JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 26, Number 9, 2019

Pp. 948–961
DOI: 10.1089/cmb.2018.0230

Multithread Multistring Burrows–Wheeler Transform and Longest Common Prefix Array

PAOLA BONIZZONI, GIANLUCA DELLA VEDOVA, YURI PIROLA, MARCO PREVITALI, and RAFFAELLA RIZZI

ABSTRACT

Indexing huge collections of strings, such as those produced by the widespread sequencing technologies, heavily relies on multistring generalizations of the Burrows–Wheeler transform (BWT) and the longest common prefix (LCP) array, since solving efficiently both problems are essential ingredients of several algorithms on a collection of strings, such as those for genome assembly. In this article, we explore a multithread computational strategy for building the BWT and LCP array. Our algorithm applies a divide and conquer approach that leads to parallel computation of multistring BWT and LCP array.

Keywords: Burrows–Wheeler transform, longest common prefix array, multithreading, parallel algorithms.

1. INTRODUCTION

In this article, we address the problem of building the Burrows–Wheeler transform (BWT) and the longest common prefix (LCP) array for a large collection of strings using a divide and conquer approach. Efficient indexing of very large collections of strings is strongly motivated by the widespread use of next-generation sequencing (NGS) technologies that cheaply produce data that fill several terabytes of secondary storage, which has to be analyzed. This kind of data typically consist of millions to hundreds of millions strings, whose length is between 50 and 200 (but each experiment produces strings with essentially the same length).

The BWT (Burrows and Wheeler, 1994) is a reversible transformation of a text that was originally designed for text compression (and it is still at the core of the widely used bzip2 tool). In the past decade, the BWT has gained importance beyond its initial purpose, becoming the basis for self-indexing data structures such as the FM index (Ferragina and Manzini, 2005), which allows to efficiently search a pattern in a text (Ferragina and Manzini, 2005; Rosone and Sciortino, 2013; Li, 2014) [we refer the reader to Gagie et al. (2017) for a recent survey] or in a graph (Beretta et al., 2017; Denti et al., 2018) and several other crucial tasks on sequences in bioinformatics (Li and Durbin, 2009; Bonizzoni et al., 2016).

Multiple generalization of the BWT to a set of sequences (Mantaci et al., 2007), trees (Ferragina et al., 2009), and graphs (Belazzougui et al., 2016) has been proposed in the past years. The generalization of the BWT (and the FM index) to a collection of strings (Mantaci et al., 2007), usually called multistring BWT, is an ideal tool in genome assembly under the overlap-layout-consensus approach (Staden, 1979). This

Dipartimento di Informatica Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Milan, Italy.

approach requires the efficient construction of a string graph (Myers, 2005), which is the result of finding all prefix–suffix matches (or overlaps) between reads. For this purpose, the BWT of a collection of strings allows efficient constructions of a string graph even from a huge amount of available biological data (Simpson and Durbin, 2010). Indeed, light string graph (LSG) (Bonizzoni et al., 2016) is an external memory algorithm for computing the string graph, whereas fast string graph (Bonizzoni et al., 2017b) is a faster in-memory alternative: both algorithms require the construction of the BWT and LCP array of a collection of strings. We note that most of the approaches for building the BWT and the LCP array of a set of strings usually build them independently or in two successive steps. Since the two data structures are closely related, it makes sense from a theoretical point of view to design methods that build them together to further highlight their interconnection.

The construction of the BWT and LCP array of a huge collection of strings is a challenging and computation-intensive task and the investigation of possible algorithmic approaches is of theoretical interest, besides its practical application. For this purpose, we explore new strategies for computing the BWT and the LCP array of a collection of strings that can efficiently exploit the multithreaded architecture of modern PCs. Current algorithms for the construction of the BWT and the LCP array of a set of strings (Bauer et al., 2012, 2013; Cox et al., 2016) exploit the external memory by storing *partial BWT*s and *partial LCP* arrays that will eventually converge to the BWT and LCP array of the input set in k iterations (where k is the length of the input sequences). Such procedures are hard to parallelize since the partial BWTs are computed sequentially.

The algorithm we describe in this article consists of two phases: the first phase has been originally defined in an external memory algorithm by the same authors (Bonizzoni et al., 2017a), whereas the second phase is original. More precisely, although the strategy in Bonizzoni et al. (2017a) follows the approach of Holt and McMillan (2014) for merging a set of BWTs based on the well-known backward extension operation on a BWT, the strategy of the second phase is based on the opposite forward extension operation on a BWT. This forward extension operation is the novel ingredient of our approach and it allows for a simple parallel implementation of our algorithm. We have implemented the multithread strategy of our algorithm in the tool bwt_lcp and experimented it over real biological data and on a simulated scenario, by showing a significant improvement in the time efficiency even with a moderate use of multiple threads. A preliminary version of this article has appeared in the proceedings of CiE18 (Bonizzoni et al., 2018).

2. PRELIMINARIES

Let $\Sigma = \{c_0, \ldots, c_\sigma\}$ be a finite alphabet where $c_0 = \$$ (called *sentinel*), and $c_0 < \cdots < c_\sigma$ where $<$ specifies the lexicographic ordering over alphabet Σ. We consider a collection $S = \{s_1, \ldots, s_m\}$ of m strings, where each string s_j consists of k symbols over the alphabet $\Sigma \backslash \{\$\}$ and is terminated by the symbol [DOLLAR]. For ease of presentation, we assume that all the strings in S have the same length, but we note that the algorithm does not have such limitation. The i-th symbol of string s_j is denoted by $s_j[i]$ and the substring $s_j[i]s_j[i+1] \cdots s_j[t]$ of s_j is denoted by $s_j[i:t]$. The *suffix* and *prefix* of s_j of length l are the substrings $s_j[k-l+1:k+1]$ (denoted by $s_j[k-l+1:]$) and $s_j[1:l]$ (denoted by $s_j[:l]$), respectively. Note that the string $s_l[k-l+1:k+1]$ is composed of $l+1$ symbols, but we say that its length is l because we do not consider [DOLLAR] as part of the input string when considering its length. The l *suffix* and l *prefix* of a string s_j are the suffix and prefix with length l, respectively. If l is greater than the length of s_j, the l prefix and the l suffix of s_j are equal to s_j. The lexicographic ordering among the strings in S is defined in the usual way. Although we use the same sentinel to terminate strings, we sort equal suffixes of different strings by assuming an implicit ordering of the sentinels that is induced by the ordering of the input strings. More precisely, we assume that given $s_i, s_j \in S$, with $i < j$, then the sentinel of s_i is lexicographically smaller than the sentinel of s_j, that is, $s_i[k+1] < s_j[k+1]$.

Given the lexicographic ordering X of the suffixes of S, the *Suffix Array* of S is the $(m(k+1))$ long array SA such that $SA[i]$ is equal to (p, j) if and only if the i-th element of X is the p suffix of string s_j. The *multistring BWT* of S is the $(m(k+1))$ long array B s.t. if $SA[i] = (p, j)$, then $B[i]$ is the first symbol of the $(p+1)$ suffix of s_j if $p < k$, or \$ otherwise. In other words, B is the concatenation of the symbols preceding the ordered suffixes of S. The *LCP array* of S is the $(m(k+1))$ long array LCP s.t. $LCP[i]$ is the length of the longest prefix between suffixes $X[i-1]$ and $X[i]$. Conventionally, $LCP[1] = -1$. Figure 3 illustrates the main concepts described in this paragraph for the example instance shown in Figure 1.

s_1:	G	T	T
s_2:	C	T	G
s_3:	T	G	G

FIG. 1. Example instance.

Given $n+1$ arrays $V_0, V_1, \ldots, V_n$, an array W is an *interleave* of $V_0, V_1, \ldots, V_n$ if W is the result of merging the arrays s.t. (i) there is a 1-to-1 function ψ_W from the set $\cup_{i=0}^{n}\{(i,j) : 1 \leq j \leq |V_i|\}$ to the set $\{q : 1 \leq q \leq |W|\}$, (ii) $V_i[j] = W[\psi_W(i,j)]$ for each i, j, and (iii) $\psi_W(i, j_1) < \psi_W(i, j_2)$ for each $j_1 < j_2$.

