Faster External Memory LCP Array Construction

Juha Kärkkäinen[1] and Dominik Kempa[2]

1 **Helsinki Institute for Information Technology HIIT & Department of Computer Science, University of Helsinki, Helsinki, Finland**
juha.karkkainen@cs.helsinki.fi
2 **Helsinki Institute for Information Technology HIIT & Department of Computer Science, University of Helsinki, Helsinki, Finland**
dominik.kempa@cs.helsinki.fi

Abstract

The suffix array, perhaps the most important data structure in modern string processing, needs to be augmented with the longest-common-prefix (LCP) array in many applications. Their construction is often a major bottleneck especially when the data is too big for internal memory. We describe two new algorithms for computing the LCP array from the suffix array in external memory. Experiments demonstrate that the new algorithms are about a factor of two faster than the fastest previous algorithm.

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.ESA.2016.61

1 Introduction

The suffix array [22, 10], a lexicographically sorted list of the suffixes of a text, is the most important data structure 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 [25]. Modern textbooks spend dozens of pages in describing their applications, see e.g. [28, 21].

The construction of the suffix and LCP arrays has been heavily studied over the years. Recently, several algorithms and implementations for constructing the suffix array in external memory have been published [7, 4, 6, 11, 27, 26, 19, 15]. Such algorithms are frequently needed for handling large texts or text collections that are too big to process in RAM. Some of the algorithms can also compute the LCP array simultaneously with the suffix array [4, 6] and others could probably be modified to do so. However, such a modification is unique to each algorithm and can significantly increase the construction time as well as the disk space usage of the algorithm [4].

A better solution for constructing the LCP array is to construct the suffix array separately first and then compute the LCP array from the suffix array. This has been a standard practice in internal memory for 15 years [18] but became possible in external memory only recently with the introduction of the LCPscan algorithm [12, 13]. This led to a significant improvement in construction time as well as in disk space usage over previous approaches.

Furthermore, since LCPscan can be combined with any suffix array construction algorithm, it can immediately benefit from any progress in the fast developing field of suffix array construction. For example, the recent pSAscan algorithm [15] can often construct the suffix

Table 1 The time and I/O complexities of LCPscan and the new algorithms in the standard external memory model [34]. The parameters are the text length n, the alphabet size σ, the available RAM M (in units of $\log n$ bits), and the disk block size B (in units of $\log n$ bits).

Algorithm	Time complexity	I/O complexity
LCPscan	$\mathcal{O}\left(\frac{n^2}{M(\log_\sigma n)^2} + n \log_{\frac{M}{B}} \frac{n}{B}\right)$	$\mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$
Sparse-Φ	$\mathcal{O}\left(\frac{n^2}{M} + n \log_{\frac{M}{B}} \frac{n}{B}\right)$	$\mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$
SuccinctIrreducible	$\mathcal{O}\left(\frac{n^2}{M(\log_\sigma n)^2} + n \log n\right)$	$\mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n \log \sigma}{B} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$

array significantly faster than the LCPscan algorithm can construct the LCP array [13, Table IX]. Thus there is a need for even faster LCP array construction in external memory.

Our contribution. In this paper, we describe two new external memory algorithms for constructing the LCP array from the suffix array. Although the new algorithms share some features with LCPscan their more immediate ancestors are two semi-external algorithms introduced in [16]. The semi-external algorithms need to keep the text and some additional small data structures in RAM but the larger suffix and LCP arrays are kept on disk and are accessed sequentially only. When there is enough RAM, these algorithms are many times faster than LCPscan.

We show how the requirement to keep the text (and some additional data structures) can be removed from the algorithms. Although this adds a significant amount of computation and I/O, the resulting algorithms are still about a factor of two faster than LCPscan in our experiments. Asymptotically, LCPscan has a slight advantage over the new algorithms (see Table 1). The main disadvantage of LCPscan is that it relies heavily on external memory sorting, which is completely avoided by the new algorithms.

The advantage of the new algorithms over LCPscan is particularly large when the text is only slightly larger than the available RAM. This is a common situation when dealing with compressed full-text self-indexes [25]. A compressed index can be significantly smaller than an uncompressed text and fit in RAM even though the text does not. However, the construction of the index still requires external memory computation.

