Space-Efficient Computation of the Burrows-Wheeler Transform

José Fuentes-Sepúlveda[1,2], Gonzalo Navarro[1,2], and Yakov Nekrich[3]

[1]Center for Biotechnology and Bioengineering, University of Chile, Chile
[2]Department of Computer Science, University of Chile, Chile
[3]Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

The Burrows-Wheeler Transform (BWT) has become an essential tool for compressed text indexing. Computing it efficiently and within little space is essential for the practicality of the indexes that build on it. A recent algorithm (Munro, Navarro & Nekrich, SODA 2017) computes the BWT in $O(n)$ time using $O(n \lg \sigma)$ bits of space for a text of length n over an alphabet of size σ. The result is of theoretical nature and its practicality is far from obvious. In this paper we engineer their solution and show that, while a basic implementation is slow in practice, the algorithm is amenable to parallelization. For a wide range of alphabet sizes, our resulting implementation outperforms all the compact constructions in the space/time tradeoff map. On the smallest alphabets we are outperformed in time, but nevertheless achieve the least space within reasonable time. For example, in DNA sequences, the most widely used application of BWTs, our construction uses 4.84 bits per base and builds the BWT at a rate of 2.13 megabases per second, whereas the closest previous alternative uses around 7.09 bits per base and runs at 4.17 megabases per second.

Introduction

The *Burrows-Wheeler Transform* (BWT) [1] has become in a key tool for areas such as bioinformatics and text searching [2]. For example, it is the kernel component of `BowTie` (`bowtie-bio.sourceforge.net`) and `BWA` (`bio-bwa.sourceforge.net`).

Given a sequence $S[1..n]$ and the suffix array $\mathrm{SA}[1..n]$ of S [3], the BWT $B[1..n]$ of S is defined as $B[i]{=}S[\mathrm{SA}[i]{-}1]$, taking $S[0]{=}S[n]$. Note that, on an alphabet of size σ, either S or B can be stored in $n \lg \sigma$ bits (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.

It is easy to compute the BWT by first building SA, which can be done in linear time and space (e.g., [4–7]). However, this requires at least $n \lg n$ bits of intermediate space to build 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. Compact solutions for the BWT computation, without building SA as an intermediate structure, have been proposed. Kärkkäinen [8] showed how to build it in $O(n \lg n + \sqrt{v}n)$ average time ($O(n \lg n + vn)$ worst case) and $O(n \lg n/\sqrt{v})$ bits of space, excluding S, for $v{\in}[3, n^{2/3}]$. Hon *et al.* [9] built it in $O(n \lg n)$ time and within $O(n \lg \sigma)$ bits of space; improved later to $O(n \lg\lg \sigma)$ time [10]. Okanohara and Sadakane [11] built it in $O(n)$ time, using $O(n \lg \sigma \lg\lg_\sigma n)$ bits. Belazzougui [12] obtained $O(n)$ randomized time within $O(n \lg \sigma)$ bits of space, making the time linear worst-case in the extended version [12]. The recent work of Munro *et al.* [13] also achieves linear $O(n)$ worst-case time within $O(n \lg \sigma)$ bits of space. Their presentation, however, is theoretical and leaves open how would the construction perform in practice, both in space and time.

The first author received funding from Fondecyt grant 3170534. The first and second authors received funding from Basal Funds FB0001, Conicyt, Chile.

In this work we focus on *in-memory* algorithms for the BWT construction. Specifically, we engineer the recent algorithm of Munro *et al.* [13]. We manage to retain the theoretical complexities for the case $\sigma{=}O(\frac{\lg n}{\lg\lg n})$. Since our main focus is to use $O(n\lg\sigma)$ instead of $O(n\lg n)$ bits, we are indeed interested in small and moderate alphabet sizes (say, up to $\sigma{=}256$). Our experiments show that it is indeed possible to build the BWT within little space in practice, around $n\lg\sigma$ to $2n\lg\sigma$ bits, but a sequential implementation, even if optimized, is too slow. We show, however, that the algorithm allows for significant parallelization, developing a multithreaded version of the construction. This multithreaded version shows to be competitive for interesting cases, such as DNA sequences (the most common application of the BWT).

