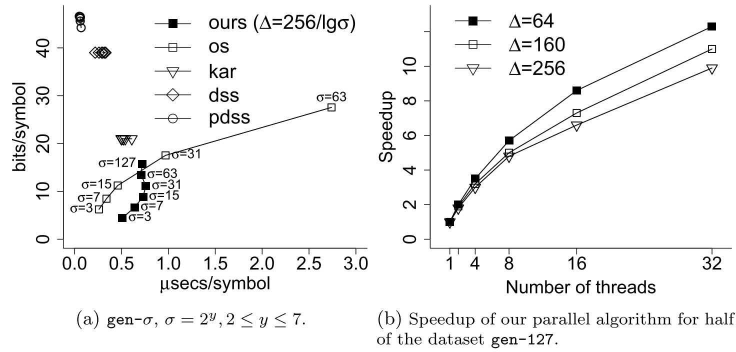
Suffix array on integers. After using the parallel quicksort to compute the new alphabet of the meta-symbols, we use the *parallel range* algorithm [32] to compute the suffix array of an array of meta-symbols. In its implementation, the parallel range algorithm uses two temporary integer arrays besides the input and output arrays, spending $2\frac{n}{\Delta} \lg \frac{n}{\Delta}$ bits of extra space, $O(\frac{n}{\Delta} \lg \frac{n}{\Delta})$ work and $O((\frac{n}{\Delta})^{\varepsilon})$ depth, for $0 < \varepsilon < 1$.

Rotations. In favor of reducing running time, we store explicitly the starting position of each meta-symbol in the input sequence. We store only the positions of the current rotation, since the first rotation S_1 can be derived from the current one. This requires $\frac{n}{\Delta} \lg n$ further bits.

Steps 1 and 2. The new alphabet (line 6) is computed with parallel quicksort. The suffix array (line 7) is built using the parallel range algorithm. The *for* loop of lines 8–14 can be parallelized straightforwardly, since each new symbol can be processed independently, giving $O(n/\Delta)$ work, $O(\lg \frac{n}{\Delta})$ depth and no extra space. Finally, for the frequency counting of line 15, B is divided into p equal-sized segments, and the frequency counting is performed concurrently in all the segments. Then, the results of the segments are aggregated by performing σ parallel prefix sums, one for each symbol. The results of the prefix sums are stored in *Acc*. At this point, *Acc* stores the frequency of each symbol in B. A final prefix sum is performed over *Acc* in order to obtain the accumulated frequency. The space used by the arrays of each segment is $O(p\sigma \lg \frac{n}{\Delta})$ bits. The work is $O(n/\Delta)$ and the depth is $O(\lg \frac{n}{\Delta})$. This analysis is also valid for the counting of suffixes of S_1 (lines 22–24).

Batch of rank operations. The batch of *rank* operations is sorted with parallel radix sort (line 17). The operations are supported by dividing the input sequence B into p subsequences of equal size. All the subsequences are concurrently traversed left-to-right using a global array F_g and a local array F_l per subsequence. The global array F_g stores the accumulated frequency in the subsequence, spending $\sigma \lg n$ bits, whereas the local array F_l stores the accumulated frequency in subsequences of $\lg^2 n$ symbols, spending $2\sigma \lg\lg n$ bits. Each subsequence is traversed left-to-right, counting in constant time the number of occurrences of all the symbols in $\frac{\lg n}{2}$ consecutive bits, by using a lookup table of $2\sqrt{n}\sigma \lg\lg n = o(n)$ bits. With the symbol frequencies from the lookup table, we update F_l in time $O(\frac{\sigma \lg\lg n}{\lg n})$, by adding various fields in parallel on a machine word of $\Theta(\lg n)$ bits. After reading $\lg^2 n$ consecutive symbols, we update the array F_g by adding to it the accumulated frequencies of F_l, in time $O(\sigma)$, and then all the entries of F_l are set to zero. Thus, the total time used to update the arrays F_l and F_g is $O(\frac{n\sigma \lg \sigma \lg\lg n}{\lg^2 n})$. During the traversal of a subsequence $B[\frac{jn}{p}+1 \ldots \frac{(j+1)n}{p}]$, each operation $rank_s(B,i)$ with $\frac{jn}{p} < i \le \frac{(j+1)n}{p}$ is partially answered by counting the number of occurrences of the symbol s in the subsequence $B[\frac{jn}{p}+1 \ldots i]$. After the traversal, the p arrays F_g contain the frequency counts of the p subsequences. We apply σ parallel prefix sums over the F_g arrays, and then correct the partial answers of the operations by adding the values of the F_g arrays. For the operation $rank_s(B,i)$, we sum the occurrences of the symbol s in the $(\lfloor \frac{(i-1)p}{n} \rfloor + 1)$-th F_g array. The space usage is $O(p\sigma \lg n)$ bits for the F_g and F_l arrays. The parallel time is $O(\frac{n\sigma \lg \sigma \lg\lg n}{p \lg^2 n})$, plus $O(\sigma)$ for the partial sums (by using $p/\lg p$ threads per partial sum and then summing $\sigma/\lg p$ groups of $\lg p$ symbols in parallel). Multiplied by the Δ rounds, this yields a work of $O(\frac{n\sigma \lg \sigma \lg\lg n}{\lg^2 n})$ and $O(\sigma\Delta)$ depth. Note that the total cost is $O(\frac{n}{\Delta})$ whenever $\sigma = O(\frac{\lg n}{\lg\lg n})$.

