Engineering External Memory LCP Array Construction: Parallel, In-Place and Large Alphabet

Juha Kärkkäinen[1] and Dominik Kempa[2]

1 **Department of Computer Science, University of Helsinki, Helsinki, Finland**
juha.karkkainen@cs.helsinki.fi

2 **Helsinki Institute for Information Technology HIIT, Helsinki, Finland; and Department of Computer Science, University of Helsinki, Helsinki, Finland**
dominik.kempa@cs.helsinki.fi

Abstract

The suffix array augmented with the LCP array is perhaps the most important data structure in modern string processing. There has been a lot of recent research activity on constructing these arrays in external memory. In this paper, we engineer the two fastest LCP array construction algorithms (ESA 2016) and improve them in three ways. First, we speed up the algorithms by up to a factor of two through parallelism. Just 8 threads is sufficient for making the algorithms essentially I/O bound. Second, we reduce the disk space usage of the algorithms making them in-place: The input (text and suffix array) is treated as read-only and the working disk space never exceeds the size of the final output (the LCP array). Third, we add support for large alphabets. All previous implementations assume the byte alphabet.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases LCP array, suffix array, external memory algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.17

1 Introduction

The suffix array [12, 3], a lexicographically sorted list of the suffixes of a text, is one of the most important data structures in modern string processing. It is frequently augmented with the longest-common-prefix (LCP) array, which stores the lengths of the longest common prefixes between lexicographically adjacent suffixes. Together they are the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [13]. Modern textbooks spend dozens of pages in describing their applications, see e.g. [14, 11].

The construction of the two arrays is a bottleneck in many applications. There has been a lot of recent research on external memory construction of these data structures. Here we are interested in the construction of the LCP array given the suffix array and the text. The two fastest external memory algorithms for this task are currently EM-SΦ and EM-SI, recently introduced in [4]. In this paper, we improve EM-SΦ and EM-SI in several ways.

First, we modify both algorithms to use multiple threads during their execution. Parallelization does not reduce or speed up I/O as such, but it can speed up those stages that are dominated by computation rather than I/O. Cache misses in particular can be expensive enough to dominate I/O. Our experimental results show that EM-SI benefits only a little from parallelization, but the speed of EM-SΦ improves by as much as a factor of two for

© Juha Kärkkäinen and Dominik Kempa;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 17; pp. 17:1–17:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

some texts. Eight threads is sufficient to achieve an essentially maximum speed up as the computation becomes I/O bound.

Second, we reduce the disk space usage of both algorithms. Disk space usage can be more crucial than speed, because a lack of sufficient free disk space can prevent a computation entirely. We make both algorithms in-place in the sense that the disk space usage never exceeds what is needed for the input (the text, the suffix array, and for EM-SI, the Burrows–Wheeler transform (BWT)) and the output (the LCP array). Thus any machine that has sufficient disk space for the inputs and the outputs can run these algorithms. The input is treated as read-only, i.e., it is never deleted or written over even temporarily. The working disk space, i.e., disk space used in addition to the inputs, is reduced by more than a factor of two in some cases. The fully in-place computation slows down the algorithms but never more than 36% and often much less.

Third, we modify both algorithms to handle large alphabets. All previous implementations work only for the byte alphabet. While it is possible to split large characters into multiple bytes, construct suffix and LCP arrays for the resulting text over the byte alphabet, and then post-process to construct the desired arrays, this requires much more time and disk space than using algorithms that can handle large alphabets natively. We demonstrate this for EM-SΦ and EM-SI in our experiments.

Related work. Suffix array construction in external memory has a long history. The most recent addition is fSAIS [8], which is also the first implementation able to handle large alphabets natively. LCP array construction in external memory has been studied much less. It was first achieved by modifying suffix array construction to produce the LCP array simultaneously [2]. Independent construction of the LCP array given the suffix array as input is preferable and was first achieved by LCPScan [5]. The only further practical improvements are EM-SΦ and EM-SI. A very recent theoretical break-through is the first LCP array construction algorithm [6] with I/O complexity $\mathcal{O}(\mathrm{sort}(n))$ for a text of length n, where $\mathrm{sort}(n)$ is the complexity of sorting n integers, but it is not competitive in practice. Another recent result is an algorithm for computing the succinct representation of the PLCP (permuted LCP) array in external memory [16].

2 Basic Data Structures

Throughout we consider a string $\mathsf{X} = \mathsf{X}[0..n) = \mathsf{X}[0]\mathsf{X}[1]\ldots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from an alphabet of size σ. Here and elsewhere we use $[i..j)$ as a shorthand for $[i..j-1]$. For $i \in [0..n]$, we write $\mathsf{X}[i..n)$ to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}[i..n) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n-1]$. We will often refer to suffix $\mathsf{X}[i..n)$ simply as "suffix i".

The *suffix array* [12, 3] of X is an array $\mathrm{SA} = \mathrm{SA}[0..n]$ which contains a permutation of the integers $[0..n]$ such that $\mathsf{X}[\mathrm{SA}[0]..n) < \mathsf{X}[\mathrm{SA}[1]..n) < \cdots < \mathsf{X}[\mathrm{SA}[n]..n)$. In other words, $\mathrm{SA}[j] = i$ iff $\mathsf{X}[i..n)$ is the $(j+1)^{\mathrm{th}}$ suffix of X in ascending lexicographical order. Another representation of the permutation is the *Φ array* [9] $\Phi[0..n)$ defined by $\Phi[\mathrm{SA}[j]] = \mathrm{SA}[j-1]$ for $j \in [1..n]$. In other words, the suffix $\Phi[i]$ is the immediate lexicographical predecessor of the suffix i, and thus $\mathrm{SA}[n-k] = \Phi^k[\mathrm{SA}[n]]$ for $k \in [0..n]$. An example illustrating the arrays is given in Table 1.