Related work. Kasai et al. [18] introduced the first (internal memory) algorithm for computing the LCP array from the suffix array. It is simple and fairly fast but requires a lot of space. Thus a lot of the later work focused on reducing the space [20, 23, 30, 16, 33, 9, 3]. A culmination of this line of work are semi-external algorithms that keep most of the data structures on disk but need to have at least the text in RAM [30, 16]. There is also recent research on speeding up LCP computation by using parallelism [8, 32].

External memory algorithms for constructing the suffix array have been around since the early days [10], but until recently the only way to construct the LCP array when the text does not fit in RAM was as byproduct of a suffix array construction algorithm [17, 4, 6, 2]. To the best of our knowledge, LCPscan [12] is still the only external memory algorithm that can construct the LCP array from the suffix array independently of how it was constructed.

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

Table 2 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	-
$\Phi[i] + \mathrm{PLCP}[i]$	12	12	12	12	12	12	3	3	7	4	4	4	-

$[i..j-1]$. 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* [22, 10] SA of X is an array $\mathrm{SA}[0..n]$ 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 $(j+1)^{\text{th}}$ suffix of X in ascending lexicographical order. Another representation of the permutation is the Φ *array* [16] $\Phi[0..n)$ 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 2.

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 2, $\mathrm{lcp}(0,6) = 3 = |\texttt{bab}|$ and $\mathrm{lcp}(7,4) = 5 = |\texttt{abbab}|$. The *longest-common-prefix array* [22, 18], $\mathrm{LCP}[1..n]$, 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* [16] $\mathrm{PLCP}[0..n)$ is the LCP array 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 2 shows example LCP and PLCP arrays. The last two rows in Table 2 illustrate the following property of the PLCP array, which is the basis of all efficient algorithms for LCP array construction.

▶ **Lemma 1** ([13]). *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* [31] $\mathrm{PLCP}_{\mathrm{succ}}[0..2n)$ 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 $2i + \mathrm{PLCP}[i]$ must be unique for each i by Lemma 1. Any lcp value can be recovered by the equation $\mathrm{PLCP}[i] = \mathrm{select}(\mathrm{PLCP}_{\mathrm{succ}}, i) - 2i$, where $\mathrm{select}(\mathrm{PLCP}_{\mathrm{succ}}, i)$ returns the location of the $(i+1)^{\text{th}}$ 1-bit in $\mathrm{PLCP}_{\mathrm{succ}}$. The select query can be answered in $\mathcal{O}(1)$ time given a precomputed data structure of $o(n)$ bits [5, 24].

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 qth entry of PLCP. It 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** ([16]). *For any $i \in [0..n)$, let $a = \lfloor i/q \rfloor$ and $b = i \bmod q$, so that $i = aq + b$. If $(a+1)q \le n-1$, then $\mathrm{PLCP}_q[a] - b \le \mathrm{PLCP}[i] \le \mathrm{PLCP}_q[a+1] + q - b$. If $(a+1)q > n-1$, then $\mathrm{PLCP}_q[a] - b \le \mathrm{PLCP}[i] \le n - i \le q$.*

Let the slack, denoted by $\mathrm{slack}_q(i)$, be the difference of the upper and lower bounds for $\mathrm{PLCP}[i]$ given by Lemma 2. Although there is no non-trivial bound on an individual slack, the sum of the slacks is bounded by the following lemma.

▶ **Lemma 3** ([16]). $\sum_{i\in[0..n)} \text{slack}_q(i) \leq (q-1)n + q^2$.

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

▶ **Lemma 4** ([16]). *If* PLCP[i] *is reducible, then* PLCP[i] = PLCP[$i-1$] − 1 *and* $\Phi[i] = \Phi[i-1]+1$.

▶ **Lemma 5** ([16, 14]). *The sum of all irreducible lcp values is* $\leq n \log n$.

3 Basic Semi-External Algorithms

We will next describe the two semi-external algorithms introduced by Kärkkäinen, Manzini and Puglisi [16]. The semi-external versions are only briefly mentioned in [16] but they are essentially the same as the space-efficient versions. Both algorithms need enough RAM for the text and for either the succinct or the sparse PLCP array. The larger data structures SA and LCP are stored on disk in the semi-external algorithms.

The first algorithm, which we call Sparse-Φ, performs the following main steps:

1. Compute a sparse version Φ_q of the Φ-array defined so that $\text{PLCP}_q[i] = \text{lcp}(qi, \Phi_q[i])$.
2. Compute PLCP_q using Φ_q. When computing $\text{PLCP}_q[i]$, we take advantage of the fact that $\text{PLCP}_q[i] \geq \text{PLCP}_q[i-1] - q$ (Lemma 1).
3. Compute LCP using PLCP_q based on Lemma 2.