Batches of insertions. To support a batch of insertions we can use the solution of Section 5.1 for any σ. However, to avoid increasing the space, we chose to perform the insertions sequentially using a sequential version of the solution of Section 5.1.



(a) gen-σ, $\sigma = 2^y, 2 \leq y \leq 7$. (b) Speedup of our parallel algorithm for half of the dataset gen-127.

Fig. 4. Running time for the computation of the BWT.

Other notes. The *for* loop of lines 16–28 is carried out sequentially. Note, however, that the instructions of lines 17–21 and instructions of lines 22–23 can be executed concurrently, since they execute independent procedures. The update of *Acc* (line 27) can be done in parallel, as well as the line 15, by storing the frequency counts of the recently inserted symbols in a temporary array of σ entries and then adding them to *Acc* in parallel for each symbol.

Complexity. The average complexity of our algorithm, with p threads, is then $O(\frac{n}{p}\lg\frac{n}{\Delta} + \frac{n\Delta\sigma\lg\sigma\lg\lg n}{p\lg^2 n} + \Delta\sigma + \text{polylog}\, n)$, with an average work of $O(n\lg\frac{n}{\Delta} + \frac{n\Delta\sigma\lg\sigma\lg\lg n}{\lg^2 n} + \Delta\sigma)$. Note that we preserve a linear dependence on Δ. Such linear dependence will be evident later in our experimental study. The depth is $O(\Delta\sigma + \Delta^{1-\epsilon}n^{\epsilon})$. We exclude the batches of *inserts*, which are done sequentially to save space, adding $O(\frac{n\Delta\lg\sigma}{\lg n})$ time, but they could be done in $O(\frac{n\Delta\lg\sigma}{p\lg n})$ parallel time, $O(\frac{n\Delta\lg\sigma}{\lg n})$ work, and $O(\Delta\lg\frac{n}{\Delta})$ depth.

The peak of space consumption is $9\frac{n}{\Delta}\lg n + o(n)$ bits, to which parallelization adds $O(p\sigma\lg n)$ bits. This extra space is negligible for practical values of p. The peak of space consumption occurs in line 23 of Algorithm 1, where the $2\frac{n}{\Delta}\lg n$ bits of W, the $2\frac{n}{\Delta}\lg\frac{n}{\Delta}$ bits of T_c, the $2\frac{n}{\Delta}\lg\frac{n}{\Delta}$ bits of the output SA, the $2\frac{n}{\Delta}\lg\frac{n}{\Delta}$ bits of the suffix array computation, the $\frac{n}{\Delta}\lg n$ bits of the position of the meta-symbols in the current rotation, and the $o(n)$ bits of *Acc* are allocated at the same time.

7.2. Experimental study

Setup. We implemented our algorithm in C++ and compiled with GCC 6.3 and option -O3. The implementation is available at https://github.com/jfuentess/bwt. We compare our implementation against those of Okanohara and Sadakane [13] (OS), Kärkkäinen [10] (KAR),[5] the SA algorithm *divSufSort* implemented by Mori (DSS), and the parallel SA algorithm of Labeit et al. [21] (PDSS). We only consider one parallel algorithm, PDSS, since it is the parallel version of DSS and, as reported [21], it outperforms a parallel version of KAR. A parallelization of OS is far from trivial, since it exhibits several data dependencies. We could avoid such dependencies by storing temporary data during the computation, but that will increase the working space. All the baselines were compiled with their best optimization. We tried to include the algorithm of Hayashi and Taura [20], but had problems to compile it. Yet, according to them [20], OS is more competitive both in space and time.