Let $\mathrm{lcp}(i,j)$ denote the length of the longest-common-prefix (LCP) of suffix i and suffix j. For instance, in the example of Table 1, $\mathrm{lcp}(0,6) = 3 = |\mathtt{bab}|$ and $\mathrm{lcp}(7,4) = 5 = |\mathtt{abbab}|$. The *longest-common-prefix array* [12, 10], $\mathrm{LCP}[1..n]$, is defined such that $\mathrm{LCP}[i] = \mathrm{lcp}(\mathrm{SA}[i], \mathrm{SA}[i-1])$ for $i \in [1..n]$. The *permuted LCP array* [9] $\mathrm{PLCP}[0..n)$ is the LCP array

Table 1 Examples of the arrays used by the algorithms for the text $\mathsf{X} = \texttt{babaabbabbab}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathsf{X}[i]$	b	a	b	a	a	b	b	a	b	b	a	b	-
$\mathrm{SA}[i]$	12	3	10	1	7	4	11	2	9	0	6	8	5
$\mathrm{BWT}[i]$	b	b	b	b	b	a	a	a	b	$	b	a	a
$\Phi[i]$	9	10	11	12	7	8	0	1	6	2	3	4	-
$\mathrm{LCP}[i]$	-	0	1	2	2	5	0	1	2	3	3	1	4
$\mathrm{PLCP}[i]$	3	2	1	0	5	4	3	2	1	2	1	0	-
$i + \mathrm{PLCP}[i]$	3	3	3	3	9	9	9	9	9	11	11	11	-
$2i + \mathrm{PLCP}[i]$	3	4	5	6	13	14	15	16	17	20	21	22	-
$\mathrm{PLCP}_4^{\mathrm{lo}}(i)$	3	2	1	0	5	4	3	2	1	0	0	0	-
$\mathrm{PLCP}_4^{\mathrm{lo}}(i)$	3	8	7	6	5	4	3	2	1	2	1	0	-

permuted from the lexicographical order into the text order, i.e., $\mathrm{PLCP}[\mathrm{SA}[j]] = \mathrm{LCP}[j]$ for $j \in [1..n]$. Then $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \Phi[i])$ for all $i \in [0..n)$. Table 1 shows example LCP and PLCP arrays.

The row $i + \mathrm{PLCP}[i]$ in Table 1 illustrates (the first part of) the following property of the PLCP array, which is the basis of all efficient algorithms for LCP array construction.

▶ **Lemma 1** ([5]). *Let $i, j \in [0..n)$. If $i \le j$, then $i + \mathrm{PLCP}[i] \le j + \mathrm{PLCP}[j]$. Symmetrically, if $\Phi[i] \le \Phi[j]$, then $\Phi[i] + \mathrm{PLCP}[i] \le \Phi[j] + \mathrm{PLCP}[j]$.*

The *succinct PLCP array* [15] $\mathrm{PLCP}_{\mathrm{succ}}[0..2n)$ represents the PLCP array using $2n$ bits. Specifically, $\mathrm{PLCP}_{\mathrm{succ}}[j] = 1$ if $j = 2i + \mathrm{PLCP}[i]$ for some $i \in [0..n)$, and $\mathrm{PLCP}_{\mathrm{succ}}[j] = 0$ otherwise. Notice that the value $2i + \mathrm{PLCP}[i]$ is unique for each i by Lemma 1 as illustrated in Table 1.

For $q \ge 1$, the *sparse PLCP array* $\mathrm{PLCP}_q[0..\lceil n/q \rceil)$ is defined by $\mathrm{PLCP}_q[i] = \mathrm{PLCP}[iq]$, i.e., it contains every q^{th} entry of PLCP. We also define $\Phi_q[0..\lceil n/q \rceil)$ by $\Phi_q[i] = \Phi[iq]$ so that $\mathrm{PLCP}_q[i] = \mathrm{lcp}(qi, \Phi_q[i])$. The sparse PLCP array can be used as a compact representation of the full PLCP array because the other entries can be bounded using the following lemma.

▶ **Lemma 2** ([9]). *For any $i \in [0..n)$, let*

$$\mathrm{PLCP}_q^{\mathrm{lo}}(i) = \max(0, \mathrm{PLCP}_q[\lfloor i/q \rfloor] - (i - q\lfloor i/q \rfloor))$$

$$\mathrm{PLCP}_q^{\mathrm{hi}}(i) = \begin{cases} \mathrm{PLCP}_q[\lceil i/q \rceil] + (q\lceil i/q \rceil - i) & \text{if } q\lceil i/q \rceil < n \\ n - i - 1 & \text{otherwise} \end{cases}$$

Then $\mathrm{PLCP}_q^{\mathrm{lo}}(i) \le \mathrm{PLCP}[i] \le \mathrm{PLCP}_q^{\mathrm{hi}}(i)$.

Although the difference $\mathrm{PLCP}_q^{\mathrm{hi}}(i) - \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ has no non-trivial limit for an individual i, the sum of the differences is bounded by the following lemma.

▶ **Lemma 3** ([9]). $\sum_{i \in [0..n)} \mathrm{PLCP}_q^{\mathrm{hi}}(i) - \mathrm{PLCP}_q^{\mathrm{lo}}(i) \le (q-1)n + q^2$.

The *Burrows–Wheeler transform* $\mathrm{BWT}[0..n]$ of X is defined by $\mathrm{BWT}[i] = \mathsf{X}[\mathrm{SA}[i] - 1]$ if $\mathrm{SA}[i] > 0$ and otherwise $\mathrm{BWT}[i] = \$$, where $\$$ is a special symbol that does not appear in the text. We say that an lcp value $\mathrm{LCP}[i] = \mathrm{PLCP}[\mathrm{SA}[i]]$ is *reducible* if $\mathrm{BWT}[i] = \mathrm{BWT}[i-1]$ and *irreducible* otherwise. The significance of reducibility is summarized in the following two lemmas.