Preliminaries. Given a sequence $S[1..n]$ and its alphabet Σ of size σ, $rank_c(S,i)$ reports the number of occurrences of c in the prefix $S[1..i]$, whereas $insert_c(S,i)$ 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 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 [3]. To make lexicographic comparison of suffixes well defined, it is assumed that S ends with a special terminator character \$ that is smaller than the others.

By default, we assume the RAM model of computation, with a word w of $\Theta(\lg n)$ bits supporting the typical operations. For parallel algorithms, we use the *Dynamic Multithreading Model* (DYM) [14], which provides two basic measures: The *work* (T_1), which corresponds to the running time of the multithreaded computation using only one thread, and the *span* (T_∞), which can be interpreted as the best running time we can obtain, no matter the number of available threads. With both measures, the model provides an upper bound $T_p{=}O(T_1/p{+}T_\infty)$ using p threads.

Related Work

In this work we design a practical version of the algorithm of Munro *et al.* [13]. Theirs is an iterative algorithm that divides the input sequence S into subsequences of size $\Delta{=}\lg_\sigma n$, 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, copying the last j symbols of S to the beginning of S_j. The concatenation $S_1{\circ}S_2$ is represented as a sequence of $\frac{2n}{\Delta}$ 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 $S_1{\circ}S_2$ using a linear time and space algorithm [6]. The resulting suffix array is equivalent to the suffix array of the suffixes of S starting at positions $i\Delta{-}1$ and $i\Delta{-}2$, with $1{\le}i{\le}\frac{n}{\Delta}$. With the suffix array, the symbols at positions $i\Delta{-}2$ and $i\Delta{-}3$ are inserted in B. Two arrays, W and Acc, are needed to store tracking information. W stores the position of the suffixes $i\Delta{-}2$ in B ($i\Delta{-}j$, in general) and $Acc[a]$ stores the number of occurrences of symbols $c{<}a$ in B. For the remaining steps $j{=}3,4,...,\Delta$, the symbols $S[i\Delta{-}j{-}1]$ are inserted at positions $p_i{=}Acc[a]{+}rank_a(B,W[i]){+}c_i$, where $a{=}S[i\Delta{-}j]$ and c_i corresponds to the number of S_1 suffixes appearing before the suffix $S[i\Delta{-}j..n]$ in the suffix array of $S_1{\circ}S_j$. On each step, $\frac{n}{\Delta}$ new symbols are inserted in B. Batches of $\frac{n}{\Delta}$ *rank* and *insert* operations are done in $O(\frac{n}{\Delta})$ time, spending $O(1)$ amortized time per operation. On polylogarithmic-sized alphabets,

this is achieved by using the dynamic wavelet tree of Navarro and Nekrich [15]; more sophisticated solutions are used for larger σ. At the end of each step, the arrays W and Acc are updated considering the new inserted symbols. Once the Δth step is completed, the array B is the BWT of S.

Engineering the Algorithm

Rotations. In the jth step, the algorithm needs to compute the rotation S_j of S. We can simulate any rotation of S without storing it explicitly, by prepending the last Δ symbols of S at its beginning, using $\Delta\lg\sigma$ extra bits. Thus, given any jth step, 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 $1\leq i\leq\frac{n}{\Delta}$ and $1\leq j\leq\Delta$.

Suffix Array. To compute the suffix array of the rotations, we first need to reduce the meta-alphabet of size σ^{Δ} to a continuous alphabet of size $\frac{2n}{\Delta}$. For that, we store a temporal 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. The sorting is carried out in $O(\frac{n}{\epsilon\lg n})$ time and $O(n^{\epsilon}\lg\frac{n}{\Delta})$ bits using radix sort with radix n^{ϵ}, for any constant $0<\epsilon<1$ (note that we sort $O(\frac{n}{\Delta})$ symbols but need $\Theta(\frac{\Delta}{\lg n})$ passes of radix sort). After the sorting, we replace the meta-symbols by their new representations in the continuous alphabet. 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.