By denoting with $\mathcal{L} = \sum_{i=0}^{n} |V_i|$ the total length of the arrays, the interleave W is an $\mathcal{L}$ long array representing the fusion of the arrays $V_0, V_1 \ldots, V_n$ that preserves the relative order of the elements of each array. Hence, for each i with $0 \leq i \leq n$, the j-th element of V_i corresponds to the j-th occurrence in W of an element of V_i. This fact allows to encode the function ψ_W as an $\mathcal{L}$ long array I_W s.t. $I_W[q] = i$ if and only if $W[q]$ is an element of V_i. Given I_W, we reconstruct W by noticing that $W[q]$ is equal to $V_{I_W[q]}[j]$, where j is the number of values equal to $I_W[q]$ in the interval $I_W[1, q]$; we refer to this value as the *rank* of the element $I_W[q]$ at position q. In the following, we refer to vector I_W as *encoding*. Algorithm 1 shows how to reconstruct an interleave from its encoding.

Let l be an integer between 0 and k and let X_l and B_l be m long arrays s.t. $X_l[i]$ is the i-th smallest l suffix of S and $B_l[i]$ is the symbol preceding it.

Note that the BWT B is an interleave of the $k+1$ arrays $B_0, B_1, \ldots, B_k$, since the ordering of symbols in B_l is preserved in B, that is, B is *stable* w.r.t. each array $B_0, B_1, \ldots, B_k$. This fact is a direct consequence of the definition of B and B_l. Then let us denote by I_B the encoding of the interleave B. For the same reason, the lexicographic ordering X of all suffixes of S is an interleave of the arrays $X_0, X_1, \ldots, X_k$ and let I_X denote the encoding of that interleave. Then $I_B = I_X$. Therefore, computing either I_B or I_X is equivalent to computing the *BWT* of the input collection S.

Figure 2 illustrates the arrays $B_0, B_1, \ldots, B_k$ for the example of Figure 1, whereas Figure 3 shows the interleave on the same example.

Algorithm 1: Reconstruct the interleave W from the encoding I_W

```
for i ← 0 to n do
    rank[i] ← 0;
for q ← 1 to |I_W| do
    i ← I_W[q];
    rank[i] ← rank[i] + 1;
    W[q] ← V_i[rank[i]];
```

3. THE ALGORITHM

Our algorithm for building the BWT and the LCP array of a set $S = \{s_1, \ldots, s_m\}$ of strings with length k consists of two distinct steps: in the first step, the arrays $B_0, \ldots, B_k$ are computed, whereas the second step determines $I_X = I_B$ by implicitly reordering the whole set of suffixes in arrays X_l, thus reconstructing the BWT B as an interleave of $B_0, \ldots, B_k$ whose encoding is I_B. At the same time, the algorithm computes the LCP array.

A first preprocessing step performs a column-wise splitting of the input sequences in $k+1$ arrays $S_0, \ldots, S_k$ s.t. for $i \neq k$, $S_i[j]$ is equal to the symbols at position $k-i$ of string s_j, that is, S_i lists the symbols preceding the i suffixes of the sequences $s_1, \ldots, s_k$, whereas for $i = k$, $S_i[j]$ is equal to the last character of s_j, that is, \$. These arrays are used to compute the arrays $B_0, \ldots, B_k$ as follows. Array B_0 is trivially the vector of the last characters $s_1[k], \ldots, s_m[k]$ of the reads, that is, it is equal to the array S_0. For $l > 0$, array B_l is a permutation of S_l and it is computed from B_{l-1} by a bucket sort strategy. The procedure for computing arrays $B_1, \ldots, B_k$ for a set of constant-length strings is detailed in Section 3.1. It is easy to note that this step requires

FIG. 2. Arrays B_0, B_1, B_2, B_3 and the arrays X_0, X_1, X_2, X_3 for the example shown in Figure 1.

B_0	X_0
T	\$
G	\$
G	\$

B_1	X_1
T	G\$
G	G\$
T	T\$

B_2	X_2
T	GG\$
C	TG\$
G	TT\$

B_3	X_3
\$	CTG\$
\$	GTT\$
\$	TGG\$

B	X	$I_B = I_X$	LCP		
T	\$	0	-1	$B_0[1]$	$X_0[1]$
G	\$	0	0	$B_0[2]$	$X_0[2]$
G	\$	0	0	$B_0[3]$	$X_0[3]$
\$	CTG\$	3	0	$B_3[1]$	$X_3[1]$
T	G\$	1	0	$B_1[1]$	$X_1[1]$
G	G\$	1	1	$B_1[2]$	$X_1[2]$
T	GG\$	2	1	$B_2[1]$	$X_2[1]$
\$	GTT\$	3	1	$B_3[2]$	$X_3[2]$
T	T\$	1	0	$B_1[3]$	$X_1[3]$
C	TG\$	2	1	$B_2[2]$	$X_2[2]$
\$	TGG\$	2	2	$B_3[3]$	$X_3[3]$
G	TT\$	2	1	$B_2[3]$	$X_2[3]$

FIG. 3. Encoding $I_B = I_X$ and LCP array for the set of reads of Figure 1. The last two columns report, for each element of B and X (first two columns), the source element in arrays B_l and X_l (respectively). LCP, longest common prefix.

$\mathcal{O}(mk)$ time. Extending the procedure to handle a set of strings with different lengths is rather simple and causes variable-length arrays B_i. In practice, variable-length strings can happen when the strings are the result of NGS sequencing, since those strings usually undergo a trimming step to remove low-quality regions.

The second step of the method computes the interleave $I_X = I_B$ in $\mathcal{O}(kmL)$ time, where L is the maximum value stored in the LCP array, that is length of the longest common string that appears at least twice in the input set. This second step implicitly sorts the whole set of suffixes of S. The idea is to construct I_B through L iterations, where each iteration p, from 1 to L, computes the encoding of the interleave of arrays X_l giving the sorting of the suffixes according to their first p symbols, from the encoding giving the sorting according to their first $p-1$ symbols.

The first iteration starts from the encoding of the interleave given by the concatenation $X_0, \ldots, X_k$ (i.e., suffixes in X_0 are followed by suffixes in X_1, …, are followed by suffixes in X_k). We can maintain a partial LCP array Lcp_p together with the encoding I_{X^p}, where Lcp_p is the LCP array of the p prefixes of the suffixes sorted by I_{X^p}. Since L is the length of the longest common substring in the input set of reads, the encoding I_{X^p} computed by the last ($p = L$) iteration gives the lexicographic ordering X of the suffixes in S and thus Lcp_p is the LCP array of S. We point out that the task of implicitly sorting the suffixes of S can be accomplished by following two different strategies, both exploiting a bucket sort approach and both strategies work even for variable-length reads.

The first strategy to compute the interleave I_B is to adopt the approach proposed in Holt and McMillan (2014) for merging a set of BWTs, and also used in Egidi et al. (2018), to compute also the LCP array together with the BWT. This approach is based on the *backward extension* of the suffixes, where at each iteration p the order of the suffixes sorted by their p prefix is computed by the order of the suffixes sorted by their $(p-1)$ suffix by considering the symbol that must be prepended to the latter suffixes to produce the former. The second strategy to compute I_B is based on the *forward extension* of the $(p-1)$ prefixes of the suffixes in the ordering given by $I_{X^{p-1}}$ to get the encoding I_{X^p} by ordering suffixes by their p prefixes and is the strategy presented in this article. Both methods require $\mathcal{O}(mkL)$ time. We note that storing all the suffixes of X would require too much space and in the following sections we provide an efficient algorithm to induce the order of the suffixes from the arrays B_l computed in the previous step.

This section is laid out as follows: in Section 3.1 we detail the procedure to compute the arrays $B_0, B_1, \ldots, B_k$, in Section 3.2 we provide an algorithm to efficiently compute the interleave I_B, in Section 3.2.1 we show how to extend the previous method to compute the *LCP* array and how we can limit the number of iterations of the algorithm to L, in Section 3.2.2 we describe how we can compute in parallel a fundamental data structure used in Section 3.2, and, finally, in Section 3.3 we analyze the time complexity of the overall method.

