

Lightweight LCP Construction for Next-Generation Sequencing Datasets

Markus J. Bauer¹, Anthony J. Cox¹,
Giovanna Rosone² and Marinella Sciortino²

¹ Illumina Cambridge Ltd., United Kingdom
{mbauer,acox}@illumina.com

² University of Palermo, Dipartimento di Matematica e Informatica, Italy
{giovanna,mari}@math.unipa.it

Abstract. The advent of “next-generation” DNA sequencing (NGS) technologies has meant that collections of hundreds of millions of DNA sequences are now commonplace in bioinformatics. Knowing the longest common prefix array (LCP) of such a collection would facilitate the rapid computation of maximal exact matches, shortest unique substrings and shortest absent words. CPU-efficient algorithms for computing the LCP of a string have been described in the literature, but require the presence in RAM of large data structures. This prevents such methods from being feasible for NGS datasets.

In this paper we propose the first lightweight method that simultaneously computes, via sequential scans, the LCP and BWT of very large collections of sequences. Computational results on collections as large as 800 million 100-mers demonstrate that our algorithm scales to the vast sequence collections encountered in human whole genome sequencing experiments.

1 Introduction

The longest common prefix array (LCP) of a string contains the lengths of the longest common prefixes of the suffixes pointed to by adjacent elements of the suffix array (SA) of the string [15]. The most immediate utility of the LCP is to speed up suffix array algorithms and to simulate the more powerful, but more resource consuming, suffix tree. When combined with the suffix array or the Burrows-Wheeler transform (BWT) of a string the LCP facilitates, among other things, the rapid search for maximal exact matches, shortest unique substrings and shortest absent words [14,5,10,1]. Existing algorithms for computing the LCP require data structures of size proportional to the input data to be held in RAM, which has made it impractical to compute the LCP of massive datasets such as the collections of hundreds of millions of reads produced by so-called Next-Generation Sequencing (NGS) technologies.

In this context, the aim of our paper is designing an algorithm for the computation of the LCP of large collections of strings which works on an external

memory system, by performing disk data accesses only via sequential scans, and is lightweight in the sense that its working space requirement is very low.

Computing the LCP of a collection of strings has been considered in the literature [17]. Defining N and K as respectively the sum of the lengths of all strings and the length of the longest string in the collection, the described approach requires $O(N \log K)$ time, but the $O(N \log N)$ bits of internal memory needed to store the collection and its SA in internal memory prevents the method from scaling to massive data.

One can note that several algorithms to compute the LCP of a single string in semi-external memory (see for instance [11]) or directly via BWT (see [5]) could be adapted to solve the problem of computing the LCP of a collection of strings. It could be sufficient to concatenate all the members of the collection into a single string and use distinct end marker symbols as separators. However, assigning a different end marker to each string is not feasible when the collection is very large, but the alternative of terminating each member with the same symbol could lead to LCP values that exceed the lengths of the strings and that depend on the order in which the strings are concatenated. In our approach, we compute the LCP of the collection directly from the strings, without needing to concatenate them and without requiring precomputed auxiliary information such as the SA or BWT of the collection.

In fact, building upon the method of BWT computation introduced in [2], our algorithm adds some lightweight data structures and allows the LCP and BWT of a collection of m strings to be computed simultaneously in $O((m + \sigma^2) \log(N))$ bits of memory, with a worst-case time complexity of $O(K(m + \text{sort}(m)))$, where $\text{sort}(m)$ is the time taken to sort m integers, σ is the size of the alphabet, N is the sum of the lengths of all strings and K is the length of the longest string.

The low memory requirement enables our algorithm to scale to the size of dataset encountered in human whole genome sequencing datasets: in our experiments, we compute the BWT and LCP of collections as large as 800 millions 100-mers.

Section 2 gives preliminaries that we will use throughout the paper, whereas Section 3 describes the sequential computation of the LCP. We present details on the efficient implementation of the algorithm and computational results on real data in Sections 4 and 5, respectively.

2 Preliminaries

Let $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$ be a finite ordered alphabet with $c_1 < c_2 < \dots < c_\sigma$, where $<$ denotes the standard lexicographic order. We append to a finite string $w \in \Sigma^*$ an end marker symbol $\$$ that satisfies $\$ < c_1$. We denote its characters by $w[0], w[1], \dots, w[k]$, where $k + 1$ is the *length* of w , denoted by $|w|$. Note that, for $i < k$, $w[i] \in \Sigma$ and $w[k] = \$$. A *substring* of a string w is written as $w[i, j] = w[i] \dots w[j]$, with a substring $w[0, j]$ being called a *prefix*, while a substring $w[i, k]$ is referred to as a *suffix*.