The first two steps need $\mathcal{O}(n)$ time and the third step $\mathcal{O}(qn)$ time. The pseudocode for Sparse-Φ is given in Figure 1. All accesses to SA and LCP are sequential allowing them to be stored on disk.

The second algorithm is called SuccinctIrreducible and has the following steps:

1. Compute irreducible lcp values and store them in the succinct PLCP array $\text{PLCP}_{\text{succ}}$.
2. Compute the reducible lcp values using Lemma 4 and store them in $\text{PLCP}_{\text{succ}}$.
3. Compute LCP from $\text{PLCP}_{\text{succ}}$.

During steps 1 and 2 we also need a bitvector R[0..n] for marking the irreducible positions in PLCP. The first step needs $\mathcal{O}(n \log n)$ time by Lemma 5, and the other steps need $\mathcal{O}(n)$ time. Again, all accesses to SA and LCP are sequential.

The algorithm also uses BWT (line 4). Since BWT[i] = X[SA[i]-1] (unless SA[i] = 0), we can compute the values on-the-fly when the text X is in RAM as was done in [16]. However, we want to get rid of the requirement that the text is in RAM and instead assume that the BWT is available on disk and is accessed sequentially during the algorithm.

4 Moving Text into External Memory

The semi-external algorithms described above need to have the text X in RAM. While a single comparison of two suffixes is sequential, the first character accesses in each comparison are random accesses often enough so that any straightforward way to deal with texts larger than RAM will not work. Instead, the computation in the steps involving text accesses have to be completely reorganized as will be described in this section. Here we still assume that Φ_q, PLCP_q, $\text{PLCP}_{\text{succ}}$ and R fit in RAM; handling them in a fully external memory algorithm is covered in the next section.

SPARSE-Φ

— *Step 1: Compute* Φ_q
1: **for** $i \leftarrow 1$ **to** n **do**
2: **if** $\mathrm{SA}[i] \bmod q = 0$ **then**
3: $\Phi_q[\mathrm{SA}[i]/q] \leftarrow \mathrm{SA}[i-1]$
— *Step 2: Compute* PLCP_q *using* Φ_q
4: $\ell \leftarrow 0$
5: **for** $i \leftarrow 0$ **to** $\lceil n/q \rceil - 1$ **do**
6: $j \leftarrow \Phi_q[i]$
7: **while** $\mathsf{X}[qi+\ell] = \mathsf{X}[j+\ell]$ **do**
8: $\ell \leftarrow \ell+1$
9: $\mathrm{PLCP}_q[i] \leftarrow \ell$
10: $\ell \leftarrow \max(\ell - q, 0)$
— *Step 3: Compute* LCP *using* PLCP_q
11: **for** $i \leftarrow 1$ **to** n **do**
12: $j \leftarrow \mathrm{SA}[i-1]$
13: $k \leftarrow \mathrm{SA}[i]$
14: $k' \leftarrow \lfloor k/q \rfloor$
15: $\ell \leftarrow \mathrm{PLCP}_q[k'] - (k - k'q)$
16: $\ell \leftarrow \max(\ell, 0)$
17: **while** $\mathsf{X}[k+\ell] = \mathsf{X}[j+\ell]$ **do**
18: $\ell \leftarrow \ell+1$
19: $\mathrm{LCP}[i] \leftarrow \ell$