▶ **Lemma 4** ([9]). *If $\mathrm{PLCP}[i]$ is reducible, then $\mathrm{PLCP}[i] = \mathrm{PLCP}[i-1] - 1$ and $\Phi[i] = \Phi[i-1] + 1$.*

▶ **Lemma 5** ([9, 7]). *The sum of all irreducible lcp values is $\le n \log n$.*

3 Basic Algorithms

In this section, we describe the basic algorithms EM-SΦ and EM-SI introduced in [4]. Some details are omitted and others only sketched; we refer to [4] for full details.

3.1 EM-SΦ Algorithm

The first algorithm EM-SΦ gets the text X and the suffix array SA as input, and performs the following steps:

1. Compute PLCP_q for q chosen so that PLCP_q fits in RAM. During this step, we consider the text divided into *segments* that fit in RAM.
   a. Compute Φ_q by scanning SA.
   b. Generate all pairs $(i, \Phi[i])$ such that i is a multiple of q using Φ_q. Write each pair $(i, \Phi[i])$ to disk into the file associated with the text segment that contains $\Phi[i]$. Notice that the pairs in each file are naturally sorted by i.
   c. For each text segment, load the segment into RAM. Read the pairs $(i, \Phi[i])$ from the associated file while simultaneously scanning the full text so that the position $\mathsf{X}[i]$ is reached when the pair $(i, \Phi[i])$ is processed. For each pair, compute $\mathrm{lcp}(i, \Phi[i])$ and write it to disk into a separate file for each segment. When computing $\ell = \mathrm{lcp}(i, \Phi[i])$ we use the fact that $\ell \geq \mathrm{lcp}(i', \Phi[i']) - (i - i')$ by Lemma 1, where $(i', \Phi[i'])$ is the pair processed just previously. This ensures that the text scan never needs to backtrack.
   d. Construct PLCP_q in RAM by reading each value $\mathrm{PLCP}_q[i]$ from the file associated with the text segment containing $\Phi_q[i]$.
2. Compute LCP using PLCP_q based on Lemma 2. During this step, we consider the text divided into *half-segments* so that two half-segments fit in RAM. Every possible pair of half-segments is loaded into RAM once.
   a. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair use PLCP_q (stored in RAM) to compute the lower bound $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ (and the upper bound $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$) for $\mathrm{PLCP}[i]$ using Lemma 2. Write the pair $(i + \ell, \Phi[i] + \ell)$ to disk, where there is a separate file for each pair of half-segments. If $\mathrm{PLCP}_q^{\mathrm{lo}}(i) = \mathrm{PLCP}_q^{\mathrm{hi}}(i)$, no pair is written since we already know the exact lcp value.
   b. For each pair of half-segments, load them to RAM and compute $\mathrm{lcp}(j, k)$ for each pair (j, k) obtained from the associated file. The resulting value $\mathrm{lcp}(j, k)$ is written to disk to a separate file for each pair of half-segments.
   c. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair, compute $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$ (and $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$) as in step 2(a) and read the value $\ell' = \mathrm{lcp}(i + \ell, \Phi[i] + \ell)$ from the appropriate file. Then $\mathrm{PLCP}[i] = \ell + \ell'$ is the next value in the LCP array.

In Step 1(c), the computation of the lcp value may overflow the segment, i.e., $\Phi[i]$ is in the segment but $\Phi[i] + \ell$ is not. To deal with an overflow, we have in RAM an overflow buffer (of size of one disk block) containing the beginning of the next segment. An overflow beyond even the overflow buffer is handled by reading from disk. Similar overflows can occur in Step 2(b). There too we use overflow buffers but comparisons beyond the overflow buffers are simply aborted. In cases, where such an aborted comparison is possible (based on the upper bound $\mathrm{PLCP}_q^{\mathrm{hi}}(i)$), extra pairs are generated in Step 2(a) to continue the potentially aborted comparison when the appropriate pair of half-segments is in RAM. We refer to [4] for further details and analysis of overflow handling.

3.2 EM-SI Algorithm

The second algorithm EM-SI gets X, SA and BWT as input, and has the following steps:

1. Compute the succinct PLCP array $\text{PLCP}_{\text{succ}}$. For this step, we consider $\text{PLCP}_{\text{succ}}$ divided into segments that fit in RAM and the text divided into half-segments so that two half-segments fit in RAM.
   a. Scan SA and BWT to form a pair $(i, \Phi[i])$ for each i such that $\text{PLCP}[i]$ is irreducible. The pairs are written to disk where there is a separate file for each pair of text half-segments.
   b. Compute the bitvector $\mathsf{R}[0..n]$, where $\mathsf{R}[i] = 1$ iff $\text{PLCP}[i]$ is irreducible. If R does not fit in RAM, it is computed one RAM-sized segment at a time. The irreducible positions are determined either by scanning SA and BWT or by scanning the irreducible $(i, \Phi[i])$ pairs produced in Step 1(a), whichever takes less I/O.
   c. For each pair of text half-segments, load them to RAM and compute $\text{PLCP}[i] = \text{lcp}(i, \Phi[i])$ for each pair $(i, \Phi[i])$ obtained from the associated file. For each computed $\text{PLCP}[i]$, we write the value $2i + \text{PLCP}[i]$ to disk into a separate file for each $\text{PLCP}_{\text{succ}}$ segment.
   d. For each $\text{PLCP}_{\text{succ}}$ segment, initialize it with zeros in RAM, read the values from the associated file and set the corresponding bits to 1. Then read the corresponding part of R to determine the reducible lcp values using Lemma 4 and set the corresponding bits of $\text{PLCP}_{\text{succ}}$ to 1. See [4] for details.
2. Compute LCP from $\text{PLCP}_{\text{succ}}$. For this step, we consider the full PLCP array (not $\text{PLCP}_{\text{succ}}$) divided into segments that fit into RAM.
   a. Scan SA and write each value $\text{SA}[i]$ to disk into the file associated with the PLCP segment that contains the position $\text{SA}[i]$.
   b. For each PLCP segment, create it in RAM by scanning the relevant part of $\text{PLCP}_{\text{succ}}$. Then read the $\text{SA}[i]$ values from the associated file, compute $\text{LCP}[i] = \text{PLCP}[\text{SA}[i]]$, and write it to disk into a separate file for each segment.
   c. Scan SA, and for each $i \in [0..n)$, read $\text{LCP}[i]$ from the file associated with the PLCP segment that contains $\text{SA}[i]$ and write it to the final LCP file.