The *suffix array* of a string w is an array SA containing the permutation of the integers $0 \dots |w| - 1$ that arranges the starting positions of the suffixes of w into lexicographical order. There exist some natural extensions of the suffix array to a collection of sequences (see [17]). We denote by \mathcal{S} the collection of m strings $\{w_0, w_1, \dots, w_{m-1}\}$. We append to each sequence w_i an end marker symbol $\$$ _{i} smaller than c_1 , and $\$$ _{i} < $\$$ _{j} if $i < j$. Let us denote by N the sum of the lengths of all strings in \mathcal{S} .

Let us denote by $\mathcal{S}_{(\text{Pos}, \text{Seq})}$ the suffix starting at the position Pos of the string w_{Seq} . We define the *generalized suffix array GSA* of the collection \mathcal{S} as the array of N pairs (Pos, Seq) , sorted by the lexicographic order of their corresponding suffixes $\mathcal{S}_{(\text{Pos}, \text{Seq})}$. In particular, $GSA[i] = (t, j)$ is the pair corresponding to the i -th smallest suffix of the strings in \mathcal{S} .

The *longest common prefix array* (denoted by LCP) of a collection \mathcal{S} of strings is an array storing the length of the longest common prefixes between two consecutive suffixes of \mathcal{S} in the lexicographic order. For every $j = 1, \dots, N - 1$, if $GSA[j - 1] = (p_1, p_2)$ and $GSA[j] = (q_1, q_2)$, $LCP[j]$ is the length of the longest common prefix of suffixes starting at positions p_1 and q_1 of the words w_{p_2} and w_{q_2} , respectively. We set $LCP[0] = 0$.

Note that the generalization of the suffix array to a collection \mathcal{S} of strings is related to an extension of the notion of the *Burrows-Wheeler transform* to a collection of strings that is a reversible transformation introduced in [12] (see also [13]). Actually, in its original definition, such a transformation produces a string that is a permutation of the characters of all strings in \mathcal{S} but it does not make use of any end marker.

In this paper we suppose that a different end marker is appended to each string of \mathcal{S} . Let us denote by $BWT(\mathcal{S})$ the Burrows-Wheeler transform of the collection \mathcal{S} and its output is produced according to the generalized suffix array of \mathcal{S} . In particular, if $GSA[i] = (t, j)$ then $BWT[i] = w_j[(t - 1) \bmod |w_j|]$.

Note that the output of $BWT(\mathcal{S})$ differs, for at least m symbols, from BWT applied to the string obtained by concatenating all strings in \mathcal{S} . External memory methods for computing $BWT(\mathcal{S})$ are given in [2].

3 LCP computation of a collection of strings via BWT

The main goal of this section is to describe the strategy to compute the LCP of a massive collection of strings via sequential scans of the disk data. In particular, the main theorem of the section enables the simultaneous computation of both LCP and BWT of a collection \mathcal{S} of m strings $\{w_0, w_1, \dots, w_{m-1}\}$. We suppose that the last symbol of each sequence w_i is the end marker $\$$ _{i} . Our method scans all the strings from right to left and both LCP and BWT are incrementally built by simulating, step by step, the insertion of all suffixes having the same length in the generalized suffix array.

We refer to the suffix starting at the position $|w_i| - j - 1$ of a string w_i as its j -*suffix*. With the end marker $\$$ _{i} included, the j -suffix is of length $j + 1$; the

0-suffix contains $\$_i$ alone. Let us denote by \mathcal{S}_j the collection of the j -suffixes of all the strings of \mathcal{S} .

Let us denote by K the maximal length of the strings in \mathcal{S} and by $\text{lcp}_j(\mathcal{S})$ the longest common prefix array of the collection \mathcal{S}_j . It is easy to see that when $j = K$, $\text{lcp}_j(\mathcal{S})$ coincides with the LCP of \mathcal{S} . Since all m end-markers are distinct, the longest common prefix of any pair of the 0-suffixes is 0, so the first m positions into $\text{lcp}_j(\mathcal{S})$ are 0 for any $j \geq 0$.

Note that $\text{lcp}_j(\mathcal{S})$ can be considered to be the concatenation of $\sigma + 1$ arrays $L_j(0), L_j(1), \dots, L_j(\sigma)$ where, for $h = 1, \dots, \sigma$, the array $L_j(h)$ is the LCP of the suffixes of \mathcal{S}_j that start with $c_h \in \Sigma$, while $L_j(0)$ (corresponding to the 0-suffixes) is an array of m zeroes. It is easy to see that $\text{lcp}_0(\mathcal{S}) = L_0(0)$ and that $L_0(h)$ is empty for $h > 0$. For sake of simplicity, each segment is indexed starting from 1. We note that, for each $0 < h \leq \sigma$, $L_j(h)[1] = 0$ and $L_j(h)[i] \geq 1$ for $i > 1$.