SUCCINCTIRREDUCIBLE

— *Step 1: Compute irreducible lcps*
1: $\mathrm{PLCP}_{\mathrm{succ}}[0..2n) \leftarrow (0, 0, \ldots, 0)$
2: $\mathsf{R}[0..n] \leftarrow (0, 0, \ldots, 0, 1)$
3: **for** $i \leftarrow 1$ **to** n **do**
4: **if** $\mathrm{BWT}[i] \neq \mathrm{BWT}[i-1]$ **then**
5: $j \leftarrow \mathrm{SA}[i-1]$; $k \leftarrow \mathrm{SA}[i]$
6: $\mathsf{R}[k] \leftarrow 1$
7: $\ell \leftarrow 0$
8: **while** $\mathsf{X}[k+\ell] = \mathsf{X}[j+\ell]$ **do**
9: $\ell \leftarrow \ell+1$
10: $\mathrm{PLCP}_{\mathrm{succ}}[2k+\ell] \leftarrow 1$
— *Step 2: Fill in reducible lcps*
11: $i \leftarrow 0$; $j \leftarrow 0$
12: **while** $i < n$ **do**
13: **while** $\mathrm{PLCP}_{\mathrm{succ}}[j] = 0$ **do** $j \leftarrow j+1$
— *Now* $j = \mathrm{PLCP}[i] + 2i$
14: $i \leftarrow i+1$; $j \leftarrow j+1$
15: **while** $\mathsf{R}[i] = 0$ **do**
16: $\mathrm{PLCP}_{\mathrm{succ}}[j] \leftarrow 1$
17: $i \leftarrow i+1$; $j \leftarrow j+1$
— *Step 3: Compute* LCP *from* $\mathrm{PLCP}_{\mathrm{succ}}$
18: Construct select-structure for $\mathrm{PLCP}_{\mathrm{succ}}$
19: **for** $j \leftarrow 1$ **to** n **do**
20: $i \leftarrow \mathrm{SA}[j]$
21: $\mathrm{LCP}[j] \leftarrow \mathrm{select}(\mathrm{PLCP}_{\mathrm{succ}}, i) - 2i$

Figure 1 Two semi-external algorithms.

We will analyze the algorithms in the standard external memory model [34], where the memory system consists of a fast random access memory (RAM) of size M and a slow (disk) memory of unbounded size divided into blocks of size B, both measured in units of $\mathcal{O}(\log n)$ bits. We are primarily interested in the I/O complexity which measures the number of blocks read from or written to disk. Notice that we can fit $\mathcal{O}(M \log_\sigma n)$ characters in RAM and $\mathcal{O}(B \log_\sigma n)$ characters in a disk block.

All text accesses in both algorithms happen in loops where the goal is to compute $\mathrm{lcp}(i, \Phi[i])$ for some i. The basic idea is to divide the text into segments of size at most $m = \mathcal{O}(M \log_\sigma n)$ such that two segments fit in RAM. For each pair of segments at a time, we load them into RAM and compute $\mathrm{lcp}(i, \Phi[i])$ for each i such that i and $\Phi[i]$ are in those two segments. Further details we will consider separately for each step involving text accesses.

Step 1 in SuccinctIrreducible. The computation on lines 1–10 in SUCCINCTIRREDUCIBLE including the text access loop on line 8 is replaced by the following steps:

1.1. Scan SA and BWT to form a pair $(i, \Phi[i])$ for each i such that $\mathrm{PLCP}[i]$ is irreducible. The pairs are written to disk where there is a separate file for each pair of text segments. Simultaneously, compute the bitvector R, which is kept in RAM during the step and written to disk at the end of the step.

1.2. For each pair of text segments, load them to RAM and compute $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \Phi[i])$ for each pair $(i, \Phi[i])$ obtained from the associated file. For each computed $\mathrm{PLCP}[i]$, we store the value $2i + \mathrm{PLCP}[i]$ to disk.

1.3. With $\mathrm{PLCP}_{\mathrm{succ}}$ in RAM, read the output of the previous step and set the corresponding bits of $\mathrm{PLCP}_{\mathrm{succ}}$ to 1. Then read R from disk.

The rest of the algorithm is as in Section 3. The total number of additional I/Os resulting from this procedure is $\mathcal{O}((n/\log_\sigma n)^2/(MB))$.

An additional detail to consider in step 1.2 is that although i and $\Phi[i]$ are in the segments in RAM, the common prefix of those suffixes may continue beyond the end of the segments. To deal with this, we also keep the first $B\log_\sigma n$ symbols after the segments in RAM. These are called *overflow buffers*. If the comparison continues beyond the overflow buffers, we read the relevant parts of the text from disk sequentially. Since the total length of the irreducible lcps is $\mathcal{O}(n\log n)$, the additional number of I/Os from this is never more than $\mathcal{O}((n\log n)/(B\log_\sigma n)) = \mathcal{O}((n\log\sigma)/B)$.

Step 3 in Sparse-Φ. The loop on lines 11-19 in SPARSE-Φ including the text access loop on line 17 is replaced with the following steps:

3.1. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair use PLCP_q (stored in RAM) to compute the lower bound $\ell_{\min}$ (and the upper bound $\ell_{\max}$) for $\text{PLCP}[i]$. Write the pair $(i+\ell_{\min}, \Phi[i]+\ell_{\min})$ to disk, where there is a separate file for each pair of text segments.