The experiments ran on a machine with two Intel® Xeon® Silver 4110 Processors with 16 physical cores clocked at 2.1 GHz, with per-core L1 and L2 caches of 32 KB and 1 MB respectively, a per-processor L3 cache of 11 MB and 252 GB of DDR3 RAM memory (126 GB per NUMA node). Hyperthreading was enabled. Running time was measured with the functions in `<time.h>`. The memory consumption was measured using malloc_count (http://panthema.net/2013/malloc_count). We report the median running time of 10 repetitions. Different values of n, σ and Δ were tested. In particular, we tested $\Delta = \lfloor\frac{2^z}{\lg\sigma}\rfloor$ symbols, for $6 \leq z \leq 12$.

For the experiments, we use six random sequences gen-σ, with $n = 2^{30}$ and $\sigma \in \{3, 7, 15, 31, 63, 127\}$. Each dataset was generated with the random generator of *Pizza&Chili*, which distributes the symbols uniformly and independently. We remind that, during the computation of the BWT, a special symbol \$ is inserted, which increases the alphabet size by one.

Results. Fig. 4a shows the space/time tradeoff for the gen-σ datasets. The reported running time of the two parallel algorithms, OURS and PDSS, considers 32 threads. Using $\Delta = \lfloor\frac{256}{\lg\sigma}\rfloor$ symbols, our algorithm uses less memory than all the competitors for all values of σ. The algorithms PDSS, DSS and KAR store each symbol of the input in an 8-bit variable, allowing a maximum alphabet of size 256. This is why they are not affected by the increasing values of σ. The algorithm KAR is slightly faster than ours, but still uses much space. The fastest of all the algorithms is PDSS, but it is also the one

[5] We used the parameter $v = 7$. We tried increasing v to reduce space, but the time increases fast and makes it not competitive.

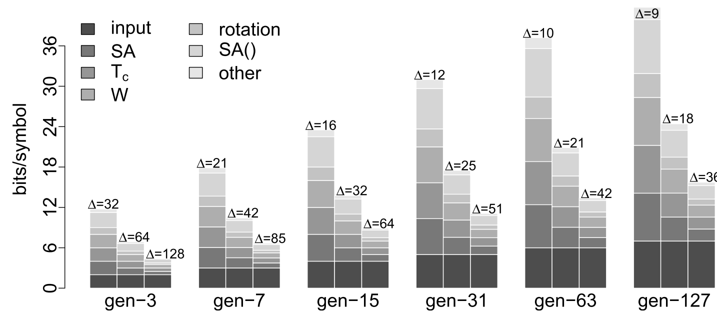


Fig. 5. Memory consumption.

with the highest memory consumption. For $\sigma \le 30$, our closer competitor is os. For those, our algorithm with $\Delta = \lfloor \frac{256}{\lg \sigma} \rfloor$ is slower than os but it is the one using the least space. For $\sigma > 30$, our algorithm dominates os both in time and space.

Fig. 4b shows the scalability of our algorithm. For 16 threads, our algorithm shows an efficiency (speedup/number of threads) of 50%, decreasing to 30% for 32 threads. The reason is that our machine has 16 physical cores, increased to 32 logical ones with hyperthreading. The speedup slightly decreases with Δ, which is in line with our analysis, where the depth increases with Δ.

Fig. 5 shows the effect of Δ and σ on the memory consumption of our algorithm. In the figure, the peak of memory does not consider the size of the output. The size of the arrays SA, T_c, W and *rotation*, and the memory usage of the suffix array algorithm, SA(), depend on Δ and σ. The segment *other* corresponds to additional arrays needed during the computation. The size of such arrays is negligible.

Since the main point is to limit the consumption of main memory, external memory constructions of the BWT and SA are an interesting alternative to consider. We leave as future work a formal comparison against external memory algorithms, such as the recent work of Kärkkäinen et al. [33] and the parallel algorithm Kärkkäinen et al. [34]

8. Conclusions

We have presented the first parallel construction of the Burrows-Wheeler Transform (BWT) that works in compact space, that is, $O(n \lg \sigma)$ bits for a sequence of length n over alphabet $[1..\sigma]$. Our construction, in the CREW PRAM model, uses $O(n\sqrt{\lg n})$ work and $O(\lg^3 n / \lg \sigma)$ depth. We have also implemented a simplified version of our construction in a multi-threading scenario, showing that it uses the least amount of space among various state-of-the-art sequential and parallel alternatives, within competitive construction time.

An improvement in the work or depth of our algorithm that holds for any σ requires improvements in the more basic problems of parallel sorting or parallel suffix array construction, respectively. It is possible, however, that better complexities may be obtained for small values of σ.