Similarly, we define the string $\text{bwt}_j(\mathcal{S})$ as the Burrows-Wheeler transform of the collection of the j -suffixes of \mathcal{S} . This can be partitioned in an analogous way into segments $B_j(0), B_j(1), \dots, B_j(\sigma)$, where the symbols in $B_j(0)$ are the characters preceding the lexicographically sorted 0-suffixes of \mathcal{S}_j and the symbols in $B_j(h)$, with $h \geq 1$, are the characters preceding the lexicographically sorted suffixes of \mathcal{S}_j starting with $c_h \in \Sigma$. Moreover, $\text{bwt}_0(\mathcal{S}) = B_0(0)$ and the segments $B_0(h)$ are empty for $h > 0$.

In this section we show that, for each $j > 0$, $\text{lcp}_j(\mathcal{S})$ can be sequentially constructed by using $\text{bwt}_{j-1}(\mathcal{S})$ and $\text{lcp}_{j-1}(\mathcal{S})$ (in previous work [2], three of the present authors showed how $\text{bwt}_j(\mathcal{S})$ may be computed from $\text{bwt}_{j-1}(\mathcal{S})$). Note that $\text{bwt}_0(\mathcal{S})$ and $\text{lcp}_0(\mathcal{S})$ are defined above.

Given the segments $B_j(h)$ and $L_j(h)$, $h = 0, \dots, \sigma$, for the symbol x occurring at position r of $B_j(h)$ we define the (j, h) -LCP Current Interval of x in r (denoted by $LCI_j^h(x, r)$) as the range $(d_1, r]$ in $L_j(h)$, where d_1 is the greatest position smaller than r of the symbol x in $B_j(h)$, if such a position exists. If such a position does not exist, we define $LCI_j^h(x, r) = L_j(h)[r]$. Analogously, we define for the symbol x the (j, h) -LCP Successive Interval of x in r (denoted by $LSI_j^h(x, r)$) as the range $(r, d_2]$ in $L_j(h)$, where d_2 is the smallest position greater than r of the symbol x in $B_j(h)$, if it exists. If such a position does not exist we define $LSI_j^h(x, r) = L_j(h)[r]$. In our notation, a range is delimited by a square bracket if the correspondent endpoint is included, whereas the parenthesis means that the endpoint of the range is excluded.

Actually, it is easy to verify that $d_1 = \text{select}(\text{rank}(x, r) - 1, x)$ and $d_2 = \text{select}(\text{rank}(x, r) + 1, x)$, where $\text{rank}(x, r)$ counts the number of x 's until position r and $\text{select}(p, x)$ finds the position of the p -th occurrence of x in a segment B_j .

The following theorem shows how to compute the segments $L_j(h)$, with $j > 0$, by using $L_{j-1}(h)$ and $B_{j-1}(h)$ for any $h > 0$. We denote by $\text{Suf}_j(0)$ the lexicographically sorted 0-suffixes and by $\text{Suf}_j(h)$, for $h > 0$, the lexicographically sorted t -suffixes of \mathcal{S}_j , with $t \leq j$ starting with $c_h \in \Sigma$.

Theorem 1. *Let $\mathcal{I} = \{r_0 < r_1 < \dots < r_{q-1}\}$ be the set of the positions in $\text{Suf}_j(z)$ of the j -suffixes starting with the letter c_z . For each position $r_p \in \mathcal{I}$*

$(0 \leq p < q)$,

$$L_j(z)[r_p] = \begin{cases} 0 & \text{if } r_p = 1 \\ 1 & \text{if } r_p > 1 \text{ and } LCI_{j-1}^v(c_z, t) = L_{j-1}(v)[t] \\ \min LCI_{j-1}^v(c_z, t) + 1 & \text{otherwise} \end{cases}$$

where c_v is the first character of the $(j-1)$ -suffix of w_i , and t is the position in $B_{j-1}(v)$ of symbol c_z preceding the $(j-1)$ -suffix of w_i .

For each position $(r_p + 1) \notin \mathcal{I}$ (where $r_p \in \mathcal{I}$ and $0 \leq p < q$), then

$$L_j(z)[r_p + 1] = \begin{cases} 1 & \text{if } LSI_{j-1}^v(c_z, t) = L_{j-1}(v)[t] \\ \min LSI_{j-1}^v(c_z, t) + 1 & \text{otherwise} \end{cases}$$

For each position s , where $1 \leq s < r_p$ (for $p = 0$), $r_{p-1} < s < r_p$ (for $0 < p < q-1$), $s > r_p$ (for $p = q-1$) then

$$L_j(z)[s] = L_j(z)[s - p]$$

For lack of space, the proof of the theorem is omitted and we defer it in the full paper.

A consequence of the theorem is that the segments B_j and L_j can be constructed sequentially and stored in external files. This fact will be used in the next section.