3.1. Computing arrays $B_0, B_1, \cdots, B_k$

The input strings $s_1, s_2, \cdots s_m$ are preprocessed to compute $k+1$ arrays $S_0, S_1, \ldots, S_k$ giving a fast access to the input symbols. Recall that, for each $l \neq k$, array S_l lists the symbols at position $k-l$ of the input strings, whereas S_k is a list of m sentinel symbols. Observe that $S_l[i]$ is the symbol preceding the l suffix of s_i. Arrays S_l can be computed in $O(km)$ time by reading sequentially the input strings. Algorithm 2 takes in input the arrays $S_0, S_1, \ldots, S_k$ and uses $k+1$ m long arrays $N_0, N_1, \ldots, N_k$, s.t. $N_l[i] = q$ iff the i-th element in X_l is the l suffix of the string s_q. In other words, array N_l lists the indexes of the input strings induced by the lexicographic ordering of their l suffixes.

The symbol $B_l[i]$, for $0 \leq l < k$, precedes the l suffix of s_q if $N_l[i] = q$, that is $B_l[i] = s_q[k-l]$. When $l = k$, $B_k[i] = \$$. Note that N_0 is the sequence of indexes $1, 2, \ldots, m$ since sentinels, corresponding to the 0 suffixes, are sorted according to the order of the input strings, and B_0 is the sequence $s_1[k], s_2[k], \ldots, s_m[k]$ of the last symbols of the input strings (i.e., the symbols before the sentinels), that is $B_0 = S_0$.

Given a symbol c_h in the alphabet Σ, we define the c_h *projection* over the array N_l as the operation $\Pi_{c_h}(N_l)$ projecting from N_l only the entries q such that $s_q[k-(l+1)] = c_h$. In other words, $\Pi_{c_h}(N_l)$ extracts the entries of N_l corresponding to strings whose l suffix is preceded by symbol c_h. Then, the following proposition holds.

Proposition 1. *The ordering of the sequence of indexes in N_{l-1} of strings in S with respect to the l suffix starting with symbol c_h is equal to $\Pi_{c_h}(N_{l-1})$.*

As a main consequence, the array N_l can be simply obtained from N_{l-1} as the concatenation $\Pi_{c_0}(N_{l-1}) \cdot \Pi_{c_1}(N_{l-1}) \cdots \Pi_{c_\sigma}(N_{l-1})$. Observe that the c_h projection of N_{l-1} can be computed by listing in order the entries q at the positions i of N_l s.t. $B_{l-1}[i] = c_h$ since B_{l-1} lists the symbols preceding the ordered $(l-1)$ suffixes.

Algorithm 2 computes the arrays $B_0, \ldots, B_k$ and arrays $N_0, \ldots, N_k$ in k iterations. At iteration l, arrays B_l and N_l are computed from arrays B_{l-1} and N_{l-1}. The c_h projection of N_{l-1} is stored in a list $\mathcal{P}(c_h)$ that is empty at the beginning of the iteration. First N_l is computed from B_{l-1} and N_{l-1} as follows. B_{l-1} and N_{l-1} are sequentially read and, for each position i, the value $q = N_{l-1}[i]$ is added to the list $\mathcal{P}(c_h)$, where c_h is the symbol $B_{l-1}[i]$ (lines 7–10). Then, the array N_l is obtained as the concatenation $\mathcal{P}(c_0) \cdot \mathcal{P}(c_1) \cdot \ldots \cdot \mathcal{P}(c_\sigma)$ (line 11). After computing N_l, the array B_l can easily be induced. Indeed, assuming that the j-th element in the ordered list of l suffixes is the suffix of string s_q (i.e., $N_l[j] = q$), then the symbol preceding such suffix can be directly accessed at position q of array S_l (recall that $S_l[q]$ is $s_q[k-l]$ when $k \neq l$, or the sentinel \$ otherwise). Thus, the algorithm reads sequentially N_l, and, for each entry q at position i, sets $B_l[i]$ to the value $S_l[q]$ (lines 12–14).

Algorithm 2: Compute arrays $B_0, B_1, \cdots, B_k$

```
   Input: Arrays S_0, ..., S_k.
   Output: Arrays B_0, ..., B_k.
 1 for i ← 1 to m do
 2 |   B_0[i] ← S_0[i];
 3 |   N_0[i] ← i;
 4 for l ← 1 to k do
 5 |   for h ← 0 to σ do
 6 |   |   P(c_h) ← empty list;
 7 |   for i ← 1 to m do
 8 |   |   c_h ← B_{l-1}[i];
 9 |   |   q ← N_{l-1}[i];
10 |   |   Append q to P(c_h);
11 |   N_l ← P(c_0)P(c_1)···P(c_σ);
12 |   for i ← 1 to m do
13 |   |   q ← N_l[i];
14 |   |   B_l[i] ← S_l[q];
```

3.2. *Computing the interleave* $\mathrm{I_B}$

The main part of our algorithm computes the encoding I_X of the interleave X of the arrays $X_0, X_1, \ldots, X_k$, giving the lexicographic ordering of all suffixes of the input set S and at the same time computing the LCP array. Recall that I_X is equal to the encoding I_B of the interleave of the arrays $B_0, B_1, \ldots, B_k$ giving the BWT B.

Before entering into the details, we provide some definitions that allow us to introduce the notion of p segment, which is fundamental to our algorithm.

Definition 2 (p precedes)**.** *Let $\alpha = s_{i_\alpha}[k - l_\alpha + 1:]$ and $\beta = s_{i_\beta}[k - l_\beta + 1:]$ be two generic suffixes of S, with length, respectively, l_α and l_β. Then, given an integer p, $\alpha \prec_p \beta$ (and we say that α p precedes β) iff one of the following conditions hold: (1) $\alpha[:p]$ is lexicographically strictly smaller than $\beta[:p]$, (2) $\alpha[:p] = \beta[:p]$ and $l_\alpha < l_\beta$, (3) $\alpha[:p] = \beta[:p]$, $l_\alpha = l_\beta$ and $i_\alpha < i_\beta$.*

Definition 3 (p interleave). *Given the arrays $X_0, X_1, \ldots, X_k$, the p interleave X^p ($0 \le p \le k$) is the interleave s.t. $X^p[i]$ is the i-th smallest suffix in the $\prec_p$ ordering of all the suffixes of S.*

Definition 4 (p segment)**.** *Let X^p be the p interleave of $X_0, X_1, \ldots, X_k$, and let i be a position. Then, the p segment of i in X^p is the maximal interval $[b, e]$ s.t. $b \le i \le e$ and all suffixes in $X^p[b, e]$ have the same p prefix. Positions b and e are called, respectively, begin and end position of the segment, and the common p prefix is denoted by $w_p(b, e)$.*

It is immediate to observe that the set of all p segments of a p interleave forms a partition of its positions. Observe that, by definition, a suffix shorter than p characters belongs to a p segment $[b, e]$, with $b = e$. In other words, such suffix is the only one in its p segment.

It is immediate to verify that X^k (i.e., the suffixes sorted according to the $\prec_k$ relation) is equal to X, hence $I_X = I_{X^k}$. Our approach determines I_{X^k} by iteratively computing I_{X^p} from $I_{X^{p-1}}$ by increasing values of p, starting from I_{X^0}. Observe that X^0 lists the suffixes in the same order given by the concatenation of arrays $X_0, X_1, \ldots, X_k$ and the encoding I_{X^0} is trivially given by $|X_0|$ 0's, followed by $|X_1|$ 1's, ..., followed by $|X_k|$ values equal to k.

Now, we focus on describing how to compute I_{X^p} from $I_{X^{p-1}}$ (Algorithm 3) at the iteration p. Note that Algorithm 3 computes also a partial LCP array Lcp_p (whose description is postponed until Section 3.2.1), which at the end of the iterations is the LCP array of the input set of reads. We point out that Algorithm 3 is a sequential procedure that can be easily parallelized (we defer its analysis to the end of this section).