Batches of operations. The crux of the theoretical algorithm [13] is a (complicated) data structure to perform a batch of $O(\frac{n}{\Delta})$ *rank* or *insert* operations in time $O(\frac{n}{\Delta})$. For polylogarithmic-size alphabets, this boils down to a dynamic multiary wavelet tree [15], but this is still difficult to implement. We present a simple alternative for the case $\sigma=O(\frac{\lg n}{\lg\lg n})$. Let $Q=\{rank_{b_1}(B,i_1),...,rank_{b_m}(B,i_m)\}$ be the set of *rank* operations to be solved and let A be an array of size $m=O(\frac{n}{\Delta})$ where the answers will be stored. The set Q comes already sorted by increasing indexes. We allocate two arrays, F_l and F_g, of $2\sigma\lg\lg n$ and $\sigma\lg n$ bits, respectively, to store the symbol frequencies during the queries processing. We traverse B 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 $\lg^2 n\lg\sigma$ consecutive bits, 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 $O(\frac{n\sigma\lg\sigma\lg\lg n}{\lg^2 n})$. Once we reach the position i_k of an operation $rank_{b_k}(S,i_k)$, we compute its answer as $A[k]=F_g[b_k]+F_l[b_k]+c$, where $F_g[b_k]$ and $F_l[b_k]$ represent the accumulated frequency of the symbol b_k in F_g and F_l, and c represents the number of occurrences of b_k in the range $B[\lfloor\frac{2i_k}{\lg n}\rfloor\frac{\lg n}{2}...i_k]$, computed by the lookup table described above. This adds $O(m)=O(\frac{n}{\Delta})$ time, which is absorbed by the cost of traversing B. Note that the total cost is $O(\frac{n}{\Delta})$ whenever $\sigma=O(\frac{\lg n}{\lg\lg n})$. The extra space is $O(\sigma\lg n)$ bits.

For a batch of *inserts*, $Q=\{insert_{b_1}(B,i_1),\ldots,insert_{b_m}(B,i_m)\}$, the increasing indexes $i_1,i_2,...$ correspond to their final positions after all the insertions have been done.

```
Input: S, n=|S| and alphabet size σ
Output: Burrows-Wheeler Transform B
 1: Δ←lg_σ n; n_ms←⌈n/Δ⌉
 2: W[1..n_ms] is an array of pairs
 3: Acc[1..σ] is an array of integers
 4: B[1..n] is the output BWT
 5: STEP12(S_1∘S_2, W, Acc, B)
 6: for j←3 to Δ do
 7:     BATCHOPS(B, S_1∘S_j, W, Acc)
 8:     COUNTS_1(S_1∘S_j, W)
 9:     SORT(W)
10:     BATCHINSERT(B, S_1∘S_j, W)
11:     UPDATEACC(Acc)
12:     SORT2(W)
```

Algorithm 1: BWT Computation

```
 1: function STEP12(S_1∘S_j, W, Acc, B)
 2:     T_c← CONTALPHABET(S_1∘S_j)
 3:     SA← SA(T_c)
 4:     for i←1 to 2n_ms do
 5:         j←SA[i]−1
 6:         if SA[i]=1 or SA[i]=n_ms+1 then
 7:             j←j+n_ms
 8:         B[i]← RM(T_c[j])
 9:         if SA[i]>n_ms then
10:             x←SA[i]−n_ms
11:             W[x].f←i
12:             W[x].s←x
13:     COUNTSYMBOLS(B, Acc)
```

Function 2: Steps 1 and 2

```
 1: function BATCHOPS(B, S_1∘S_j, W, Acc)
 2:     SORT(W)
 3:     BATCHRANK(B, W)
 4:     for i←1 to n_ms do
 5:         s← LM(S_1∘S_j[n_ms+W[i].s])
 6:         W[i].f←W[i].f+Acc[s]
```

Function 3: Batch of operations

```
 1: function COUNTS_1(S_1∘S_j, W)
 2:     T_c← CONTALPHABET(S_1∘S_j)
 3:     SA← SA(T_c); k←0
 4:     for i←1 to 2n_ms do
 5:         if SA[i]>n_ms then
 6:             x←SA[i]−n_ms
 7:             W[x].f←W[x].f+k
 8:         else k←k+1
```

Function 4: Count suffixes of S_1

We start moving entries of B to the right to make room for the insertions. The range $B[i_m-m+1..n-1]$ is moved to $B[i_m+1..n+m-1]$ and the ranges $B[i_{k-1}-k+2..i_k-k]$ are moved to $B[i_{k-1}+1..i_k-1]$, for $m \geq k \geq 2$. The ranges must be copied right-to-left to avoid overwriting entries. Finally, we set $B[i_k]=b_k$, for $1 \leq k \leq m$. Using the RAM model, we can move all the entries to the right in $O(\frac{n\lg\sigma}{\lg n}+m)$ time and the time to insert the new symbols is $O(m)$, giving a total time of $O(\frac{n\lg\sigma}{\lg n}+\frac{n}{\Delta})$ for the $\frac{n}{\Delta}$ *inserts*.