4 Lightweight implementation via sequential scans

Based on the strategy described in the previous section, here we propose an algorithm (named **extLCP**) that simultaneously computes the BWT and LCP of a set of strings \mathcal{S} . Memory use is minimized by reading data sequentially from files held in external memory: only a small proportion of the symbols of \mathcal{S} need to be held in RAM and we do not need to keep the generalized suffix array of \mathcal{S} . Obviously, the generalized suffix array can be a side output of our implementation.

Our method extends previous work [2,3] on computing the BWT of a collection of strings and we follow the notation therein.

Although our algorithm is not restricted to collections of strings of uniform length, for sake of simplicity our description supposes that \mathcal{S} comprises m strings of length k and we assume that $j = 0, 1, \dots, k$ and $i = 0, 1, \dots, m-1$. We simulate m distinct end-markers by using a single end-marker $\$ = c_0$ and setting $w_s[k] < w_t[k]$ if and only if $s < t$, so that if two strings w_s and w_t share the j -suffix, then $w_s[k-j, k] < w_t[k-j, k]$ if and only if $s < t$. Moreover, we assume that the values of $\text{lcp}_j(\mathcal{S})$ do not exceed j and the first m positions into $\text{lcp}_j(\mathcal{S})$ are 0 for any $j \geq 0$.

The main part of the algorithm consists of k consecutive iterations. At iteration j , we consider all the j -suffixes of \mathcal{S} and simulate their insertion in the GSA.

In other words, for each i , we have to find the position of the suffix $w_i[k-j, k]$ according to the lexicographic order of all the suffixes of \mathcal{S} of length at most j , then insert the new symbol circularly preceding the j -suffix of w_i into $B_j(z)$, where $c_z = w_i[k-j]$, for some $z = 1, \dots, \sigma$, and update the values in $L_j(z)$. Consequently, both $\text{bwt}_j(\mathcal{S})$ and $\text{lcp}_j(\mathcal{S})$ are updated accordingly. Note that, at each iteration j , both the segments B_j and L_j , initially empty, are stored in different external files that replace the files used in the previous iteration.

In order to compute $\text{bwt}_j(\mathcal{S})$ and $\text{lcp}_j(\mathcal{S})$, the algorithm needs to hold six arrays of m integers in internal memory. Four of these (P_j , Q_j , N_j and U_j) are useful to compute the BWT (see [2]), a further two (C_j and S_j) are needed to compute and update the values of the longest common prefixes. We will give a description of all these arrays but, for brevity, we will focus on the computation of C_j and S_j .

Each of the arrays P_j , Q_j , N_j and U_j contains m elements, as detailed in the following. At the end of iteration j , if $w_i[k-j, k]$ is the q -th j -suffix then:

- $N_j[q]$ contains the index i . It uses $O(m \log m)$ bits of workspace.
- $Q_j[q]$ stores the index z where $c_z = w_i[k-j]$, i.e. the first symbol of the j -suffix. It uses $O(m \log \sigma)$ bits of workspace.
- $P_j[q]$ contains the position in $B_j(z)$ of the symbol circularly preceding the j -suffix $w_i[k-j, k]$, such a symbol is $w_i[k-j-1]$ and it is stored at the position $N_j[q]$ of U_j . So, it needs $O(m \log(mk))$ bits of workspace.
- U_j stores the new characters to be inserted, one for each sequence in \mathcal{S} , so it uses $O(m \log \sigma)$ bits of workspace.

The arrays C_j and S_j each contain m integers useful to compute $\text{lcp}_j(\mathcal{S})$ for $j > 0$. In particular, $C_j[q]$ stores the value in LCP between the j -suffix $w_i[k-j, k]$ and the previous suffix in the GSA with respect to the lexicographic order of all the suffixes of \mathcal{S} of length at most j , whereas $S_j[q]$ contains the value in LCP between the j -suffix and the next suffix in GSA (if it exists). Such values will be computed at the iteration $j-1$ according to Theorem 1. We observe that C_j and S_j contain exactly one integer for each sequence in the collection and they use $O(m \log k)$ bits of workspace.

At the iteration $j = 0$, the algorithm initializes the segments B_0 and L_0 as described in previous section, i.e. $B_0(0) = w_0[k-1]w_1[k-1] \dots w_{m-1}[k-1]$ and for each $q = 0, \dots, m-1$, we set $L_0(0)[q] = 0$. Consequently, for $q = 0, \dots, m-1$, the arrays are initialized by setting $N_0[q] = q$, $P_0[q] = q+1$, $Q_0[q] = 0$, $C_1[q] = 1$ and $S_1[q] = 1$.

For I/O efficiency, each iteration $j > 0$ can be divided into two consecutive phases: during the first one we read only the segments B_{j-1} in order to find the arrays P_j , Q_j , N_j and U_j . In phase 2, the segments B_{j-1} and L_{j-1} are read once sequentially both for the construction of new segments B_j and L_j and for the computation of the arrays C_{j+1} and S_{j+1} , as they will be used in the next iteration. In the following we describe both the phases of the generic iteration $j > 0$. Figure 1 illustrates the execution of the algorithm for a simple collection at the iterations 12 and 13.