Computation of I_{X^p} from $I_{X^{p-1}}$ involves an implicit sorting of the suffixes in X^{p-1} by their p-th symbol. It is quite easy to see that this sorting must be performed inside the $(p-1)$ segments independently of each other. The procedure scans sequentially each $(p-1)$ segment on the encoding $I_{X^{p-1}}$; for a position i, the i-th suffix, w.r.t. $\prec_{p-1}$, has length $j = I_{X^{p-1}}[i]$. For each value of j ($0 \le j \le k$), the array *rank* maintains the number of suffixes having length j that have been encountered so far (i.e., up to position i). When processing a $(p-1)$ segment $[b, e]$, $\sigma + 1$ lists L_{c_i} ($0 \le i \le \sigma$) are maintained, each one storing the lengths of the suffixes (as encountered in the segment) whose p-th character is c_i. Once the procedure has completed the $(p-1)$ segment $[b, e]$, then the $n \le \sigma + 1$ nonempty lists among $L_{c_0}, L_{c_1}, \cdots, L_{c_\sigma}$ (considered in the lexicographic order of the alphabet symbols) give n p segments of I_{X^p} originating from the $(p-1)$ segment $[b, e]$. Observe that the start of the i-th p segment has an offset off_i with respect to the start b equal to the total length of the first $i-1$ nonempty lists. Therefore, its absolute start is $b_i = b + \mathit{off}_i$. Moreover, observe that $b_1 = b$ and the n p segments induce over $[b, e]$ the partition $[b, b_2 - 1], [b_2, b_3 - 1], \cdots, [b_n, e]$ into n intervals.

Algorithm 3: Compute I_{X^p} from $I_{X^{p-1}}$

$L_{c_0}, L_{c_1}, \ldots, L_{c_\sigma} \leftarrow$ empty lists;
rank $\leftarrow$ a vector of k 0's;
for each $(p-1)$ *segment* $[b, e]$ *in increasing order of start* b **do**
  $L_{c_0}, L_{c_1}, \ldots, L_{c_\sigma} \leftarrow$ empty lists;
  for $i \leftarrow b$ *to* e **do**
    $j \leftarrow I_{X^{p-1}}[i]$;
    $rank[j] \leftarrow rank[j] + 1$;
    $c \leftarrow Q_j^p[rank[j]]$;
    Append j *to* L_c;
  $I_{X^p}[b, e] \leftarrow$ concatenation $L_{c_0}, L_{c_1}, \ldots, L_{c_\sigma}$;
  $B \leftarrow$ empty list;
  for each *symbol* c_h, $0 \le h \le \sigma$ **do**
    Append $b + \sum_{c < c_h} |L_c|$ *to* B iff L_{c_h} is nonempty;
  $Lcp_p[b, e] \leftarrow Lcp_{p-1}[b, e]$;
  for each $i \in [b, e]$ **do**
    $Lcp_p[i] = Lcp_p[i] + 1$ iff $i \notin B$;

To access the p-th symbols of the suffixes in the $\prec_{p-1}$ relation, we introduce the arrays $Q_0^p, Q_1^p, \cdots, Q_k^p$. More precisely, Q_l^p ($0 \le l \le k$) is the m long array s.t. $Q_l^p[i]$ is the p-th symbol of the suffix $X_l[i]$ if $p \le l$, or $Q_l^p[i] = \$$ otherwise. Moreover, let Q^p be the interleave of the arrays $Q_0^p, Q_1^p, \ldots, Q_k^p$ s.t. its encoding is $I_{X^{p-1}}$. In other words, $Q^p[i]$ is the p-th symbol of the suffix $X^{p-1}[i]$. As a consequence of this definition, the p-th symbol of the suffix in position i of $I_{X^{p-1}}$ is $Q_j^p[rank[j]]$, where $j = I_{X^{p-1}}[i]$.

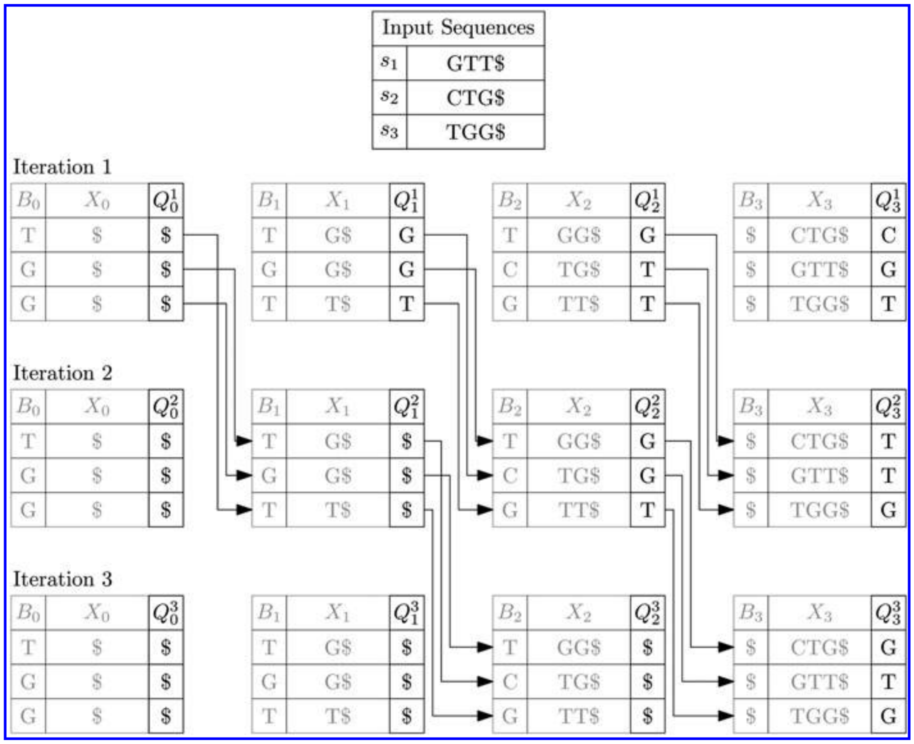

FIG. 4. Schematic showing how arrays Q_l^p are computed at each iteration. For each iteration i, this figure shows arrays B_l along the suffix associated with X_l and the elements in all the arrays Q_l^i. We recall that the arrays X_l are not stored but we show them for ease of presentation and that arrays B_l are static through the execution. For each suffix in X_l, Q_l^p reports the p-th character of the suffix. Arrows map each element of Q_{l-1}^{p-1} to its correct position in Q_l^p.

Computing the Q_l^p arrays is the most complex part of our algorithm, thus we will devote Section 3.2.2 to its description.

The following lemma shows that a p segment $[b_p, e_p]$ of I_{X^p} is computed from a $(p-1)$ segment $[b, e]$ of $I_{X^{p-1}}$ s.t. $b \leq b_p$ and $e \geq e_p$. Notice that Lemma 5 proves the correctness of Algorithm 3.

Lemma 5. *Let $[b, e]$ be a $(p-1)$ segment of X^{p-1}. Then, $X^p[b, e]$ is a permutation of $X^{p-1}[b, e]$ defined by the permutation $\Pi_{b,e}^{p-1}$ of the indexes $(b, b+1, \ldots, e)$ producing the stable ordering of the symbols in $Q^p[b, e]$, s.t. the r-th suffix of $X^p[b, e]$ is the suffix of X^{p-1} in position $\Pi_{b,e}^{p-1}[r]$.*

Proof. First we prove that $X^p[b, e]$ is a permutation of $X^{p-1}[b, e]$. Let us denote with w the $(p-1)$ prefix common to suffixes in $X^{p-1}[b, e]$, and let i be a position in $[b, e]$. Given a position $q < b$, by definition, the $(p-1)$ prefix w_q of $X^{p-1}[q]$ is strictly smaller than w. Then, the p prefix of $X^{p-1}[q]$ is strictly smaller than the p prefix of $X^{p-1}[i]$. In the same way, given a position $q' > e$ by definition, the $(p-1)$ prefix w'_q of $X^{p-1}[q']$ is strictly greater than w. Then, the p prefix of $X^{p-1}[q']$ is strictly greater than the p prefix of $X^{p-1}[i]$. Hence, the set of the suffixes of X^{p-1} before b and the set of the suffixes after e are equal (respectively) to the set of the suffixes of X^p before b and to the set of the suffixes after e, thus deriving that for $b \leq i \leq e$ the suffix $X^{p-1}[i]$ is equal to $X^p[j]$ for some j in $[b, e]$, completing the proof of the first part. Figures 5 and 6 are related. Figure 6 shows how the partition associated with the 1-segments is refined into 2-segments. Furthermore, all suffixes in $X^{p-1}[b, e]$ share the common $(p-1)$ prefix w, and, therefore, their $\prec_p$ order can be determined by ordering the suffixes by their p-th symbols. More precisely, the suffix $X^{p-1}[i]$ $(b \leq i \leq e)$ is the r-th suffix in $X^p[b, e]$, where r is the rank of its p-th character in $Q^p[b, e]$. □