Overflows in Step 1(c) are handled as in Step 1(c) of EM-SΦ: using overflow buffers, and when that is not enough, reading directly from disk.

4 Parallelization

We have implemented both algorithms to use multiple threads during most stages of the computation. In both algorithms, several stages process a sequence of items so that the computation for one item is independent of other items and takes approximately the same time for all items. Such computation is trivial to parallelize: load a bufferful of items at a time to RAM and split the buffer evenly between threads. Below we describe the parallelization only for more complicated stages.

4.1 Parallelizing EM-SΦ

The first more complicated stage in EM-SΦ is Step 1(c). Here the algorithm processes a sequence of $(i, \Phi[i])$ pairs for each text segment, and the computation for each pair depends on the preceding pair. In the parallel version, the full sequence of pairs on disk is evenly split among the threads, and each thread processes its part completely independently from other threads.

During the stage, each thread has to scan a part of the text. Typically, each thread scans a different part of the text with only a negligible overlap between the parts. However, for highly repetitive texts with extremely large lcp values the overlaps can be large which could increase the amount of I/O significantly. To avoid this, we compute a super-sparse PLCP array PLCP_p for $p \gg q$, which can be done quickly using essentially the internal memory Φ-algorithm [9]. We then use PLCP_p to compute lower bounds according to Lemma 2, which limits the total overlaps to less than n. The sizes of the text parts for different threads may vary but this is no problem as text scanning is strongly I/O-bound. The important thing is to minimize the times when no thread is doing I/O.

The second nontrivial parallelization in EM-SΦ is Step 2(b). Here processing a single item, i.e., computing $\mathrm{lcp}(j, k)$, can involve a very long string comparison in RAM. The length of each comparison is not known in advance which makes load balancing difficult. Very long string comparisons are rare even for highly repetitive texts, but they tend to come in clusters. To see why this happens, consider a pair $(j, k) = (i + \ell, \Phi[i] + \ell)$, where $\ell = \mathrm{PLCP}_q^{\mathrm{lo}}(i)$. The lower bound ℓ ensures that the *average* length of comparisons is less than q, but there can still be rare cases where the lower bound is poor and a long comparison results. If (j, k) is such a case, then so is $(j', k') = ((i + 1) + \mathrm{PLCP}_q^{\mathrm{lo}}(i + 1), \Phi[i + 1] + \mathrm{PLCP}_q^{\mathrm{lo}}(i + 1))$ unless $i + 1$ is is a multiple of q. If furthermore $i + 1$ is a reducible position – and most positions are reducible for highly repetitive texts – $k' = k$ and $j' = j$ or $k' = k + 1$ and $j' = j + 1$, and thus (j, k) and (j', k') are processed with the same pair of half-segments. If $i + 2$, $i + 3$ and so on are reducible positions too, we can get a cluster of such bad cases. We could identify such a cluster by the fact that the difference $k - j$ is the same for all the pairs, and once identified, we can use Lemma 4 to avoid doing a long comparison more than once. However, the identification of such a cluster would require sorting the pairs (j, k) by $k - j$, and avoiding expensive sorting was one of the main ideas of the original algorithm. Consequently, our approach is to first ignore potential long comparisons and simply split a bufferful of (j, k) pairs evenly between threads. However, the average length of string comparisons is monitored, and if it exceeds a threshold, the computation is aborted and the buffer is processed in a long-lcp mode instead. In the long-lcp mode, the (j, k) pairs in the buffer are sorted by the difference $k - j$ so that we can then utilize Lemma 4. The sorting is parallelized, and the sorted buffer is split evenly among threads. Because of the sorting, the long-lcp mode is too slow to use all the time. This approach speeds up the computation significantly for highly repetitive files even in a single thread mode, and with mutiple threads it avoids bad load balancing.

4.2 Parallelizing EM-SI

The first nontrivial step in EM-SI is Step 1(c). As in Step 2(b) of EM-SΦ, some string comparisons can be long, but similar clustering of long comparisons is very unlikely because only irreducible lcp values are computed. Thus simply splitting a bufferful of $(i, \Phi[i])$ pairs evenly between threads works well enough and there is no monitoring of comparison lengths. However, the exception are comparisons that extend beyond the end of a half-segment and even the overflow buffers (which never happens in Step 2(b) in EM-SΦ). In the sequential version, such a comparison is completed immediately by reading parts of text from disk. In the parallel version, the extended comparisons are postponed until all threads are finished, and then performed separately.

The second nontrivial step in EM-SI is Step 1(d), where we want to set bits in a bitvector held in RAM (a segment of $\mathrm{PLCP}_{\mathrm{succ}}$) at positions read from disk. The problem in parallelizing this arises from two or more threads trying to simultaneously set different bits

in the same byte or word. To avoid this, the bitvector is divided into buckets and a bufferful of positions to set is first distributed into the buckets. Then each bucket is processed by a single thread. There are more buckets than threads and the buckets are assigned to threads so that each thread sets about the same number of positions.

5 In-Place Computation

When determining the disk usage of the algorithms, we assume that the inputs are on disk during the whole computation and are never modified. We are then interested in the disk space used in addition to the input, which we call the *working disk space*. The smallest possible working disk space by any algorithm is the size of the output, the LCP array. In this section, we describe modifications to the algorithms that achieve exactly this minimum working space making the algorithms in-place in this sense.

For concreteness, we assume that large integers, including lcp values, are stored using 40-bit integers by default, as they are in our current implementation, and thus the minimum working space is $40n$ bits or $5n$ bytes.