The algorithm. We use the constant time operations $\text{LM}(M)$ and $\text{RM}(M)$ to obtain the leftmost and rightmost symbol of the meta-symbol M, respectively. Algorithm 1 shows the general idea. It proceeds in $\Delta=\lg_\sigma n$ steps, and in each step it inserts $n_{ms}=\lceil \frac{n}{\Delta} \rceil$ new symbols to the BWT B. The algorithm allocates two arrays, $W[1..n_{ms}]$ and $Acc[1..\sigma]$. Given the ith meta-symbol M_i at step $j \geq 3$, W is an array of pairs, where the first $W[i].f$ stores the position where $\text{RM}(M_i)$ was inserted in B in the step $j-1$, and the second component $W[i].s=i$. During the execution of the algorithm, W will be sorted by $W[i].f$ in order to perform the batches of *rank* and *insert* operations, and by $W[i].s$ 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). The notation $S_1 \circ S_j$ to represent the concatenation of the rotations S_1 and S_j, yet such rotations are not stored explicitly.

The steps 1 and 2 are processed together (Function 2). First, the continuous alphabet of $S_1 \circ S_2$ and its suffix array are computed (lines 2–3). Then, the first $2n_{ms}$ symbols of B are inserted by extracting the rightmost symbol of each meta-symbol (lines 4–8). The position of the symbols of S_2 in B are stored in W (lines 9–12) and the array Acc is filled with the accumulated frequency of the symbols (line 13).

For the remaining $3 \leq j \leq \Delta$ steps, let $S_j[k]$ be the kth meta-symbol of S_j. The position of the new symbol $\textsc{Rm}(S_j[k-1])$ is $Acc[s]+rank_s(B,W[k])+c_k$, where s is $\textsc{Lm}(S_j[k])$, $Acc[s]$ is already stored, and c_k is the number of suffixes of S_1 appearing before the suffixes of $S_j[k]$ in the suffix array of $S_1 \circ S_j$. The values $rank_s(B,W[k])$ and c_k are computed in Functions 3 and 4, respectively. Function 3 sorts the *rank* operations by their indexes, and then it uses the solution described previously to answer the batch of ranks. The answer of each *rank* is stored in the first component of W. Finally, the algorithm adds the corresponding values of Acc (lines 4–6). In Function 4, the continuous alphabet of $S_1 \circ S_j$ and its suffix array are computed (lines 2 and 3 of Function 4). Then, a scanning over the suffix array counting the suffixes of S_1 is performed (lines 4–10). The answers are added to the first component of W. With the final position of the new symbols, they are sorted by such positions (line 10 of Algorithm 1). Then, the sorted symbols are inserted and Acc is updated (lines 11 and 12). Finally, the new symbols and their positions are restored to their original positions by sorting W by its second components (line 13). This last sorting is necessary to maintain the relative order of the meta-symbols across all the steps.