Given the suffix in position i of X^{p-1} such that i is in the $(p-1)$ segment $[b, e]$, Lemma 5 allows to compute its position $i' \in [b, e]$ on X^p. Let $\#^<$ be the number of symbols of $Q^p[b, e]$ that are strictly smaller than $Q^p[i]$ and let $\#_q^=$ be the number of symbols of $Q^p[b, q]$ that are equal to $Q^p[i]$. Then, the rank of suffix $X^{p-1}[i]$ in $X^p[b, e]$ is $r = \#^< + \#_i^=$, thus deriving that its position in X^p is $i' = b + r - 1$. Note that the positions $(b, b+1, \ldots, e)$ on X^p are partitioned into n p segments $[b_1 = b, e_1], \ldots, [b_n, e_n = e]$ (referred as

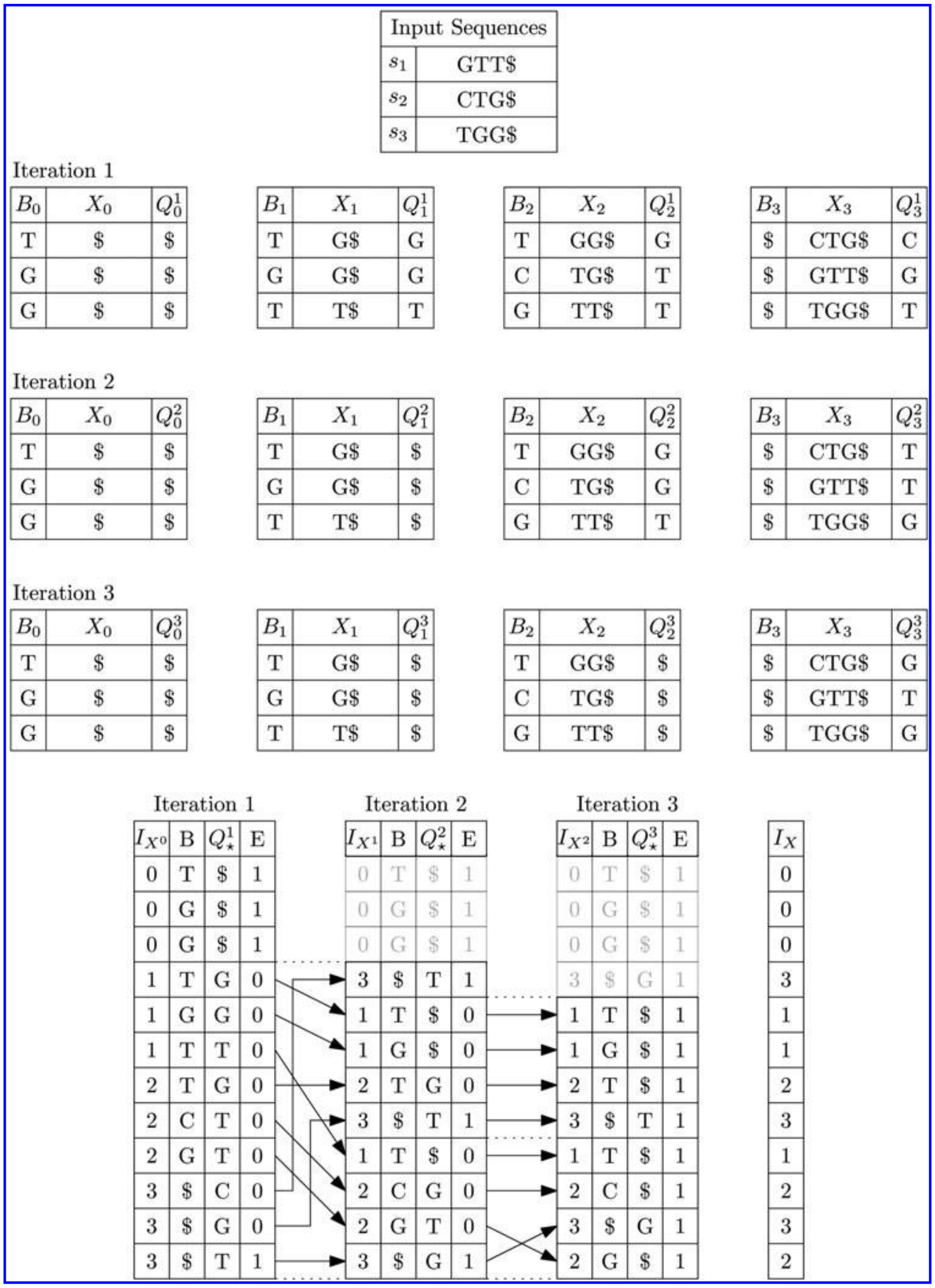

FIG. 5. Schematic showing how the encoding I_X is computed. For each iteration i, this figure shows I_X^{i-1} along with the i interleave on B. p Segments are represented by means of the bitvector E that marks the ending position of each segment. Arrows map each element of the p segment to their position in the $(p+1)$ segments induced by the p segment. The $(p+1)$ segments induced by the p segment are grouped together by dotted lines that highlight this relationship between different iterations. p Segments of width 1 that do not require additional analysis are grayed out. The computation ends when all the p segments have width equal to 1.

induced by the $(p-1)$ segment $[b, e]$ of X^{p-1}), where n is the number of distinct non-\$ symbols in $Q^p[b, e]$ plus the number $\#_\$$ of symbols \$ in $Q^p[b, e]$. Observe that the first $\#_\$$ p segments $[b_1, e_1], \ldots, [b_{\#_\$}, e_{\#_\$}]$ have width 1, whereas the width of the last $n - \#_\$$ p segments $[b_{\#_\$+1}, e_{\#_\$+1}], \ldots, [b_n, e_n]$ can be computed as follows. Let $\{c_1, \ldots, c_{n-\#_\$}\}$ be the ordered set of the distinct non-\$ symbols in $Q^p[b, e]$. Then, the width of $[b_{\#_\$+i}, e_{\#_\$+i}]$ $(1 \le i \le n - \#_\$)$ is equal to the number of occurrences of the symbol c_i in $Q^p[b, e]$.

I_{X^1}	Lcp_1	X^1
0	-1	**$**
0	0	**$**
0	0	**$**
3	0	**C**TG$
1	0	**G**$
1	1	**G**$
2	1	**G**G$
3	1	**G**TT$
1	0	**T**$
2	1	**T**G$
2	1	**T**T$
3	1	**T**GG$

I_{X^2}	Lcp_2	X^2
0	-1	**$**
0	0	**$**
0	0	**$**
3	0	**CT**G$
1	0	**G$**
1	1	**G$**
2	1	**GG**$
3	1	**GT**T$
1	0	**T$**
2	1	**TG**$
3	2	**TG**G$
2	1	**TT**$

FIG. 6. The figure depicts the transition from encoding I_{X^1} to encoding I_{X^2} and from array Lcp_1 to array Lcp_2 for the set of reads presented in Figure 1. Note that I_{X^1} and I_{X^2} list the suffix lengths ([DOLLAR] excluded) according to X^1 and X^2 giving (respectively) the p_1 and p_2 order. Horizontal lines give the partitions in 1-segments of I_{X^1} and 2-segments of I_{X^2}. Each 1-segment produces at least one 2-segment. For example, the 1-segment on I_{X^1} related to the four suffixes starting with symbol T produces three 2-segments (on I_{X^2}) composed of 1, 2, and 1 suffixes, respectively. For each 1-segment that has more than one suffix, its suffixes have been sorted by their second symbol in X^2. Moreover, the elements of Lcp_2 that are not in starting positions of any 2-segment have been set to 2 since this corresponds to a 2-segment that has more than one suffix.

From that already described, it derives that the p segments on X^p form a partition of its positions $(1, \cdots, (k+1)m)$ that is a refinement of the partition formed by the $(p-1)$ segments on X^{p-1}.