3.2. For each pair of text segments, load them to RAM and compute $\text{lcp}(i,j)$ for each pair (i,j) obtained from the associated file. The resulting value $\text{lcp}(i,j)$ is written to disk to a separate file for each pair of text segments. The order of the lcp values in the output file must be the same as the order of the pairs in the input file.

3.3. Scan SA to generate all $(i, \Phi[i])$ pairs. For each pair, compute $\ell_{\min}$ as in step 1 and read the value $\ell' = \text{lcp}(i+\ell_{\min}, \Phi[i]+\ell_{\min})$ from the appropriate file. Then $\text{PLCP}[i] = \ell_{\min}+\ell'$ is the next value in the LCP array.

The total number of additional I/Os from reading text segments is $\mathcal{O}((n/\log_\sigma n)^2/(MB))$.

Again, we have to deal with lcp comparisons continuing beyond the end of the segments in step 3.2. In step 3.1, we use the upper bound $\ell_{\max}$ to determine whether such an overflow is possible, and if it is, we generate additional pairs/triples for each possible boundary crossing. For example, if for some $\ell' \in [\ell_{\min}..\ell_{\max}]$ we find out that $i+\ell'$ is at a segment boundary, we generate the triple $(i+\ell', \Phi[i]+\ell', \ell_{\max}-\ell')$. The third value is used as an upper bound for the length of the comparison, which might be needed in the case where $\text{lcp}(i,\Phi[i]) < \ell'$. Otherwise, the triple is treated as a normal pair in step 3.2. All the comparisons in step 3.2 end at a segment boundary. In step 3.3, we generate pairs and triples as in step 3.1, read the corresponding lcp values from the appropriate files, and combine them to obtain the final lcp value. The total number of the extra triples is at most $\mathcal{O}(n + qn/(M\log_\sigma n))$. Also notice that the total number of character comparisons is still bounded by $\mathcal{O}(qn)$.

Step 2 in Sparse-Φ. The third and final place with a text access loop is on line 7 in algorithm SPARSE-Φ. In this case, the text segment size is $2m$ and only one segment is kept in RAM while the rest of the text is scanned sequentially. The loop on lines 5–10 is replaced with the following steps:

2.1. Scan Φ_q to generate all pairs $(i, \Phi[i])$ such that i is a multiple of q. Write the pairs to disk into the file associated with the text segment that contains $\Phi[i]$. Notice that the pairs in each file are sorted by i.

2.2. For each segment, load the segment into RAM. Read the pairs $(i, \Phi[i])$ from the associated file while simultaneously scanning the text so that the position $\mathsf{X}[i]$ is reached when the pair $(i, \Phi[i])$ is processed. For each pair, compute $\text{lcp}(i, \Phi[i])$ and write it to disk into a separate file for each segment. When computing $\ell = \text{lcp}(i, \Phi[i])$ we use the fact that $\ell \geq \text{lcp}(i', \Phi[i']) - (i - i')$, where $(i', \Phi[i'])$ is the pair processed just previously. This ensures that the text scan never needs to backtrack.

2.3. Scan Φ_q and for each $i \in [0..\lceil n/q \rceil)$ read $\mathrm{PLCP}_q[i]$ from the file associated with the text segment containing $\Phi_q[i]$.

The total number of additional I/Os from scanning the text is $\mathcal{O}((n/\log_\sigma n)^2/(MB))$.

Here too, we have the possibility that the position $\Phi[i]+\ell$ moves beyond the end of the segment during the comparison. We deal with this again by having an overflow buffer of $\mathcal{O}(B \log_\sigma n)$ characters and by reading text from disk when the overflow buffer is not enough. Then the extra I/Os for reading the positions $\Phi[i]+\ell$ is never more than the regular I/Os for reading the positions $i+\ell$.

As a final note, both algorithms distribute pairs into $\mathcal{O}(s^2)$ files at some point, where $s = \mathcal{O}(n/(M \log_\sigma n))$ is the number of segments. To do this efficiently we need to have enough RAM for $\mathcal{O}(s^2)$ buffers of size $\mathcal{O}(B)$ each, which means that we must have $s = \mathcal{O}(\sqrt{M/B})$ and thus $n = \mathcal{O}\left(M\sqrt{M/B}\log_\sigma n\right)$. We can get rid of this constraint by doing the distribution in multiple rounds. That is, we first distribute the pairs into $\mathcal{O}(M/B)$ files and then those files are divided into smaller files and so on. The I/O complexity of the multiround distribution is the same as for external memory sorting, $\mathcal{O}((n/B)\log_{M/B}(n/B))$, and the time complexity is $\mathcal{O}(n \log_{M/B}(n/B))$.