Complexity analysis. We omit the analysis of Function 2, since it is subsumed by the cost of the last $\Delta-3$ steps. For one step, Function 3 is dominated by the batch of *rank* operations, taking $O(\frac{n\sigma \lg\sigma \lg\lg n}{\lg^2 n})$ time and $O(n\lg\sigma)$ bits of extra space. Function 4 takes $O(\frac{n}{\lg n})$ time and $O(\frac{n}{\Delta}\lg\frac{n}{\Delta})$ bits for the continuos alphabet, and $O(\frac{n}{\Delta})$ time and $O(\frac{n}{\Delta}\lg\frac{n}{\Delta})$ bits of extra space for the suffix array construction, using the algorithm of Kärkkäinen *et al.* [6] and a radix sort with radix $\sqrt{\frac{n}{\Delta}}$ in their internal working. All the other sorting algorithms take $O(\frac{n}{\Delta})$ time and $O(n^{\epsilon}\lg\frac{n}{\Delta})$ bits using radix sort. The batch of *inserts* is done in $O(\frac{n\lg\sigma}{\lg n}+\frac{n}{\Delta})$ time, using $O(n\lg\sigma)$ bits. Finally, the update of Acc takes $O(\frac{n}{\Delta})$ time and no extra space. Thus, repeated Δ times, the final complexity is $O(n+\frac{n\Delta\lg\sigma}{\lg n}+\frac{n\Delta\sigma\lg\sigma\lg\lg n}{\lg^2 n})$, the last two terms dominated by $\textsc{batchInsert}$ and $\textsc{batchRank}$. Note that our complexity grows linearly with Δ and σ. Choosing any $\Delta=\Theta(\lg_\sigma n)$, it simplifies to $O(n+\frac{n\sigma\lg\lg n}{\lg n})$, which when $\sigma=O(\frac{\lg n}{\lg\lg n})$ is $O(n)$.

The working space is $O(\frac{n}{\Delta}\lg n)$ bits, which decreases with Δ and is $O(n\lg\sigma)$ for $\Delta=\Theta(\lg_\sigma n)$. The peak of space consumption is $18\frac{n}{\Delta}\lg n+o(n)$ bits. It occurs in line 3 of Function 4, 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 $12\frac{n}{\Delta}\lg\frac{n}{\Delta}+o(n)$ bits of the suffix array computation, and the $o(n)$ bits of Acc are allocated at the same time. The factor 12 of the suffix array computation comes from the fact that the algorithm [8] performs (in the worst case) $\lg_{3/2} n$ recursive calls, allocating two arrays of $2(\frac{2}{3})^i\frac{n}{\Delta}\lg\frac{n}{\Delta}$ bits in the ith recursive call, and performing a radix sort of radix $\sqrt{\frac{n}{\Delta}}$. We reduce this peak in the next section.

Parallelization and Implementation

Despite the good time complexity of our algorithm for $\sigma=O(\frac{\lg n}{\lg\lg n})$, in practice it turned out to be slow. We found, however, that the algorithm is amenable to parallelization, and thus we designed a multithreaded version. Besides, we further engineer the algorithm, giving up on some theoretical guarantees in favor of faster solutions. We focus on reducing space, since the constant in the space bound ($O(\frac{n}{\Delta}\lg n)$ bits) is rather high. Our code will be publicly at `https://github.com/jfuentess/bwt`.

Parameter Δ. Our analysis shows that the space decreases with Δ. In our experiments, we use values of Δ up to $\lfloor\frac{4096}{\lg\sigma}\rfloor$ symbols, at the cost of increasing linearly the time complexity, but we will take some measures to reduce this impact.

Parallel sorting. We used a parallel implementation of radix sort [16] to sort W (lines 10 and 13 of Algorithm 1 and line 2 of Function 3). The continuous alphabet 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 the quicksort, the array is divided into p equal-size segments sharing a global pivot. Then, on each segment a sequential split function is applying. 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). 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 span 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 continuous alphabet, we use the *parallel range* algorithm [16] to compute the suffix array of an array of meta-symbols. On its implementation, the parallel range algorithm uses two temporal 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 $O((\frac{n}{\Delta})^{\varepsilon})$ span, for $0<\varepsilon<1$. Besides being practical, this reduces the peak of space consumption from $18\frac{n}{\Delta}\lg n+o(n)$ to $8\frac{n}{\Delta}\lg n+o(n)$ bits.

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, leading to the final peak memory consumption of $9\frac{n}{\Delta}\lg n+o(n)$.

Function 2. The continuous alphabet (line 2 of Function 2) is computed with parallel quicksort. The suffix array (line 3 of Function 2) is built using the parallel range algorithm. The *for* loop of lines 4–11 can be parallelized straightforwardly, since each new symbol can be processed independently, giving $O(n/\Delta)$ work, $O(\lg\frac{n}{\Delta})$ span and no extra space. Finally, for the frequency counting of line 13, B is divided into p equal-sized segments, and the frequency counting is performed concurrently in all the segments. Then, the result of the segments are aggregated by performing σ parallel prefix sums, one for each symbol. The result of the prefix sums are stored in *Acc*. 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 span is $O(\lg\frac{n}{\Delta})$. This analysis is also valid for Function 4.