In the first phase the arrays P_j , Q_j and N_j are computed. In particular, if $w_i[k - j - 1]$ (or the end marker \$ for the last step) is the new symbol to be inserted, its position r is obtained by computing the number of occurrences of $c_z = w_i[k - j]$ in $B_{j-1}(0), \dots, B_{j-1}(v - 1)$ and in $B_{j-1}(v)[1, t]$, where $c_v = w_i[k - (j - 1)]$ and t is the position of c_z in $B_{j-1}(v)$. Hence, the index z is stored into Q_j at some position q , the computed position r (where storing the new symbol) is added to the array $P_j[q]$ and i is added to $N_j[q]$. Note that in order to find the positions, a table of $O(\sigma^2 \log(mk))$ bits of memory is used. Finally we sort Q_j , P_j , N_j , C_j , S_j where the first and the second keys of the sorting are the values in Q_j and P_j respectively.

Here we focus on the second phase in which the computation of the segments L_j is performed by using the arrays C_j and S_j constructed during the previous step. Note that the sorting of the arrays allows us to open and sequentially read the pair files $(B_{j-1}(h)$ and $L_{j-1}(h)$ for $h = 0, \dots, \sigma$) at most once.

For all symbols in U_j that we have to insert in the segment $B_j(h)$, the crucial point is to compute C_{j+1} by using LCI_j^h and S_{j+1} by using LSI_j^h while the new files are being constructed, instead of using auxiliary data structures to compute **rank** and **select**. For each index z , we consider all the elements in Q_j equal to z . Because of the sorting, such elements are consecutive. Let $0 \leq l, l' \leq m - 1$ be their first and the last positions, respectively. Hence for each $l \leq p \leq l'$, we have $Q_j[p] = z$ and $P_j[l] < \dots < P_j[l']$. In order to apply Theorem 1, we need to compute LCI_j^h and LSI_j^h of each new symbol in its new position. Since each $B_j(h)$ and $L_j(h)$ are constructed sequentially, we do not know *a priori* the opening positions LCI_j^h and the closing positions LSI_j^h that are used to compute C_{j+1} and S_{j+1} . However we can observe that when we write a symbol x into $B_j(h)$, its occurrence could be the opening or closing positions of some LCI_j^h and LSI_j^h of x , if x is a new symbol. Such considerations are outlined in detail in the following.

For each symbol α that we insert at position s in $B_j(z)$, with $1 \leq s < P_j[l]$, it is easy to see that $B_j(z)[s] = B_{j-1}(z)[s]$ and $L_j(z)[s] = L_{j-1}(z)[s]$. Moreover, the position of α could be the opening position of $LCI_j^z(\alpha, y)$, if α is the new symbol that will be inserted at some next position y .

For each new symbol β that we insert at position $P_j[q]$ in $B_j(z)$ ($l \leq q \leq l'$), we have $\beta = U_j[N_j[q]]$ and, by Theorem 1, it follows that $L_j(z)[P_j[q]] = 0$ if $P_j[q] = 1$ or $L_j(z)[P_j[q]] = C_j[q]$ otherwise. Moreover:

- The position $P_j[q]$ surely is the closing position of $LCI_j^z(\beta, P_j[q])$. If the position $P_j[q]$ is the first occurrence of β in $B_j(z)$, then $LCI_j^z(\beta, P_j[q]) = L_j(z)[P_j[q]]$ and we set $C_{j+1}[q] = L_j(z)[P_j[q]] + 1$ according to Theorem 1. Otherwise, we set $C_{j+1}[q] = \min(LCI_j^z(\beta, P_j[q])) + 1$, whose computation has been started when the interval was opened.
- The position $P_j[q]$ could be the opening position of $LCI_j^z(\beta, y)$, if β will be inserted, as new symbol, at some next position y .
- The position $P_j[q]$ could be the closing position of $LSI_j^z(\beta, y)$, where y represents, eventually, the largest position $P_j[f]$, with $P_j[f] < P_j[q]$, for