5 Fully External Memory Algorithms

Let us now complete the transformation of the semi-external algorithms into external memory algorithms by describing how to deal with Φ_q, PLCP_q, $\mathrm{PLCP}_{\mathrm{succ}}$ and R.

Consider first Sparse-Φ. We set $q = \min\{n/M, M\log_\sigma n\}$. This choice ensures that the number of extra triples generated in Step 3 is $\mathcal{O}(n + qn/(M\log_\sigma n)) = \mathcal{O}(n)$. When $n \le M^2 \log_\sigma n$, we have $q = \Theta(n/M)$ so that Φ_q and PLCP_q fit in RAM and the algorithm is exactly as described above. When $n > M^2\log_\sigma n$, we divide Φ_q and PLCP_q into $s = \mathcal{O}(n/(M^2\log_\sigma n))$ segments that fit in RAM. In Steps 1 and 3, instead of scanning SA once, we scan it s times, once for each segment. Step 3 also produces s subsequences of LCP which are then merged with the help of one more scan of SA. The additional I/O from the extra scans is $\mathcal{O}(n^2/(M^2 B\log_\sigma n))$, which is less than the I/O for text scanning (assuming $M = \Omega(\log_\sigma n)$).

The time complexity has three main components: $\mathcal{O}(qn)$ time for comparing suffixes, $\mathcal{O}(n^2/(M(\log_\sigma n)^2))$ for loading text segments and scanning the text, and $\mathcal{O}(n\log_{M/B}(n/B))$ time for multiround distribution. This gives the following result.

▶ **Theorem 6.** *Given a text of length n over an integer alphabet $[0..\sigma)$ and its suffix array, the associated* LCP *array is computed by* Sparse-Φ *in*

$$\mathcal{O}\left(\min\left\{\frac{n^2}{M}, nM\log_\sigma n\right\} + \frac{n^2}{M(\log_\sigma n)^2} + n\log_{\frac{M}{B}}\frac{n}{B}\right) = \mathcal{O}\left(\frac{n^2}{M} + n\log_{\frac{M}{B}}\frac{n}{B}\right) \text{ time}$$

and $\mathcal{O}\left(\dfrac{n^2}{MB(\log_\sigma n)^2} + \dfrac{n}{B}\log_{\frac{M}{B}}\dfrac{n}{B}\right)$ *I/Os using $\mathcal{O}(n/B)$ blocks of disk space.*

Consider then SuccinctIrreducible. Since we cannot make $\mathrm{PLCP}_{\mathrm{succ}}$ and R arbitrarily small, we must be able to handle them even if they do not fit in RAM. We do this by dividing them into segments that are small enough. Consider first the computation of R. While scanning BWT and SA we create a list of irreducible positions, which are then distributed into files corresponding to segments of R. Then each segment of R can be computed by scanning the corresponding file. Similarly, the irreducible bits in $\mathrm{PLCP}_{\mathrm{succ}}$ are set one segment at

a time by reading the $2i + \text{PLCP}[i]$ values from disk, which are now in separate files for each segment. The reducible bits in $\text{PLCP}_{\text{succ}}$ can be easily set by scanning $\text{PLCP}_{\text{succ}}$ and R simultaneously. None of this increases the complexity of the algorithm.

Finally, the last stage of SuccinctIrreducible is performed as follows when $\text{PLCP}_{\text{succ}}$ does not fit in RAM:

3.1. Scan SA and store each value $\text{SA}[i]$ into the file corresponding to the $\text{PLCP}_{\text{succ}}$ segment that contains the $(\text{SA}[i]+1)^{\text{th}}$ 1-bit in $\text{PLCP}_{\text{succ}}$.

3.2. For each segment of $\text{PLCP}_{\text{succ}}$ read the $\text{SA}[i]$ values from the corresponding file, compute $\text{LCP}[i]$ by a select query, and write it to a separate file for each segment.

3.3. Scan SA and for each element $\text{SA}[i]$ determine which segment file contains $\text{LCP}[i]$ and move it to the final output file.