Function 3. The batch of *rank* operations is sorted with parallel radix sort. The operations are supported by dividing the input sequence S 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, similarly to the sequential case. During the traversal of a subsequence $S[\frac{jn}{p}+1...\frac{(j+1)n}{p}]$, each operation $rank_s(S,i)$ with $\frac{jn}{p}<i\leq\frac{(j+1)n}{p}$ is partially answered by counting the number of occurrences of the symbol s in the subsequence $S[\frac{jn}{p}+1...i]$. After the traversal, the p F_g arrays contain the frequency counting in 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 $rank_s(S,i)$, we sum the occurrences of the symbol s in the $(\lfloor\frac{(i-1)p}{n}\rfloor+1)$-th F_l 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). This yields the same work of the sequential algorithm and $O(\sigma\Delta)$ span.

Batches of insertions. Allocating a temporary array, insertions can be parallelized by moving independently all the ranges described in the sequential case. However, we decided to avoid increasing the space, performing the insertions sequentially.

Algorithm 1. The *for* loop of lines 6–12 is carried out sequentially. Note, however, that Function 3 and lines 2–3 of Function 4 can be executed concurrently, since they execute independent procedures. The integer sortings of lines 10 and 13 use parallel radix sort. The update of *Acc* (line 12) can be done in parallel as the last step of Function 2 by storing the frecuency counting 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 $O(n\lg\frac{n}{\Delta}+\frac{n\Delta\sigma\lg\sigma\lg\lg n}{\lg^2 n}+\Delta\sigma)$. The first term replaces the original $O(n+\frac{n\Delta}{\lg n})$ complexity terms, which is more convenient for the large Δ values we use. Note that we preserve a linear dependence on Δ in the second term, coming from the batches of *ranks*. The span 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 $O(\Delta\lg\frac{n}{\Delta})$ span. The space 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.

Experiments

Setup. We implemented our algorithm in C++ and compiled with GCC 6.3 and option -O3. We compare against Okanohara and Sadakane [11] (OS), Kärkkäinen [8] (KAR)[1], the SA algorithm *divSufSort* implemented by Mori (DSS), and the parallel SA algorithm of Labeit *et al.* [17] (PDSS). We only consider one parallel algorithm, PDSS, since it is the parallel version of DSS and, as reported in [17], 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 temporal data during the computation, but it will increase the working space. Baseline were compiled with their best optimization. We tried to include the algorithm of Hayashi and Taura [18], but

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

had problems to compile it. Yet, according to [18], 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 2.1GHz, with per-core L1 and L2 caches of 32KB and 1MB respectively, a per-processor L3 cache of 11MB and 252GB of DDR3 RAM memory (126GB 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 $6 \leq z \leq 12$.

Datasets.
Table 1 shows our datasets. The datasets `gen-x` are sequences with $\sigma = x$, $x \in \{3,7,15,31,63,127\}$, generated with the random generator of *Pizza&Chili*, which distributes the symbols uniformly and independently. We also include real datasets `dna`, `prot`, `dblp` (`xml`), and `eng`, which correspond to sequences of the same name from the *Pizza&Chili* corpus. We left only the four symbols `A`, `C`, `G`, `T` from `dna`.[2] We remind that, during the computation of the BWT, a special symbol `$` is inserted, which increases the alphabet size by one.

Table 1: Datasets.

Dataset	n	σ
`gen-x`	1,073,741,824	x
`dna`	403,919,018	4
`prot`	1,073,741,824	27
`xml`	294,724,056	97
`eng`	1,073,741,824	239