$LCI_{12}(G, 3)$ \rightarrow	$L_{12}(0)$	$B_{12}(0)$	Sorted Suffixes	$L_{13}(0)$	$B_{13}(0)$	Sorted Suffixes	
	0	C	$\$0$		0	C	$\$0$
	0	C	$\$1$		0	C	$\$1$
	$L_{12}(1)$	$B_{12}(1)$	Sorted Suffixes		$L_{13}(1)$	$B_{13}(1)$	Sorted Suffixes
	0	G	AAAGCTC $\$1$		0	G	AAAGCTC $\$1$
	2	C	AAC $\$0$		2	C	AAC $\$0$
	3	G	AACAGAAAGCTC$\\$1$		3	G	AACAGAAAGCTC $\$1$
	2	A	AAGCTC $\$1$		2	A	AAGCTC $\$1$
	1	A	AC $\$0$		1	A	AC $\$0$
	2	A	ACAGAAAGCTC $\$1$		2	$\$0$	ACACTGTACCAAC$\\$0$
$LCI_{12}(A, 4)$ \rightarrow	$L_{12}(2)$	$B_{12}(2)$	Sorted Suffixes	$L_{13}(2)$	$B_{13}(2)$	Sorted Suffixes	
	0	A	C $\$0$		0	A	C $\$0$
	1	T	C $\$1$		1	T	C $\$1$
	1	C	CAAC $\$0$		1	C	CAAC $\$0$
	2	A	CACTGTACCAAC$\\$0$		2	G	CACTGTACCAAC $\$0$
	2	A	CAGAAAGCTC $\$1$		2	A	CAGAAAGCTC $\$1$
	1	A	CCAAC $\$0$		1	A	CCAAC $\$0$
	1	G	CTC $\$1$		1	G	CTC $\$1$
	2	A	CTGTACCAAC $\$0$		2	A	CTGTACCAAC $\$0$
	$L_{12}(3)$	$B_{12}(3)$	Sorted Suffixes		$L_{13}(3)$	$B_{13}(3)$	Sorted Suffixes
$LSI_{12}(A, 4)$ \rightarrow	0	A	GAAAGCTC $\$1$	$L_{13}(3)$	0	A	GAAAGCTC $\$1$
	1	A	GCTC $\$1$		3	$\$1$	GAACAGAAAGCTC$\\$1$
	1	T	GTACCAAC $\$0$		<u>1</u>	A	GCTC $\$1$
					1	T	GTACCAAC $\$0$
	$L_{12}(4)$	$B_{12}(4)$	Sorted Suffixes		$L_{13}(4)$	$B_{13}(4)$	Sorted Suffixes
	0	G	TACCAAC $\$0$		0	G	TACCAAC $\$0$
	1	C	TC $\$1$		1	C	TC $\$1$
	1	C	TGTACCAAC $\$0$		1	C	TGTACCAAC $\$0$

Fig. 1. Iteration 12 (on the left) and iteration 13 (on the right) on the collection $\mathcal{S} = \{ACACTGTACCAAC, GAACAGAAAGCTC\}$. We append different end-marker to each string ($\$0$ and $\$1$, respectively) to make the explanation more immediate but the same situation would occur using the same symbol. The first two columns represent the partial LCP and the partial BWT after the iterations. The positions of the new symbols corresponding to the 13-suffixes (shown in bold on the right) are computed from the positions of the 12-suffixes (in bold on the left), which were retained in the array P after the iteration 12. The new values in LCP (shown in bold on the right) are computed during the iteration 12 and are contained in C_{12} . The updated values in LCP (shown in bold and underlined on the right) are computed during the iteration 12 and are contained in S_{12} .

- $l \leq f < q$, where β has been inserted. In this case, we set $S_{j+1}[f] = \min(LSI_j^z(\beta, P_j[q])) + 1$ in according with Theorem 1.
- The position $P_j[q]$ surely is the opening position of $LSI_j^z(\beta, P_j[q])$. We observe that if the position $P_j[q]$ is the last occurrence of β in $B_j(z)$ (we will discover this at the end of the file), it means that $LSI_j^z(\beta, P_j[q]) = L_j(z)[P_j[q]]$, i.e. $S_{j+1}[q] = 1$.

For each symbol α that we insert at position $(P_j[q] + 1)$, with $P_j[q] + 1 \neq P_j[q + 1]$, $B_j(z)[P_j[q] + 1] = B_{j-1}(z)[P_j[q - p]]$ (where p is the number of the new symbols already inserted) and, by Theorem 1, $L_j(z)[P_j[q] + 1] = S_j[q]$. For each

symbol α that we insert at position s in $B_j(z)$, with $P_j[q] < s < P_j[q+1]$ ($l < q \leq l'$), we have $B_j(z)[s] = B_j(z)[s-p]$ and, by Theorem 1, $L_j(z)[s] = L_{j-1}(z)[s-p]$, where p is the number of the new symbols already inserted. Moreover:

- The position s could be the opening position of $LCI_j^z(\alpha, y)$, if α will be inserted, as new symbol, at some next position y .
- The position s could be the closing position of $LSI_j^z(\alpha, P_j[f])$, if α has been inserted, as new symbol, at some previous position $P_j[f]$, with $P_j[f] < P_j[q]$, for $l \leq f < q$. In this case, we set $S_{j+1}[f] = \min(LSI_j^z(\alpha, P_j[f])) + 1$ according to Theorem 1.