All the $(k+1)$ segments of the encoding $I_{X^{(k+1)}}$ have width equal to 1. Moreover, if L is the length of the longest common substring of two strings in S, after $L+1$ iterations the two following properties hold: (1) the encoding $I_{X^{(L+1)}}$ is equal to $I_{X^{(k+1)}}$ and (2) each I_{X^j} with $j > L$ is identical to $I_{X^{(L+1)}}$. These two properties are a consequence of the following two observations: (i) the length p of the LCP between two strings is equal to the length of the longest common substring in S and (ii) if all the $(p+1)$ prefixes of the suffixes are distinct, then the $\prec_{p+2}$ relation does not affect the ordering given by $\prec_{p+1}$, that is $I_{X^{p+2}} = I_{X^{p+1}}$.

Finally, notice that all $(p+1)$ segments are independent, therefore, they can be computed in parallel from the p segments. Although our description of Algorithm 3 is sequential, it is possible to give a parallel version, since all $(p-1)$ segments on $I_{X^{p-1}}$ can be managed independently to obtain the induced p segments.

Figure 5 highlights that, at each iteration p, the elements of a p segment $[b, e]$ produce $(p+1)$ segments s.t. no $(p+1)$ segment starts before b or ends after e. Note that at each iteration, the array B is not computed and it is depicted only for ease of presentation. Furthermore, it is easy to verify that the correctness of Algorithm 3 is preserved if the loop at line 5 starts at $i \leftarrow b$ and ends at e, for some b and e that are the starting and ending positions of a p segment (not necessarily the same segment), provided that each $rank[j]$ represents the number of elements of $I_{X^{p-1}}[1, b]$ that are equal to j. Clearly, the correctness of the algorithm will be restricted to $I_{X^p}[b, e]$. Finally, note that if b' is the starting position of a p segment, then the values of array $rank$ at the beginning of the for loop of line 5 when $i = b'$ are equal to those of the same array in the subsequent invocation of Algorithm 3 (i.e., when $I_{X^{p+2}}$ is computed). Hence, a natural approach to compute in parallel I_{X^p} is to partition the interval $[1, (k+1)m]$ in several clusters of contiguous p segments that are independently processed by different threads.

3.2.1 Computing the LCP array and limiting the iterations. Let Lcp_p be the $((k+1)m)$ long array s.t. $Lcp_p[i]$ is the length of the LCP between the p prefix of suffix $X^p[i]$ and the p prefix of suffix $X^p[i-1]$. Algorithm 3 computes Lcp_p from Lcp_{p-1} along with the encoding I_{X^p} from encoding $I_{X^{p-1}}$. Clearly the last iteration $k+1$ computes Lcp_{k+1}, which is equal (by definition) to the LCP array of the input set S. As explained at the end of this section, computing the LCP array allows also to reduce the iterations (calls to Algorithm 3) to the number ($\leq k+1$) strictly necessary for computing the encoding I_B and the LCP array of the input set.

The LCP array is obtained by exploiting Proposition 6, which follows from the definition of p segment.

Proposition 6. *Let i be a position on the longest common prefix array LCP. Then $LCP[i]$ is the largest p s.t. i is the start of a p segment (of I_{X^p}) and is not the start of a $(p-1)$ segment (of $I_{X^{p-1}}$).*

At each iteration p, Algorithm 3 (see lines from 11 to 16) computes Lcp_p from Lcp_{p-1} by increasing each entry of Lcp_{p-1} that is not the starting position of a p segment. Note that B is the list storing the starting positions of the p segments obtained from a given $p-1$ segment $[b, e]$ since B is built by storing for each symbol c_h the number of symbols that are smaller in lexicographic order. Indeed, the partition of the $p-1$ segment $[b, e]$ into p segments is induced by the concatenation of the lists L_{c_h}, for $0 \leq h \leq \sigma$. Since the LCP between two empty strings is 0, the array Lcp_0 is set to all 0's (apart from the first position that is set to -1) before the first iteration (Algorithm 5). The following invariant, which directly implies the correctness of the procedure, is maintained at each iteration.

Lemma 7. *After the execution of Algorithm 3 to obtain I_{X^p}, $Lcp_p[i]=p$ iff i is not the start position of any p segment.*

Proof. We will prove the lemma by induction on p. The array Lcp_0 is set to all 0's (apart from the first position that is set to -1); therefore, we only have to consider the general case. Let $[b, e]$ be a $(p-1)$ segment, at the beginning of iteration p we have that $Lcp_{p-1}[i]=p-1$ for $b+1 \leq i \leq e$. Then, the procedure (Algorithm 3) sets to p the array Lcp_p in all positions of the induced p segments different from their begin positions, completing the proof. □

An immediate consequence of Lemma 7 is that a p segment $[b, e]$ of width 1 (that is when $b=e$) cannot be further partitioned into $(p+1)$ segments. From this it derives that when all the p segments of I_{X^p} have width 1, then $I_{X^p}=I_k$; that happens only when $Lcp_p[i] < p$ for all i. This fact implies that Algorithm 3 is called exactly $\max_i\{LCP[i]\}+1$ times.

3.2.2 Computing the Q_l^p arrays. This section is the most sophisticated from a technical point of view, especially because we want to spread the load over different processors by exploiting a parallel approach. First, we want to give an intuition of our procedure that builds the arrays Q_l^p from the B_l arrays.

First, recall that, for a given p, Q_l^p $(0 \leq l \leq k)$ is the m long array s.t. $Q_l^p[i]$ is the p-th symbol of the suffix $X_l[i]$ if $p \leq l$, or $Q_l^p[i]=\$$ otherwise. In other words, each $Q_l^p[i]$ is the p-th character of the suffix in $X_l[i]$.

Algorithm 4: Compute all lists Q_l^p for any given $p \geq 2$.

Input: The lists $B_0, \ldots, B_k$ on alphabet $c_0, \cdots, c_\sigma$, an integer p with $2 \leq p \leq k$, and all Q_l^{p-1}.
Output: The lists Q_l^p for each $k \geq l \geq p$

1 **for** $l \leftarrow p$ *to* k **do**
2 | $Q_l^p \leftarrow$ empty list;
3 | **for** $h \leftarrow 0$ *to* σ **do**
4 | | $Q_l^p(c_h) \leftarrow$ empty list;
5 | **for** $j \leftarrow 1$ *to* m **do**
6 | | Append $Q_{l-1}^{p-1}[j]$ *to* $Q_l^p(B_{l-1}[j])$;
7 | **for** $h \leftarrow 0$ *to* σ **do**
8 | | Append $Q_l^p(c_h)$ *to* Q_l^p;

We now show how to compute the arrays $Q_0^p, \ldots, Q_k^p$. We first present the following proposition that establishes a recursive definition of Q_l^p that is fundamental for Algorithm 4.

Proposition 8. *Let X_l and X_{l-1} be, respectively, the sorted l suffixes and $(l-1)$ suffixes of the set S. Let α_l and α_{l-1} be, respectively, the l suffix and the $(l-1)$ suffix of a generic input string s_i. Then the p-th symbol of α_l is the (p – 1)-th symbol of α_{l-1}.*

Algorithm 4 shows how to compute all the lists Q_l^p iteratively from Q_{l-1}^{p-1} exploiting Proposition 8. We note that, for $l \geq 1$, Q_l^1 is the result of sorting B_{l-1}, whereas Q_0^1 is a sequence of sentinels. Therefore, the arrays $Q_0^1, \ldots, Q_k^1$ can be trivially computed and form the initial step of the recursion.

To prove the correctness of Algorithm 4, we need to show that the permutation St^{l-1} over indexes $1, \ldots, m$ of B_{l-1} induced by the lexicographic ordering of B_{l-1} is the correct permutation of Q_{l-1}^{p-1} to obtain Q_l^p. Indeed, observe that St^{l-1} is the permutation that relates positions of indexes of strings in X_{l-1} to their positions in X_l. More precisely, given a string s_q of S, s.t. its $(l-1)$ suffix is in position j of list X_{l-1}, then if $St^{l-1}[j]=t$, it means that the l suffix is of the string s_q is in position t of list X_l. The mentioned observation is a consequence of the fact that to get the lexicographic ordering of X_l from the list X_{l-1}, we simply sort the $(l-1)$ suffixes by the first symbol that precedes them, that is, they are sorted by the list B_{l-1}.

Finally, Algorithm 5 shows how to combine Algorithms 3 and 4 to compute I_{X^k} from the input arrays $B_0, \ldots, B_k$.