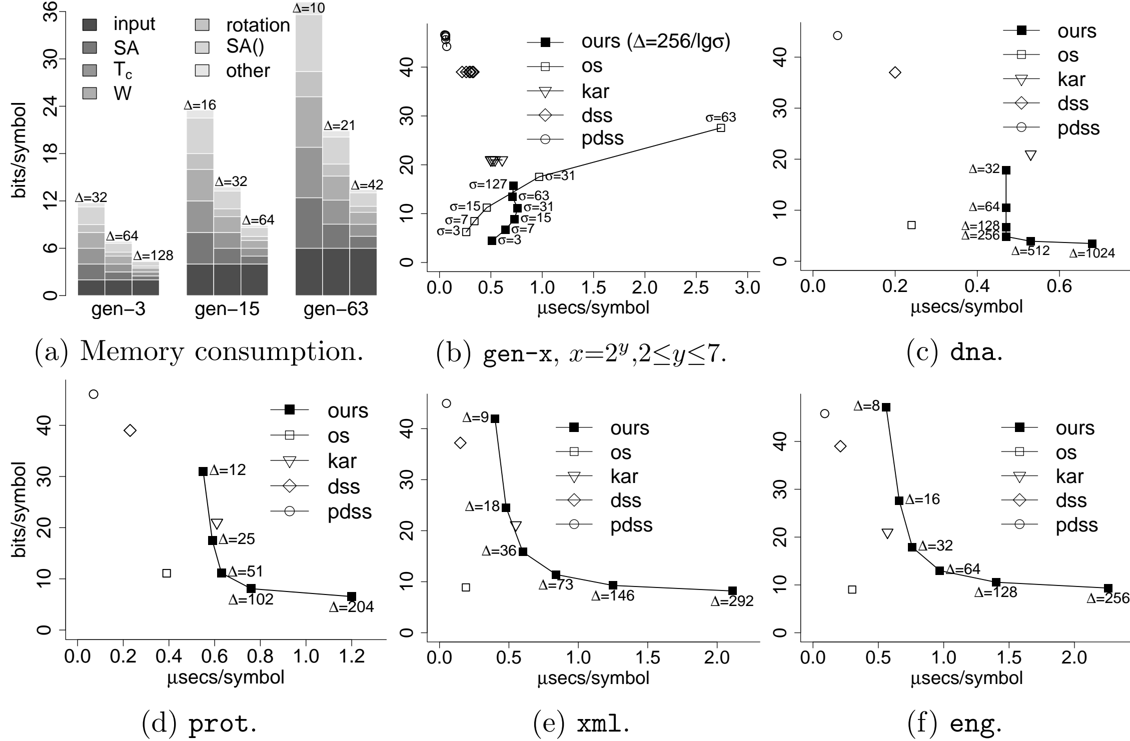
Results. Figure 1a shows the effect of Δ and σ over the memory consumption of our algorithm. In the figure, the peak of memory does not consider the size of the output. The size of the arrays SA, T_c, W and *rotation*, and the memory usage of the suffix array algorithm, SA(), depend on Δ and σ. The segment *other* corresponds to additional arrays needed during the computation. The size of such arrays is negligible.

Figure 1b shows the space/time tradeoff for the `gen-x` 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 with the highest memory consumption. For $\sigma \leq 30$, 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.

Figure 1c shows the results for `dna`, the most common application of BWT construction (e.g., in bioinformatics). In this case, we need to use $\Delta \geq 128$ to outperform OS in space. With $\Delta = 256$ we use 68% of the space of OS while being twice as slow. The price of further reducing space by increasing Δ is an increase in the computation time, lead by the functions BATCHRANK and BATCHINSERT. With $\Delta = 1024$, for example, our algorithm uses 49% of the space of OS, but it is almost 3 times slower.

Considering compressible texts, our algorithm does benefit from compressibility, using less space than on the random text of similar σ, since not all the possible

[2] The use of 15 meta-symbols representing uncertainty by means of subsets of these letters is now less common; instead, explicit probabilities of certainty are separately stored in some cases.



(a) Memory consumption. (b) `gen-x`, $x=2^y, 2\leq y\leq 7$. (c) `dna`.

(d) `prot`. (e) `xml`. (f) `eng`.

Figure 1: Time and memory usage to construct the BWT.

distinct meta-symbols occur. However, the algorithm OS benefits much more, as it uses induced sorting to compute the BWT. Induced sorting classifies the suffixes of a sequence S into A, B and A^*. A suffix $S[i..n]$ is called A-type if $S[i..n]<S[i+1..n]$, B-type if $S[i..n]>S[i+1..n]$, and A^*-type if it is A-type and $S[i-1..n]$ is B-type. By defining meta-symbols as the substrings between two consecutive A^*-type suffixes, induced sorting captures repeated suffixes in the input S and produces a smaller alphabet of meta-symbols [19]. Figures 1d–1f show this effect for the datasets `prot`, `xml` and `eng`, where OS dominates. On those texts, our algorithm needs higher values of Δ to beat OS in terms of space: $\Delta\geq 102$ (`prot`), 292 (`xml`), and 512 (`eng`). The performance of OS on those texts seems indeed related to their high-order entropy H, performing as $\sigma=2^H=10.29$ for `prot`, 1.78 for `xml` and 3.69 for `eng`.