For each symbol α that we insert at the position s in $B_j(z)$, where $s > P_j[l']$, we have $B_j(z)[s] = B_j(z)[s - (l' - l + 1)]$ and, by Theorem 1, $L_j(z)[s] = L_{j-1}(z)[s - (l' - l + 1)]$. Moreover, the position of α could be the closing position of $LSI_j^z(\alpha, P_j[f])$, if α has been inserted as a new symbol at some position $P_j[f]$, for $l \leq f \leq l'$. In this case, we set $S_{j+1}[f] = \min(LSI_j^z(\alpha, P_j[f])) + 1$ in according with Theorem 1.

When $B_j(z)$ is entirely built, the closing position of some $LSI_j^z(\alpha, y)$ could remain not found. This means that the last occurrence of α appears at position y . Note that y must be equal to some $P_j[f]$, $l \leq f \leq l'$. In this case, we set $S_{j+1}[f] = 1$ according to Theorem 1.

It is easy to verify that we can run these steps in a sequential way. Moreover, one can deduce that, while the same segment is considered, for each symbol $\alpha \in \Sigma$ at most one $LCI_j^h(\alpha, t)$ for some t , and at most one $LSI_j^h(\alpha, r)$ for some $r \leq t$, will have not their closing position. For this reason we use two arrays *minLCI* and *minLSI* of σ integers that store, for each symbol α in Σ , the minimum among the values of LCP in the possible corresponding *LCI* or *LSI* without closing position, respectively.

From the size of the data structures and from the above description of the phases of the **extLCP** algorithm, we can state the following theorem.

Theorem 2. *Given a collection \mathcal{S} of m strings of length k over an alphabet of size σ , the **extLCP** algorithm computes BWT and LCP of \mathcal{S} by using $O(mk^2 \log \sigma)$ disk I/O and $O((m + \sigma^2) \log(mk))$ bits of memory in $O(k(m + \text{sort}(m)))$ CPU time, where $\text{sort}(m)$ is the time taken to sort m integers.*

The following corollary describes the performance of the method when the collection contains strings of different length.

Corollary 1. *Given a collection \mathcal{S} of m strings over an alphabet of size σ , the LCP and BWT of \mathcal{S} are computed simultaneously in $O((m + \sigma^2) \log(N))$ bits of memory, with a worst-case time complexity of $O(K(m + \text{sort}(m)))$, where $\text{sort}(m)$ is the time taken to sort m integers, N is the sum of the lengths of all strings and K is the length of the longest string.*

5 Computational experiments and discussion

To assess the performance of our algorithm on real data, we used a publicly available collection of human genome sequences from the Sequence Read Archive [8] at `ftp://ftp.sra.ebi.ac.uk/vol1/ERA015/ERA015743/srf/` and created subsets containing 43, 85, 100, 200 and 800 million reads, each read being 100 bases in length. We developed `extLCP`, an implementation of the algorithm described in Section 4, which is available upon request from the authors. Our primary goal was to analyze the additional overhead in runtime and memory consumption of simultaneously computing both BWT and LCP via `extLCP` compared with the cost of using `BCR` ([2]) to compute the BWT alone.

Table 1 shows the results for the instances that we created. We do see increase in runtime since `extLCP` writes the values of LCP after that the symbols in BWT are written, so it effectively increases the I/O operations. So, a time optimization could be obtained if we read/write at the same time both the elements in BWT and LCP by using two different disks. All tests except the 800 million read instance were done on the same machine, having 16Gb of memory and two quad-core Intel Xeon E5450 3.0GHz processors. Although the LCP of a collection of 700 million 100-mers was successfully computed on the same machine using 15Gb of RAM, the collection of 800 million reads needed slightly more than 16Gb so was processed on a machine with 64Gb of RAM and four quad-core Intel Xeon E7330 2.4GHz processors. On both machines, only a single core was used for the computation. Moreover, to examine the behaviour of our algorithm on reads longer than 100bp, we created a set of 50 million 200bp long reads based on the 100 million 100bp instance. It turns out that, although the sheer data volume is the same, `extLCP` uses 1.2Gb and takes 10.3 microseconds per input base.

Our algorithm represents the first lightweight method that simultaneously computes, via sequential scans, the LCP and BWT of a vast collection of sequences. Recall that the problem of the LCP computation of a collection of strings has been faced in [17], but such a strategy works in internal memory. Recently, however, some lightweight approaches for the LCP computation of a single string were described in the literature. Some of them use of the suffix array of the string [11,7,15,9], but the space needed to hold this in RAM is prohibitive for NGS datasets. However, in [5], the authors give an algorithm for the construction of the LCP of a string that acts directly on the BWT of the string and does not need its suffix array. A memory-optimized version of this algorithm [4] (called `bwt_based_lcp2`) needs to hold the BWT of the string in internal memory plus a further $1.5n$ bytes, where n is the length of the input string.