The peak working disk space usage in basic EM-SΦ happens in Step 2(b) and is $15n$ bytes in the worst case consisting of n (j, k) pairs and n lcp values produced as output of the step. In practice, the peak is closer to $10n$ bytes since each file of (j, k) pairs is deleted when it is no more needed, and can be even less because no (j, k) pair is stored when $\mathrm{lcp}(i, \Phi[i])$ can be determined from the sparse PLCP array. A working disk space of $10n$ bytes may be needed in Step 2(c) too.

In EM-SI, the worst case peak disk usage can be $15n$ bytes in Step 1(c). However, the disk usage is actually 10–$15r$ bytes, where r is the number of irreducible lcp values. For many files, $r < n/3$ and the disk usage of Step 1(c) is actually less than $5n$ bytes. In Step 2(c), the disk usage is always $10n$ bytes without any optimization.

Both algorithms involve large files that are scanned once and then deleted. In our implementation of the basic algorithms, such files are split into multiple subfiles that are deleted as soon as the scan has passed them. This reduces the working disk space of both algorithms to about $10n$ bytes in the worst case and just slightly more than $5n$ bytes is some cases. There are realistic inputs requiring about $10n$ bytes, which is still twice the minimum, and our goal is to reduce it to $5n$ bytes in all cases.

5.1 Compact Encoding of LCP Values

By default, lcp values are stored on disk using 5 bytes, but in some cases we can reduce the space using special representations. One such special representation is used for storing the sparse PLCP array PLCP_q. The default representation needs $5n/q$ bytes, but when $q < 40$ we instead use a bitvector of $n + n/q$ bits defined similarly to $\mathrm{PLCP}_{\mathrm{succ}}$.

The main technique to reduce the size of lcp values is the V-byte encoding [17], which uses a variable number of bytes to store each value. The total size of such encoding can be bounded by the following result.

▶ **Lemma 6.** *The total number of bytes in the V-byte encoding of a sequence of $\leq k$ non-negative integers summing up to $\leq s$ is at most*

$$\begin{cases} k + s/2^7 & \text{if } s/k \leq 2^7 \\ 2k + (s - 2^7 k)/2^{15} & \text{if } 2^7 \leq s/k \leq 2^{15} + 2^7 \end{cases}$$

Table 2 Working disk space (in bytes) during Steps 2(a) and (b) in EM-SΦ with partitioning.

i	n_i/n	q	pairs	lcp values	PLCP_q	total
1	0.395	4..39	$3.95n$	$n_1 + qn/2^7$	$\frac{n+n/q}{8}$	$\left(4.47 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.78n$
		40..50	$3.95n$	$n_1 + qn/2^7$	$5n/q$	$\left(4.345 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.83n$
		51..8551	$3.95n$	$2n_1 + \frac{qn-2^7n_1}{2^{15}}$	$5n/q$	$\left(4.74 + \frac{q}{2^{15}} - \frac{0.395}{2^8} + \frac{5}{q}\right) n < 5n$
2	0.330	4..39	$3.3n$	$n_1 + n_2 + qn/2^7$	$\frac{n+n/q}{8}$	$\left(4.15 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.46n$
		40..92	$3.3n$	$n_1 + n_2 + qn/2^7$	$5n/q$	$\left(4.025 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.8n$
		93..8264	$3.3n$	$2(n_1+n_2) + \frac{qn-2^7(n_1+n_2)}{2^{15}}$	$5n/q$	$\left(4.75 + \frac{q}{2^{15}} - \frac{0.725}{2^8} + \frac{5}{q}\right) n < 5n$
3	0.275	4..39	$2.75n$	$n + qn/2^7$	$\frac{n+n/q}{8}$	$\left(3.875 + \frac{q}{2^7} + \frac{1}{8q}\right) n < 4.19n$
		40..127	$2.75n$	$n + qn/2^7$	$5n/q$	$\left(3.75 + \frac{q}{2^7} + \frac{5}{q}\right) n < 4.79n$
		128..8300	$2.75n$	$2n + \frac{qn-2^7n}{2^{15}}$	$5n/q$	$\left(4.75 + \frac{q}{2^{15}} - \frac{1}{2^8} + \frac{5}{q}\right) n < 5n$

The output of Step 2(b) of EM-SΦ consists of the values $\mathrm{PLCP}[i] - \mathrm{PLCP}_q^{\mathrm{lo}}(i)$, which we call lcp delta values. By Lemma 3, the sum of all n lcp delta values is at most qn, and thus we can use Lemma 6 to bound the total size. The details are described in Section 5.2.

To take advantage of V-byte encoding in EM-SI, we make some modifications to it. First, in Step 1(c), we write $\mathrm{PLCP}[i]$ instead of $2i + \mathrm{PLCP}[i]$ to output and use V-byte encoding. Since the total sum of irreducible lcp values is at most $n \log n$, we can again bound the total size by Lemma 6. Instead of deleting the pairs $(i, \Phi[i])$ as soon as possible, we keep the i's so that we can compute $2i + \mathrm{PLCP}[i]$ in Step 1(d). To be able to delete the $\Phi[i]$'s earlier, they are stored in a different file than the i's.[1]

The second modification to EM-SI is in Step 2, where we now construct and use a sparse PLCP array PLCP_q that fits in RAM. In the output of Step 2(b) and input of Step 2(c), we replace each value $\mathrm{LCP}[i]$ with the corresponding lcp delta value $\mathrm{LCP}[i] - \mathrm{PLCP}_q^{\mathrm{lo}}(\mathrm{SA}[i])$. Then we can again use V-byte encoding and Lemma 6 to bound the total size of the lcp values.

5.2 Partitioning

The main tool for reducing disk space usage is a technique called *partitioning* introduced in [5]. Consider Step 2(a) in EM-SΦ that produces and stores up to n (j,k) pairs and then Step 2(b) processes and deletes the pairs. The pairs need up to $10n$ bytes of temporary disk space. To reduce the space, we divide the pairs into three parts of sizes $n_1 = 0.395n$, $n_2 = 0.33n$ and $n_3 = 0.275n$, and perform the Steps 2(a) and (b) for one part at a time. Then the peak disk usage stays under $5n$ bytes at all times as detailed in Table 2. There is an upper limit of 8264 on the value of q but that is sufficient to fit PLCP_q into RAM in all practical scenarios. Furthermore, a larger q or smaller disk usage can be achieved by using more than three parts.