In general, our algorithm offers the best option, or at least the best space within reasonable time, for texts that are close to random (such as DNA sequences), whereas OS is preferable on well-compressible texts. With respect to scalability, our algorithm exhibits a good speedup (we omit a figure for the speedup for lack of space). For 16 threads, our algorithm shows an efficiency (speedup/number of threads) of 50%, decreasing to 30% for 32 threads. The reason is that the used machine has 16 physical cores, which hyperthreading increases to 32 logical cores. The speedup slightly decreases with Δ, which is in line of our analysis, where the span increases with Δ.

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.* [20] and the parallel algorithm Kärkkäinen *et al.*[21]

The slowest functions in our implementation are CONTALPHABET, BATCHRANK,

batchInsert, and sa. In particular, implementations of batchRank and batchInsert that do not scale linearly with Δ would allow us using large Δ values, reducing space, without a sharp impact on construction time. We leave it as future work.

References

[1] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., Digital Systems Research Center, 1994.

[2] P. Ferragina and G. Manzini, "Indexing compressed texts," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[3] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[4] D. K. Kim, J. S. Sim, H. Park, and K. Park, "Constructing suffix arrays in linear time," *J. Discrete Algorithms*, vol. 3, no. 2-4, pp. 126–142, 2005.

[5] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," *J. Discrete Algorithms*, vol. 3, no. 2-4, pp. 143–156, 2005.

[6] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *J. ACM*, vol. 53, no. 6, pp. 918–936, 2006.

[7] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Trans. Comput.*, vol. 60, no. 10, pp. 1471–1484, 2011.

[8] J. Kärkkäinen, "Fast BWT in small space by blockwise suffix sorting," *Theoret. Comput. Sci.*, vol. 387, no. 3, pp. 249–257, 2007.

[9] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, vol. 48, no. 1, pp. 23–36, 2007.

[10] W.-K. Hon, K. Sadakane, and W.-K. Sung, "Breaking a time-and-space barrier in constructing full-text indices," *SIAM J. Comput.*, vol. 38, no. 6, pp. 2162–2178, 2009.

[11] D. Okanohara and K. Sadakane, "A linear-time Burrows-Wheeler transform using induced sorting," in *SPIRE*, 2009, pp. 90–101.

[12] D. Belazzougui, "Linear time construction of compressed text indices in compact space," in *STOC*, 2014, pp. 148–193, Extended version in `http://arxiv.org/abs/1401.0936`.

[13] J. I. Munro, G. Navarro, and Y. Nekrich, "Space-efficient construction of compressed indexes in deterministic linear time," in *SODA*, 2017, pp. 408–424.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Multithreaded algorithms," in *Introduction to Algorithms*, pp. 772–812. The MIT Press, 3rd edition, 2009.

[15] G. Navarro and Y. Nekrich, "Optimal dynamic sequence representations," *SIAM J. Comput.*, vol. 43, no. 5, pp. 1781–1806, 2014.

[16] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *SPAA*, 2012, pp. 68–70.

[17] J. Labeit, J. Shun, and G. E. Blelloch, "Parallel lightweight wavelet tree, suffix array and FM-index construction," *J. Discrete Algorithms*, vol. 43, pp. 2–17, 2017.

[18] S. Hayashi and K. Taura, "Parallel and memory-efficient Burrows-Wheeler Transform," in *IEEE Big Data*, 2013, pp. 43–50.

[19] D. Saad, F. Louza, S. Gog, M. Ayala-Rincón, and G. Navarro, "A grammar compression algorithm based on induced suffix sorting," in *DCC*, 2018, pp. 42–51.

[20] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova, "Engineering external memory induced suffix sorting," in *ALENEX*, 2017, pp. 98–108.

[21] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi, "Parallel external memory suffix sorting," in *CPM*, 2015, pp. 329–342.