Notice that an entirely like-for-like comparison between our implementation and the above existing implementation for BWT and LCP computation of a string would imply the concatenation of the strings of the collection by different end markers. However, for our knowledge, the existing implementations do not support the many millions of distinct end markers our test collections would require.

An alternative is to concatenate each of strings with the same end marker. This leads to values in the LCP that may possibly exhibit the undesirable prop-

instance	size	program	wall clock	efficiency	memory
0043M	4.00	BCR	0.99	0.84	0.57
	4.00	extLCP	3.29	0.98	1.00
0085M	8.00	BCR	1.01	0.83	1.10
	8.00	extLCP	3.81	0.87	2.00
0100M	9.31	BCR	1.05	0.81	1.35
	9.31	extLCP	4.03	0.83	2.30
0200M	18.62	BCR	1.63	0.58	4.00
	18.62	extLCP	4.28	0.79	4.70
0800M	74.51	BCR	3.23	0.43	10.40
	74.51	extLCP	6.68	0.67	18.00

Table 1. The input string collections were generated on an Illumina GAIIx sequencer, all reads are 100 bases long. Size is the input size in gigabytes, wall clock time—the amount of time that elapsed from the start to the completion of the instance—is given as microseconds per input base, and memory denotes the maximal amount of memory (in gigabytes) used during execution. The efficiency column states the CPU efficiency values, i.e. the proportion of time for which the CPU was occupied and not waiting for I/O operations to finish, as taken from the output of the `/usr/bin/time` command.

erties of exceeding the lengths of the strings and depending on the order in which the strings are concatenated, but does allow the BWT of the resulting string to be computed in external memory by using the algorithm `bwte` proposed in [6].

The combined BWT/LCP computation provided by `extLCP` has a faster runtime than `bwte`. In particular, for the *0085M* instance, `bwte` uses 14Gb of memory and needs 3.84 microseconds per input base vs `extLCP` that uses 2Gb of memory and 3.81 microseconds per input base.

We have also used BCR by suitable preprocessing steps, to simulate the computation of the BWT of the concatenated strings. We compared BCR, `extLCP` and `bwt_based_laca2` on the *0200M* instance. Since the memory consumption of `bwt_based_laca2` exceeded 16Gb on this dataset, we ran the tests on a machine of identical CPU to the 16Gb machine, but with 64Gb RAM.

With BCR, the BWT was created in under 5 hours of wallclock time taking only 4Gb of RAM, while `bwt_based_laca2` required 18Gb of RAM to create the LCP in about 1 hour 45 minutes. Our new method `extLCP` needed 4.7Gb of RAM to create both BWT and LCP in just under 18 hours. Attempting to use `bwt_based_laca2` to compute the LCP of the *0800M* instance exceeded the available RAM on the 64Gb RAM machine.

The experimental results show that our algorithm is a competitive tool for the lightweight simultaneous computation of LCP and BWT on the string collections produced by NGS technologies. Actually, the LCP and BWT are two of the three data structures needed to build a compressed suffix tree (CST) [16] of a string. The strategy proposed in this paper could enable the lightweight construction of CSTs of strings collections for comparing, indexing and assembling vast datasets of sequences when memory is the main bottleneck.

Our current prototype can be further optimized in terms of memory by performing the sorting step in external memory. Further saving of the working space could be obtained if we embody our strategy in **BCRext** or **BCRext++** (see [3]). These methods, although slower than BCR, need to store only a constant and (for the DNA alphabet) negligibly small number of integers in RAM regardless of the size of the input data.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004.
2. M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *CPM*, volume 6661 of *LNCS*, pages 219–231. Springer, 2011.
3. M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the bwt of string collections. *Theor. Comput. Sci.*, 2012 (in press).
4. T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*. To appear.
5. T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *SPIRE*, volume 7024 of *LNCS*, pages 197–208. Springer, 2011.
6. P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *LATIN*, volume 6034 of *LNCS*, pages 697–710. Springer, 2010.
7. J. Fischer. Inducing the LCP-Array. In *WADS*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011.
8. National Center for Biotechnology Information. Sequence Read Archive. <http://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?>
9. S. Gog and E. Ohlebusch. Fast and Lightweight LCP-Array Construction Algorithms. In *ALLENEX*, pages 25–34. SIAM, 2011.
10. J. Herold, S. Kurtz, and R. Giegerich. Efficient computation of absent words in genomic sequences. *BMC Bioinformatics*, 9(1):167, 2008.
11. J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *CPM*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.
12. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.
13. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. A new combinatorial approach to sequence comparison. *Theory Comput. Syst.*, 42(3):411–429, 2008.
14. E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010.
15. S. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *ISAAC*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008.
16. K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.
17. F. Shi. Suffix arrays for multiple strings: A method for on-line multiple string searches. In *ASIAN*, volume 1179 of *LNCS*, pages 11–22. Springer, 1996.