We also use partitioning in Step 1 of EM-SI. We perform the full step 1 except the setting of reducible bits for one part at a time. That is, after processing one part, we will have, for each irreducible i processed in that part, the bit $2i + \mathrm{PLCP}[i]$ set in $\mathrm{PLCP}_{\mathrm{succ}}$ and the bit i set in R. Once all parts have been processed, we produce the final $\mathrm{PLCP}_{\mathrm{succ}}$ by setting the reducible bits. With three parts of size at most $n/3$ each, the working disk space stays well under $5n$ bytes.

[1] The separation of i's and $\Phi[i]$'s into separate files helps with Step 1(b) too, because we need only i's to determine the irreducible positions.

The disadvantage of partitioning is that it needs some additional I/O depending on exactly how the partitioning is done. We have implemented two partitioning modes for each algorithm and always choose the one that produces less additional I/O. The first mode is called lex-partitioning and simply involves splitting SA into parts. When processing a part, we only need to scan the relevant part of SA. However, then each pair of text half-segments needs to be loaded into RAM once for each part. For large files, the loading of the half-segments dominates the I/O, and thus we instead use the second partitioning mode called text-partitioning. The items are partitioned according to which pair of half-segments they belong to. Then most pairs of half-segments need to be loaded only once. On the other hand, we then need to scan the full suffix array for each part, which makes it the slower option for smaller files.

Finally, partitioning also happens in Steps 2(a)–(b) in EM-SI. Recall that earlier we modified Step 2(b) to produce lcp delta values as output. In this case, we always perform lex-partitioning into two parts of sizes $n_1 = 0.58$ and $n_2 = 0.42$. Then the working disk space remains below $5n$ bytes when $q < 20000$.

5.3 Final Steps

The final step of EM-SΦ, Step 2(c), performs two tasks: it reads the lcp values from multiple files and merges them into a single sequence, and it converts the lcp delta values into the final lcp values. The in-place version separates the tasks, first merging in Step 2(c'), and then converting in Step 2(c"). Since the total size of the V-byte encoded delta values is always less than $2.5n$ bytes, the working disk space during merging stays below $5n$ bytes. For conversion, the merged delta value file is split into multiple subfiles so that each subfile can be deleted after it has been processed. The split points are decided adaptively during merging so that the working disk space never exceeds $5n$ bytes. The first subfile can always contain more than half the values. The following subfiles are smaller, and the last subfile is small enough so that we can load it in RAM and delete it from disk before conversion.

The final step of EM-SI, Step 2(c), after the modification to use V-byte encoded delta values, is essentially the same as the last step of EM-SΦ: merge and convert lcp delta values into the final LCP array, and is implemented similarly.

6 Experimental Results

Algorithms. The starting point and the baseline in our experiments are the original C++ implementations of the EM-SΦ and EM-SI algorithms described in [4]. Both algorithms are sequential, assume byte alphabet, and use more disk space than is needed for the output. We modified these implementations in the following ways:

- First, we parallelized the computation as described in Section 4. All basic parallelizations were done using OpenMP, and for more non-trivial parallelizations we use threads and synchronization mechanisms from the standard C++ library;
- Second, we added the "in-place mode" to both algorithms that reduces the working disk space to the space needed by the final LCP array as described in Section 5. We kept the "out-of-place mode" in the implementation for cases, where speed is the priority;
- Third, we modified the implementations to handle symbols of arbitrary size (that is a multiple of byte). This is relatively straightforward, as both algorithms only perform symbol comparisons, but our implementations are the first to explicitly support large-alphabet external-memory construction of the LCP array.

Table 3 Statistics of data used in the experiments; $100r/n$ is the percentage of irreducible lcp values among all lcp values (where r denotes the number of irreducible lcps) and Σ_r/r is the average length of the irreducible lcp value (where Σ_r is the sum of all irreducible lcps). Smaller files in experiments are prefixes of full test files. The input symbols are encoded using bytes for all files except **words**, for which we use 32 bits per symbol.

Input	$n/2^{30}$	$\lvert\Sigma\rvert$	$100r/n$	Σ_r/r
kernel	128.0	229	0.09	1494.76
geo	128.1	211	0.15	1221.49
wiki	128.7	213	16.71	29.40
dna	128.0	6	18.46	23.79
debruijn	128.0	2	99.26	35.01
words	12.5	97 002 175	42.49	5.17

Each of the two algorithms has thus four variants: depending on whether it uses parallelism, and whether it runs in the in-place mode. The implementations are capable of using arbitrary types to represent integers and text symbols, but for simplicity in all experiments in this section we use 40-bit integers. The implementations used in experiments (as well as datasets described next) are available at `http://www.cs.helsinki.fi/group/pads/`.

Setup. We performed experiments on a machine equipped with two six-core 1.9 GHz Intel Xeon E5-2420 CPUs (capable, via hyper-threading, of running 24 threads) with 15 MiB L3 cache and 120 GiB of DDR3 RAM. For experiments we limited the RAM in the system to 4 GiB (with the kernel boot flag) and all algorithms were allowed to use 3.5 GiB in all experiments. The machine had 6.8 TiB of free disk space striped with RAID0 across four identical local disks achieving a (combined) transfer rate of about 480 MiB/s (read/write).

The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.13.0. All programs were compiled using `g++` version 5.2.1 with `-O3 -march=native` options. All reported runtimes are wallclock (real) times. The machine had no other significant CPU tasks running and for all sequential algorithms only a single thread of execution was used for computation (we permit a constant number of extra threads as long as they do not perform computation, e.g., threads responsible for scheduling I/O requests are allowed). The parallel algorithms used the full parallelism available on the machine (24 threads), unless explicitly stated otherwise.