Again, none of this increases the complexity of the algorithm.

The following theorem summarizes the complexities of SuccinctIrreducible. The $\mathcal{O}(n \log n)$ and $\mathcal{O}((n \log \sigma)/B)$ terms come from the irreducible lcp comparisons.

▶ **Theorem 7.** *Given a text of length n over an integer alphabet $[0..\sigma)$ and its suffix array and BWT, the associated LCP array is computed by* SuccinctIrreducible *in*

$$\mathcal{O}\left(\frac{n^2}{M(\log_\sigma n)^2} + n \log n\right) \text{ time and } \mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n \log \sigma}{B} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right) \text{ I/Os}$$

using $\mathcal{O}(n/B)$ blocks of disk space.

In practice, we have noticed that the n select queries performed at the last stage of SuccinctIrreducible often dominate the time. In the implementation, we have replaced $\text{PLCP}_{\text{succ}}$ with the plain PLCP array, but only in the last stage. That is, instead of loading a segment of $\text{PLCP}_{\text{succ}}$ into RAM, we construct a segment of PLCP in RAM by reading a part of $\text{PLCP}_{\text{succ}}$ from disk. Then we use simple accesses to PLCP instead of select queries on $\text{PLCP}_{\text{succ}}$. This modification does not affect the time or I/O complexities of the algorithm.

As a final note, SuccinctIrreducible needs the BWT. Any suffix array construction algorithm can be modified to compute the BWT too with little overhead by storing $\text{BWT}[i]$ together with $\text{SA}[i]$. Since the algorithm has to access $\text{X}[\text{SA}[i]]$ at some point, we can compute $\text{BWT}[i]$ at the same time. We have also implemented a simple external memory algorithm for computing BWT from SA and show its performance in the next section.

6 Experimental Results

Algorithms. We performed experiments using the following algorithms:

- LCPscan, the fastest external-memory LCP array construction algorithm in previous studies [13]. The number of rounds of partial processing in LCPscan was set to 4, as this gives a similar peak disk space usage ($\sim 16n$ bytes) to the new algorithms presented in this paper (see [13] for more details). In our experiments LCPscan serves as a baseline.
- SE-SΦ, the semi-external version of the Sparse-Φ algorithm described by Kärkkäinen et al. [16] (see also Section 3 of this paper).
- EM-SΦ, the fully external-memory version of the Sparse-Φ algorithm described in Sections 3–5. The algorithm is the first contribution of this paper.
- EM-PLCP, the external-memory version of SuccinctIrreducible algorithm described in this paper restricted to perform only Step 1 and 2, i.e., the algorithm produces the $\text{PLCP}_{\text{succ}}$ array but does not convert it to LCP array. We separately consider the construction of $\text{PLCP}_{\text{succ}}$ because first, for some applications computing $\text{PLCP}_{\text{succ}}$ is sufficient

Table 3 Statistics of data used in the experiments. In addition to basic parameters, we show the percentage of irreducible lcp values among all lcp values (expression $100r/n$, where r denotes the number of irreducible lcps) and the average length of the irreducible lcp value (Σ_r/r, where Σ_r is the sum of all irreducible lcps).

Name	$n/2^{30}$	σ	$100r/n$	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
dna	512.0	6	16.13	27.25
debruijn	128.0	2	99.26	35.01

and avoiding the conversion to LCP array is a big time save, and second, this allows us to visualize differences in the methods that convert $\text{PLCP}_{\text{succ}}$ to LCP. Note: small files than can be handled by the original semi-external version of SuccinctIrreducible are processed using the original algorithm from [16]. The increase in I/O and runtime that occurs when we switch to fully-external procedure is discussed in one of the experiments.

- SE-SI, the semi-external version of the SuccinctIrreducible algorithm. It first runs EM-PLCP and then converts $\text{PLCP}_{\text{succ}}$ (held in RAM) to LCP using select queries as originally described [16]. To implement the select queries, we use the variant of the *darray* data structure [29] described in [16]. We set the darray overhead to 6.25% as we did not observe a significant speedup from using more space (e.g., increasing the overhead to 50% speeds up select queries only by about 10%).
  This algorithm is essentially identical to the original semi-external algorithm in [16] when the text and the bitvectors $\mathsf{R}[0..n]$ and $\text{PLCP}_{\text{succ}}[0..2n)$ fit in RAM, but it can also extend beyond that limit since it uses EM-PLCP. It is still a semi-external algorithm though as it needs to have enough RAM for $\text{PLCP}_{\text{succ}}[0..2n)$ in the last stage.
- EM-SI, the fully-external version of the SuccinctIrreducible algorithm described in Sections 3–5. It first uses EM-PLCP to compute $\text{PLCP}_{\text{succ}}$ and then computes LCP using plain accesses to PLCP segments instead of select queries on $\text{PLCP}_{\text{succ}}$ segments as described in Section 5. The time and I/O of EM-PLCP is included in the runtime and I/O volume of EM-SI. Together with EM-PLCP this algorithm is the second contribution of this paper.