We now describe how the performance of Algorithm 4 can be improved by using multiple threads. We recall that, at iteration p, each array Q_l^p can be computed independently as a permutation of array Q_{l-1}^{p-1} (the procedure is exemplified in Fig. 4). Moreover, we note that at each iteration p, each one of the arrays $Q_0^p, \ldots, Q_{p-1}^p$ is composed by a sequence of m sentinel symbols and it is possible to reuse the same arrays computed at the previous step. The remaining arrays $Q_p^p, \ldots, Q_k^p$ can be computed in parallel from arrays $Q_{p-1}^{p-1}, \ldots, Q_{k-1}^{p-1}$. Therefore, the for loop at line 1 of Algorithm 4 can be run in parallel using up to k threads.

Algorithm 5: Computation of the interleave

Input : The arrays $B_0, B_1, \ldots, B_k$
Output: The encoding I_{X^k}.
1 **for** $l \leftarrow 0$ *to* k **do**
2 | **for** $i \leftarrow 1$ *to* m **do**
3 | | $I_{X^0}[lm+i] \leftarrow l$; $Lcp[lm+i] \leftarrow 0$;
4 $Lcp[1] = -1$;
5 Compute lists Q_l^1 for $0 \leq l \leq k$;
6 $p \leftarrow 1$;
7 **while** *there exists some* $(p-1)$ *segments on* $I_{X^{p-1}}$ *that is wider than* 1 **do**
8 | Compute I_{X^p} from $I_{X^{p-1}}$;
9 | Compute lists Q_l^{p+1} for $0 \leq l \leq k$;
10 Output I_{X^p};

3.3. Time complexity

The overall time complexity of the method is $\mathcal{O}(kmL)$. Trivially, computing arrays S_l and B_l requires $\mathcal{O}(mk)$. The second step (Algorithm 5) requires $\mathcal{O}(kmL)$ time since initializing the support data structures (lines 1–6) and each call in lines 8 and 9 require $\mathcal{O}(km)$. Moreover, as said before, the while loop at lines 7–9 is executed $L + 1$ times, where L is the length of the longest common substring in the input set.

Notice that some parts of the algorithm can be computed in parallel. Most precisely, each $(p-1)$ segment of Algorithm 3 can be processed independently, by keeping a queue of the segments that have not been processed yet. In this case, the parallelism can be obtained with a simple producers–consumer technique that spreads the workload over the desired number of threads. A similar argument can be applied to Algorithm 4. For all those cases, the running time is effectively divided by the number t of threads. Still, we have to point out that some parts of the algorithm, such as the construction of the N_l arrays, are sequential, hence the overall time complexity is not $\mathcal{O}(kmL/t)$.

Finally, we notice that our $\mathcal{O}(mkL)$ time complexity is no worse than that of the best known lightweight approaches.

4. EXPERIMENTAL ANALYSIS

To assess the performance of our method, we implemented a prototype in C++, called bwt_lcp,* and tested it on eight data sets containing sequences representing biological data (usually referred as NGS reads in the bioinformatics field). More precisely, we tested our tool in two scenarios: the first consists of 148 base long reads produced by the well-known Genome in a Bottle consortium extracted from the NA24385 individual, whereas the second consists of random sequences of length 151 over the alphabet $\{A, C, G, T\}$ produced by a simple Python script.

The main difference between the two scenarios is that the first scenario includes duplicated sequences and represents the worst possible input for bwt_lcp (recall that the complexity of our method is $\mathcal{O}(mkL)$, thus it depends on the length of the longest common substring between the input strings), whereas in the second scenario the length of the longest common substring is roughly equal to 20% of the length of the sequences. For each scenario, we produced four data sets composed of 8, 16, 32, and 64 million sequences, for a total of 8 data sets. We will refer to the data sets of the first scenario as giab-8, giab-16, giab-32, and giab-64, whereas we will refer to the data sets of the second scenario as random-8, random-16, random-32, and random-64.

Many methods to compute the BWT and the LCP array were proposed in the literature, but only few of them build these data structures for a set of strings (i.e., the multistring version of the two), and even fewer try to exploit the multithreaded architecture of modern machines. A simple, yet widely applied, workaround for building the BWT and LCP array of a set of m strings by means of methods designed for a single string is to concatenate the input strings interposing different sentinel symbols between them. More precisely, given a set $S = \{s_1, s_2, \ldots, s_m\}$ of strings built over the alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, and m sentinel

*The software is freely available at https://github.com/AlgoLab/bwt-lcp-parallel.

symbols $\$_1 < \$_2 < \ldots < \$_m < c_1$ not in the alphabet, a text $T = s_1 \cdot s_2 \ldots s_{m-1} \cdot \$_{m-1} \cdot s_m \cdot \$_m$ is built and the BWT and the LCP array are computed for T.

Thus, as a first analysis, we compared bwt_lcp with these classical methods. We note that many of them are implemented in the sdsl-lite C++ library (Gog et al., 2014), therefore, we developed a software that first concatenates the strings in input as described before and then uses the functions provided by the library to compute the BWT and the LCP array. More precisely, such functions build the data structures by applying the methods presented in Larsson and Sadakane (2007) and in Kasai et al. (2001), which first build the Suffix Array of the input string and then create the BWT and the LCP array. We want to highlight that even though other methods are implemented in sdsl-lite, we could not test some of them since they only work for byte-alphabets [e.g., the method proposed in Beller et al. (2013) and Gog and Ohlebusch (2011)]. We, therefore, decided to use the method provided by default by the library. We will refer to this software as bl_sdsl.

Another tool we compared with is eGap (Egidi et al., 2018), an implementation of a recently proposed method to compute the BWT and the LCP array of a set of strings. This tool is an external memory algorithm that aims to build both data structures by merging together the data structures built over every string in input, and by efficiently using the available memory.

Finally, since bwt_lcp is designed to run using multiple threads, we ran it using 1, 2, 4, 8, and 16 threads. We measured bwt_lcp, bl_sdsl, and eGap performance by the amount of time required to complete their run; to this aim, we used the GNU time command to track the elapsed time (wall clock time) of the computation. The results for each data set are reported in Tables 1 and 2.

We can note that bl_sdsl is faster than eGap on the giab data sets, whereas eGap is faster than bl_sdsl on the random data set. This behavior was expected since, similarly to bwt_lcp, the time required by eGap to complete the computation depends on the longest common substring between two strings in input, hence, duplicated strings in input will degrade eGap's running time.

bwt_lcp is always faster than bl_sdsl and eGap when using two or more threads and the gap in performance increases with bigger data sets. Indeed, note that bwt_lcp is faster than bl_sdsl and eGap even when using only a single thread on the bigger data sets composed of >32 million reads (data sets giab-32, giab-64, random-32, and random-64).

We note that the speedup of bwt_lcp is not linear and decreases when increasing the number of threads. This is mostly due to the output step of bwt_lcp that is inevitably single threaded.

Table 1. Comparison of the Tools on the Giab-8, Giab-16, Giab-32, and Giab-64 Data Sets

giab-8		*giab-16*	
Tool	*Time*	*Tool*	*Time*
bwt_lcp (16)	1105	bwt_lcp (16)	2343
bwt_lcp (8)	1330	bwt_lcp (8)	2971
bwt_lcp (4)	3465	bwt_lcp (4)	5011
bwt_lcp (2)	3165	bwt_lcp (2)	6570
bl_sdsl	3316	bl_sdsl	9053
bwt_lcp (1)	3391	bwt_lcp (1)	11,925
eGap	5751	EGap	13,519

giab-32		*giab-64*	
Tool	*Time*	*Tool*	*Time*
bwt_lcp (16)	5273	bwt_lcp (16)	12,284
bwt_lcp (8)	6961	bwt_lcp (8)	16,144
bwt_lcp (4)	11,772	bwt_lcp (4)	23,142
bwt_lcp (2)	13,214	bwt_lcp (2)	28,212
bwt_lcp (1)	23,618	bwt_lcp (1)	49,756
bl_sdsl	24,245	bl_sdsl	67,430
eGap	41,473	eGap	86,062

The *Time* columns report the seconds required by each tool to complete the BWT and LCP array computation. The number of threads used is reported between parentheses.

BWT, Burrows–Wheeler transform; LCP, longest common prefix.