Datasets. For the experiments we used the following files (see Table 3 for some statistics):

- **kernel**: a concatenation of ∼10.7 million source files from over 300 versions of Linux kernel (see `http://www.kernel.org/`). This is an example of highly repetitive file;
- **geo**: a concatenation of all versions (edit history) of Wikipedia articles about all countries and 10 largest cities in the XML format. The resulting file is also highly repetitive;
- **wiki**: a concatenation of wikipedia, w-source, w-books, w-news, w-quote, w-versity, and w-voyage dumps dated 20160203 in XML (see `http://dumps.wikimedia.org/`);
- **dna**: a collection of DNA reads from multiple human genomes filtered from symbols other than $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$ and newline (see `http://www.1000genomes.org/`);
- **debruijn**: a binary De Bruijn sequence of order k is an artificial sequence of length $2^k + k - 1$ than contains all possible binary k-length substrings. It contains nearly n irreducible lcps (see [9, Lemma 5]) which is the worst case for EM-SI;
- **words**: a collection of natural language text parsed into words and converted into 4-byte integers, see `http://www.statmt.org/wmt16/translation-task.html`.

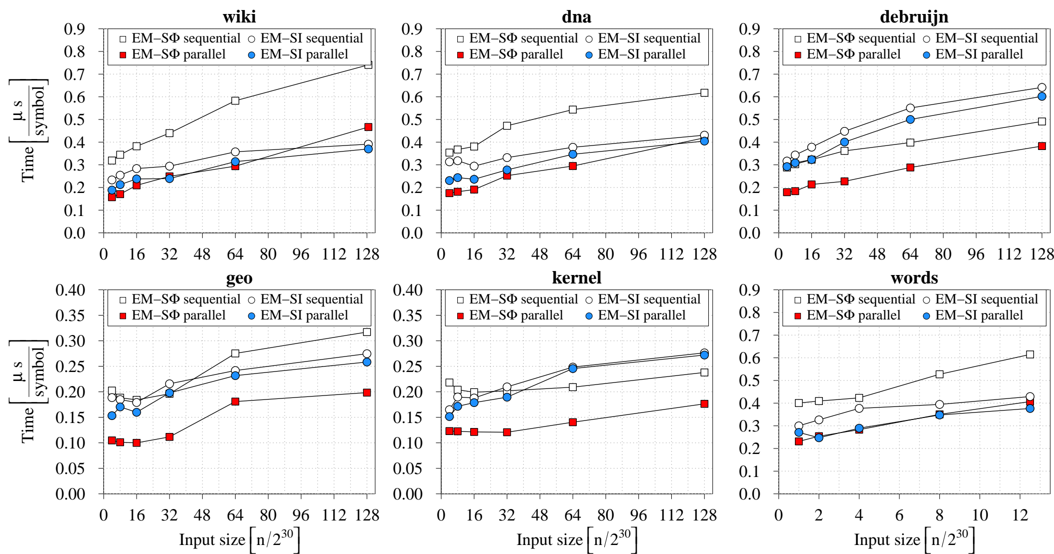

Figure 1 Scalability of the parallel algorithms compared to their sequential versions.

Parallel Algorithms. In the first experiment, we compare the parallel versions of the algorithms studied in this paper with their sequential versions. We executed the algorithms on increasing length prefixes of all testfiles and measured the runtime. For now we use all algorithms in the basic out-of-place mode. The results are given in Figure 1.

The parallel version of EM-SΦ achieves a significant speedup across all prefix sizes and input types. This is caused by the fact, that the algorithm performs a lot of random accesses to Φ_q in Step 1 and PLCP_q in Steps 2(a) and 2(c), making them strongly compute-bound. The parallelization helps even in Step 2(b), since most of the lcp comparisons are short which prevents OS from effective cache prefetching. The average speedup across all input types varies from 46% for 4 GiB inputs to 32% for full testfiles.

The speedup for EM-SI is notably smaller, since its sequential version is already largely I/O-bound. While for non-repetitive inputs the algorithm achieves a speedup of about 10%, for highly repetitive data (kernel, geo), the speedup is negligible, particularly for large text.

In the second experiment, we focus on the parallel version of EM-SΦ, as it benefits more from parallelism than EM-SI. We executed the algorithm on the largest instances of two testfiles (**wiki** and **geo** serving as examples of non-repetitive and highly-repetitive input) using different number of threads (we point out here, that the sequential version and parallel version running on a single thread are not the same implementation as the purely sequential version can avoid certain computations), and measured the runtime. As seen in Figure 2, the maximum speedup is already achieved with about 8 threads. At this point the algorithm becomes essentially I/O-bound.

In-Place Algorithms. In the next experiment we study the in-place variants of the algorithms described in Section 5. The in-place mode increases the I/O in both algorithms mostly due to additional scans of SA. The change in I/O volume is, however, not significant, hence for brevity we do not report I/O volume in this section. To study the effect on runtime, we executed the algorithms both in the in-place and the out-of-place mode on different prefixes of testfiles, and measured the runtime and disk space usage (to measure disk usage we used our own script but as a sanity check we ran a preliminary set of experiments in the in-place