An interesting challenge is to build in parallel and in compact space the other components of a compressed suffix tree [35]: the tree topology and the (permuted) longest common prefix array. These require $O(n)$ further bits and can be built in $O(n)$ sequential time and $O(n \lg \sigma)$ bits from the BWT [14,16]. Yet, current parallel suffix tree constructions require $O(n \lg n)$ bits [18,19,36].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J. Fuentes-Sepúlveda, G. Navarro, Y. Nekrich, Space-efficient computation of the Burrows-Wheeler transform, in: Proc. 29th Data Compression Conference (DCC), 2019, pp. 132–141.
[2] M. Burrows, D.J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, Tech. Rep., Digital Systems Research Center, 1994.
[3] P. Ferragina, G. Manzini, Indexing compressed texts, J. ACM 52 (4) (2005) 552–581.
[4] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer, 2008.
[5] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013.
[6] V. Mäkinen, D. Belazzougui, F. Cunial, A.I. Tomescu, Genome-Scale Algorithm Design, Cambridge University Press, 2015.
[7] G. Navarro, Compact Data Structures – A Practical Approach, Cambridge University Press, 2016.
[8] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[9] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[10] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theor. Comput. Sci. 387 (3) (2007) 249–257.
[11] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, S.-M. Yiu, A space and time efficient algorithm for constructing compressed suffix arrays, Algorithmica 48 (1) (2007) 23–36.

[12] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a time-and-space barrier in constructing full-text indices, SIAM J. Comput. 38 (6) (2009) 2162–2178.
[13] D. Okanohara, K. Sadakane, A linear-time Burrows-Wheeler transform using induced sorting, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 90–101.
[14] D. Belazzougui, Linear time construction of compressed text indices in compact space, in: Proc. 46th ACM Symposium on Theory of Computing (STOC), 2014, pp. 148–193.
[15] D. Belazzougui, F. Cunial, J. Kärkkäinen, V. Mäkinen, Linear-time string indexing and analysis in small space, CoRR, arXiv:1609.06378, arXiv:1609.06378.
[16] J.I. Munro, G. Navarro, Y. Nekrich, Space-efficient construction of compressed indexes in deterministic linear time, in: Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2017, pp. 408–424.
[17] J.A. Edwards, U. Vishkin, Parallel algorithms for Burrows–Wheeler compression and decompression, Theor. Comput. Sci. 525 (2014) 10–22.
[18] S.C. Sahinalp, U. Vishkin, Symmetry breaking for suffix tree construction, in: Proc. 26th Annual ACM Symposium on Theory of Computing (STOC), 1994, pp. 300–309.
[19] R. Hariharan, Optimal parallel suffix tree construction, J. Comput. Syst. Sci. 55 (1) (1997) 44–69.
[20] S. Hayashi, K. Taura, Parallel and memory-efficient Burrows-Wheeler Transform, in: Proc. IEEE International Conference on Big Data, 2013, pp. 43–50.
[21] J. Labeit, J. Shun, G.E. Blelloch, Parallel lightweight wavelet tree, suffix array and FM-index construction, J. Discret. Algorithms 43 (2017) 2–17.
[22] Y. Han, X. Shen, Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write prams, SIAM J. Comput. 31 (6) (2002) 1852–1878.
[23] U. Vishkin, Thinking in parallel: some basic data-parallel algorithms and techniques, class notes, 2010.
[24] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Multithreaded algorithms, in: Introduction to Algorithms, third ed., The MIT Press, 2009, pp. 772–812.
[25] J. JáJá, An Introduction to Parallel Algorithms, Addison Wesley Longman, 1992.
[26] R.P. Brent, The parallel evaluation of general arithmetic expressions, J. ACM 21 (2) (1974) 201–206.
[27] G. Navarro, Y. Nekrich, Optimal dynamic sequence representations, SIAM J. Comput. 43 (5) (2014) 1781–1806.
[28] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
[29] J. Shun, Improved parallel construction of wavelet trees and rank/select structures, in: Proc. 27th Data Compression Conference (DCC), 2017, pp. 92–101.
[30] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. 17th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA), 2006, pp. 368–373.
[31] J. Kruskal, Searching, merging, and sorting in parallel computation, IEEE Trans. Comput. C-32 (10) (1983) 942–946.
[32] 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. 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2012, pp. 68–70.
[33] J. Kärkkäinen, D. Kempa, S.J. Puglisi, B. Zhukova, Engineering external memory induced suffix sorting, in: Proc. 19th Workshop on Algorithm Engineering and Experiments (ALENEX), 2017, pp. 98–108.
[34] J. Kärkkäinen, D. Kempa, S.J. Puglisi, Parallel external memory suffix sorting, in: Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM), 2015, pp. 329–342.
[35] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[36] U. Baier, T. Beller, E. Ohlebusch, Space-efficient parallel construction of succinct representations of suffix tree topologies, J. Exp. Algorithmics 22 (2017) 1.1.