TABLE 2. COMPARISON OF THE TOOLS ON THE RANDOM-8, RANDOM-16, RANDOM-32, AND RANDOM-64 DATA SETS

Random-8		*Random-16*	
Tool	*Time*	*Tool*	*Time*
bwt_lcp (16)	431	bwt_lcp (16)	962
bwt_lcp (8)	483	bwt_lcp (8)	1132
bwt_lcp (4)	677	bwt_lcp (4)	1500
bwt_lcp (2)	1002	bwt_lcp (2)	2229
eGap	1598	EGap	3536
bwt_lcp (1)	1701	bwt_lcp (1)	3752
bl_sdsl	4134	bl_sdsl	9774
Random-32		*Random-64*	
Tool	*Time*	*Tool*	*Time*
bwt_lcp (16)	1959	bwt_lcp (16)	4165
bwt_lcp (8)	2209	bwt_lcp (8)	4880
bwt_lcp (4)	3018	bwt_lcp (4)	6312
bwt_lcp (2)	4460	bwt_lcp (2)	9449
bwt_lcp (1)	7552	bwt_lcp (1)	16,192
eGap	9575	eGap	19,914
bl_sdsl	26,846	bl_sdsl	61,381

The *Time* columns report the seconds required by each tool to complete the BWT and LCP array computation. The number of threads used is reported between parentheses.

As a last measure, we tracked the main memory requirement of each tool by using the GNU time command. For ease of presentation, we only present here the results for the bigger data set (giab-64) since the behavior of the tool is consistent across the various tests. To compute the BWT and the LCP of giab-64, bwt_lcp required 2.3, 3.3, 4.9, 8.5, and 10.3 GB when running with 1, 2, 4, 8, and 16 threads, respectively. The increase in required memory is mostly due to the local variable that each thread stores to compute the final result.

In contrast, bl_sdsl required 81.5 GB of RAM to compute both the BWT and the LCP array of giab-64, whereas eGap required 9.3 GB.

Overall, bwt_lcp is faster than both bl_sdsl and eGap and requires an amount of main memory comparable with that of eGap, while being more memory efficient than bl_sdsl.

5. CONCLUSIONS

We have described a divide and conquer approach for computing the multistring BWT and LCP array with worst-case $\mathcal{O}(mkL)$ time complexity. Moreover, our approach can be easily implemented using multiple threads, with a clear decrease of the running time, which we have investigated experimentally.

Future work will be devoted to improving, also from a theoretical point of view, the current $\mathcal{O}(kmL)$ time complexity, which is still an open problem.

AUTHOR DISCLOSURE STATEMENT

The authors declare that no competing financial interests exist.

REFERENCES

Bauer, M., Cox, A., Rosone, G., et al. 2012. Lightweight LCP construction for next-generation sequencing datasets, 326–337. In: Raphael, B., and Tang, J. (Eds): *WABI.* Springer, Berlin-Heidelberg.

Bauer, M., Cox, A., and Rosone, G. 2013. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comp. Sci.* 483, 134–148.

Belazzougui, D., Gagie, T., Mäkinen, V., et al. 2016. Bidirectional variable-order de bruijn graphs, 164–178. In: Kranakis, E., Navarro, G., and Chavez, E. (Eds): *LATIN*. Springer, Berlin-Heidelberg.

Beller, T., Gog, S., Ohlebusch, E., et al. 2013. Computing the longest common prefix array based on the burrows-wheeler transform. *J. Discrete Algorithms*. 18, 22–31.

Beretta, S., Bonizzoni, P., Denti, L., et al. 2017. Mapping rna-seq data to a transcript graph via approximate pattern matching to a hypertext, 49–61. In: Figueiredo, D., Martin-Vide, C., Pratas, D., and Vega-Rodriguez, M. (Eds): *AlCoB*. Springer, Cham.

Bonizzoni, P., Della Vedova, G., Pirola, Y., et al. 2016. LSG: An external-memory tool to compute string graphs for next-generation sequencing data assembly. *J. Comput. Biol*. 23, 137–149.

Bonizzoni, P., Della Vedova, G., Pirola, Y., et al. 2017a. Computing the BWT and LCP array of a set of strings in external memory. *CoRR*, abs/1705.07756. Available at: http://arxiv.org/abs/1705.07756 Accessed 3/22/19.

Bonizzoni, P., Della Vedova, G., Pirola, Y., et al. 2017b. FSG: Fast string graph construction for de novo assembly. *J. Comput. Biol*. 24, 953–968.

Bonizzoni, P., Della Vedova, G., Nicosia, S., et al. 2018. Divide and conquer computation of the multi-string BWT and LCP array. In *Conference on Computability in Europe*. 107–117.

Burrows, M., and Wheeler, D.J. 1994. A block-sorting lossless data compression algorithm. Technical Report, Digital Systems Research Center.

Cox, A., Garofalo, F., Rosone, G., et al. 2016. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms* 37, 17–33.

Denti, L., Rizzi, R., Beretta, S., et al. 2018. ASGAL: Aligning RNA-Seq data to a splicing graph to detect novel alternative splicing events. *BMC Bioinformatics* 19, 444.

Egidi, L., Alves Louza, F., Manzini, G., et al. 2018. External memory BWT and LCP computation for sequence collections with applications, 1–10, 14. In *18th International Workshop on Algorithms in Bioinformatics, WABI*, Dagstuhl, Germany.

Ferragina, P., and Manzini, G. 2005. Indexing compressed text. *J. ACM* 52, 552–581.

Ferragina, P., Luccio, F., Manzini, G., et al. 2009. Compressing and indexing labeled trees, with applications. *J. ACM* 57, 4:1–4:33.

Gagie, T., Manzini, G., and Sirén, J. 2017. Wheeler graphs: A framework for bwt-based data structures. *Theor. Comput. Sci.* 698, 67–78.

Gog, S., Beller, T., Moffat, A., et al. 2014. From theory to practice: Plug and play with succinct data structures, 326–337. In *13th International Symposium on Experimental Algorithms (SEA 2014)*. Springer, Cham.

Gog, S., and Ohlebusch, E. 2011. Fast and lightweight lcp-array construction algorithms, 25–34. *In* Müller-Hannemann, M., and Werneck, R.F.F., eds., *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2011*, Holiday Inn San Francisco Golden Gateway, San Francisco, CA, January 22, 2011. SIAM.

Holt, J., and McMillan, L. 2014. Merging of multi-string BWTs with applications. *Bioinformatics* 30, 3524–3531.

Kasai, T., Lee, G., Arimura, H., et al. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications, 181–192. In Amir, A., and Landau, G.M., eds. *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*. Springer.

Larsson, N.J., and Sadakane, K. 2007. Faster suffix sorting. *Theor. Comput. Sci.* 387, 258–272.

Li, H. 2014. Fast construction of FM-index for long sequence reads. *Bioinformatics* 30, 3274–3275.

Li, H., and Durbin, R. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 25, 1754–1760.

Mantaci, S., Restivo, A., Rosone, G., et al. 2007. An extension of the Burrows–Wheeler Transform. *Theor. Comput. Sci*. 387, 298–312.

Myers, E. 2005. The fragment assembly string graph. *Bioinformatics* 21(suppl. 2), ii79–ii85.

Rosone, G., and Sciortino, M. 2013. The Burrows-Wheeler transform between data compression and combinatorics on words, 353–364. In: Bonizzoni, P., Brattka, V., and Loewe, B. (Eds): *CiE 2013*. Springer, Berlin-Heidelberg.

Simpson, J., and Durbin, R. 2010. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics* 26, i367–i373.

Staden, R. 1979. A strategy of DNA sequencing employing computer programs. *Nucl. Acids Res*. 6, 2601–2610.

Address correspondence to:
Prof. Paola Bonizzoni
Dipartimento di Informatica Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336
20126 Milan
Italy

E-mail: bonizzoni@disco.unimib.it

This article has been cited by:

1. Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, Raffaella Rizzi. 2021. Computing the multi-string BWT and LCP array in external memory. *Theoretical Computer Science* **862**, 42-58. [Crossref]
2. Veronica Guerrini, Felipe A. Louza, Giovanna Rosone. 2020. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinformatics* **21**:S8. . [Crossref]
3. Lavinia Egidi, Giovanni Manzini. 2020. Lightweight merging of compressed indices based on BWT variants. *Theoretical Computer Science* **812**, 214-229. [Crossref]
4. Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, Paola Bonizzoni. 2019. Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. *Quantitative Biology* **7**:4, 278-292. [Crossref]