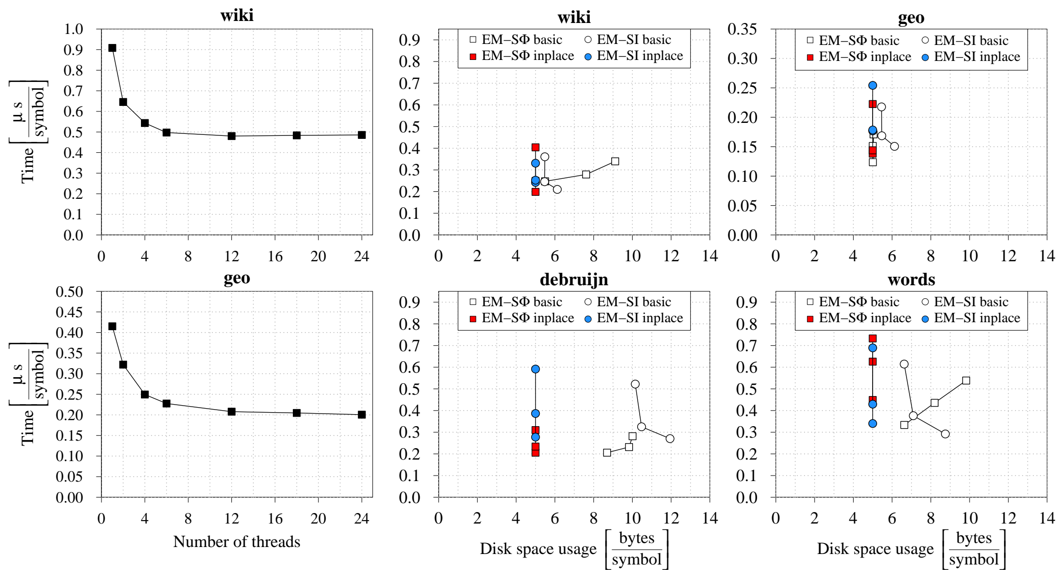

Figure 2 Left: Normalized runtime of the parallel version of EM-SΦ on the prefixes of length $n = 128 \times 2^{30}$. Right: Normalized runtime vs. disk space usage of the parallel algorithms in the in-place mode compared to the basic (out-of-place) mode. The three dots of each color/shape correspond to text prefixes of sizes 4, 16, and 64 GiB, which for the **words** file means $n \in \{2^{30}, 2^{32}, 2^{34}\}$.

Table 4 Comparison of two approaches to LCP array construction for large-alphabet inputs.

Algorithm	Time $\left[\frac{\mu s}{\text{symbol}}\right]$	Disk space $\left[\frac{\text{bytes}}{\text{symbol}}\right]$	I/O volume $\left[\frac{\text{bytes}}{\text{symbol}}\right]$
EM-SΦ native	0.41	8.43	137.25
EM-SΦ byte-based	1.00	34.07	254.12
EM-SI native	0.38	5.53	116.78
EM-SI byte-based	1.08	21.94	259.56

mode using a setup, where the available disk space is only negligibly larger than the output LCP array). For simplicity, we present the results for parallel versions but they were very similar on sequential versions. The results are given in Figure 2. We point out that due to the simultanous disk space measurement, these runs are slightly slower and thus not comparable to Figure 1, but the relative runtimes remain the same.

The slowdown of the in-place mode compared to basic versions is very moderate. The maximum slowdown for EM-SΦ was about 36% (but in most cases much smaller), and the maximum slowdown for EM-SI was about 17%. The working disk space usage in some cases, particularly for non-repetitive inputs, is reduced by more than a factor of two.

Large Alphabet. Suppose that the input string consists of symbols drawn from a large alphabet such that each symbol requires more than one byte. To compute the LCP array for such strings we can take one of two approaches. First, we can use an algorithm that natively supports large alphabet, and we have modified our implementations to provide such support. A recently published implementation of EM scalable and space-efficient suffix array construction provides a large-alphabet support, complementing this approach [8].

An alternative method is to first split each symbol into a group of symbols over byte alphabet. One can then apply a byte-based suffix sorter to compute the suffix array, then run a byte-based LCP array construction, and then compute and select the final subset of LCP

values in the postprocessing stage (which requires one scan of SA and LCP). A drawback of this approach is that reducing the alphabet increases the length of the string. For example, if the symbols of the original string of length n were encoded using 32-bit integers and we wish to obtain a string over byte alphabet, the resulting string has length $4n$.

To compare the two approaches in practice we used the parallel versions of the two algorithms studied in this paper in the basic (out-of-place) mode. We executed each algorithm on the prefix of the **words** testfile of length $n = 12.5 \times 2^{30}$ (with each symbol encoded using four bytes), first using the native large-alphabet mode, and then assuming the input is over byte alphabet, and compared the resources needed by the two approaches. We exclude the resources needed to compute the suffix array of the byte-interpreted input (needed in the second approach), as well as the postprocessing of the LCP array. The results, scaled with respect to the large-alphabet string, are reported in Table 4. Using the algorithm that natively supports large alphabet is about 2.5 × faster, uses half of the I/O volume, and requires about 4 × less working disk space. The overall disk usage (i.e., including input) of the native mode is even smaller, because the SA of the large-alphabet text is four times smaller than the SA of the byte-intepreted text.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.
2. T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 88–102. SIAM, 2013. doi:10.1137/1.9781611972931.8.
3. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.
4. J. Kärkkäinen and D. Kempa. Faster external memory LCP array construction. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *LIPIcs*, pages 61:1–61:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.61.
5. J. Kärkkäinen and D. Kempa. LCP array construction in external memory. *J. Exp. Algorithmics*, 21(1):1.7:1–1.7:22, April 2016. doi:10.1145/2851491.
6. J. Kärkkäinen and D. Kempa. LCP array construction using O(sort(n)) (or less) I/Os. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 204–217. Springer, 2016. doi:10.1007/978-3-319-46049-9_20.
7. J. Kärkkäinen, D. Kempa, and M. Piątkowski. Tighter bounds for the sum of irreducible LCP values. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 316–328. Springer, 2015. doi:10.1007/978-3-319-19929-0_27.
8. J. Kärkkäinen, D. Kempa, S. J. Puglisi, and B. Zhukova. Engineering external memory induced suffix sorting. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX 2017)*, pages 98–108. SIAM, 2017. doi:10.1137/1.9781611974768.8.
9. J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.

10 T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. doi:10.1007/3-540-48194-X_17.

11 V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

12 U. Manber and G. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

13 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):article 2, 2007. doi:10.1145/1216370.1216372.

14 E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

15 K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232. ACM/SIAM, 2002.

16 G. Tischler. Low space external memory construction of the succinct permuted longest common prefix array. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 178–190. Springer, 2016. doi:10.1007/978-3-319-46049-9_18.

17 H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999. doi:10.1093/comjnl/42.3.193.