All algorithms use 8 bits to represent characters and 40 bits to represent integers. The implementations of all LCP array construction algorithms used in experiments are available at `http://www.cs.helsinki.fi/group/pads/`.

Datasets. For the experiments we used the following files varying in the number of repetitions and alphabet size (see Table 3 for some statistics):

- kernel: a concatenation of ∼10.7 million source files from over 300 versions of Linux kernel [1]. 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;

[1] `http://www.kernel.org/`

- wiki: a concatenation of English Wiki dumps (Wikipedia, Wikisource, Wikibooks, Wikinews, Wikiquote, Wikiversity, and Wikivoyage [2]) dated 20160203 in XML;
- dna: a collection of DNA reads (short fragments produced by a sequencing machine) from multiple human genomes[3] filtered from symbols other than {A, C, G, T, N} and newline;
- 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. A file of length n is obtained as a prefix of a De Bruijn sequence of order $\lceil \log n \rceil$. It contains nearly n irreducible lcps with total length of nearly $n \log n$ (see [16, Lemma 5]) which is the worst case for LCPscan, SE-SI and EM-SI algorithms.

Setup. We performed experiments on a machine equipped with two six-core 1.9 GHz Intel Xeon E5-2420 CPUs with 15 MiB L3 cache and 120 GiB of DDR3 RAM. For experiments we limited the RAM in the system (with the kernel boot flag) to 4 GiB and all algorithms were allowed to use 3.5 GiB. 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.

The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.13.0. All programs were compiled using `g++` version 4.9.2 with `-O3 -DNDEBUG` options. All tested LCP array construction algorithms are sequential, i.e., only a single thread of execution was used for computation. In the last experiment we used parallel algorithms to compute SA and BWT in order to demonstrate the performance of currently fastest methods but those measurements have no bearing on the findings of this paper. All reported runtimes are wallclock (real) times.

Experiments. In the first experiment we compare the scalability of the new external-memory LCP array construction algorithms described in this paper (EM-SΦ, EM-SI) to LCPscan. We executed the algorithms on increasing length prefixes of testfiles using 3.5 GiB of RAM and measured the runtime and the I/O volume.

The results are presented in Figure 2. The performance of EM-SI, similarly to LCPscan, is related to the number of irreducible lcp values (see Table 3). However, avoiding the external-memory sorting gives EM-SI a consistent speed and I/O advantage (of about $60n$ bytes) over LCPscan. The EM-SI algorithm is at least two times faster than LCPscan and even more on highly repetitive data. The computation time can be further reduced by 30–65% if one stops at the $\mathrm{PLCP}_{\mathrm{succ}}$ array. Overall, if the BWT is given as input alongside the text and the suffix array, EM-SI is the fastest way to compute the LCP array. Even if we include the cost of the standalone construction of BWT from the suffix array, the algorithm still outperforms LCPscan.

The performance of EM-SΦ is also related to the number of irreducible lcp values though in a different way than EM-SI. Whenever during step 3.1 the exact value of $\mathrm{PLCP}[i]$ can be deduced from the lower/upper bounds on $\mathrm{PLCP}[i]$ (i.e., $\ell_{\min} = \ell_{\max}$), the algorithm does not write any data to disk. All other pairs are written to disk and processed in step 3.2, which usually dominates the runtime. From Lemma 4 we expect the number of skipped pairs to be high if the number of irreducible lcp values is low, and thus the algorithm runs faster and uses less I/O (by about $20n$ bytes) on kernel and geo testfiles. However, even on the non-repetitive data, EM-SΦ is still about two times faster than LCPscan. Furthermore, when BWT is not available it usually also outperforms EM-SI, making it the algorithm of choice in this case.

[2] `http://dumps.wikimedia.org/`
[3] `http://www.1000genomes.org/`

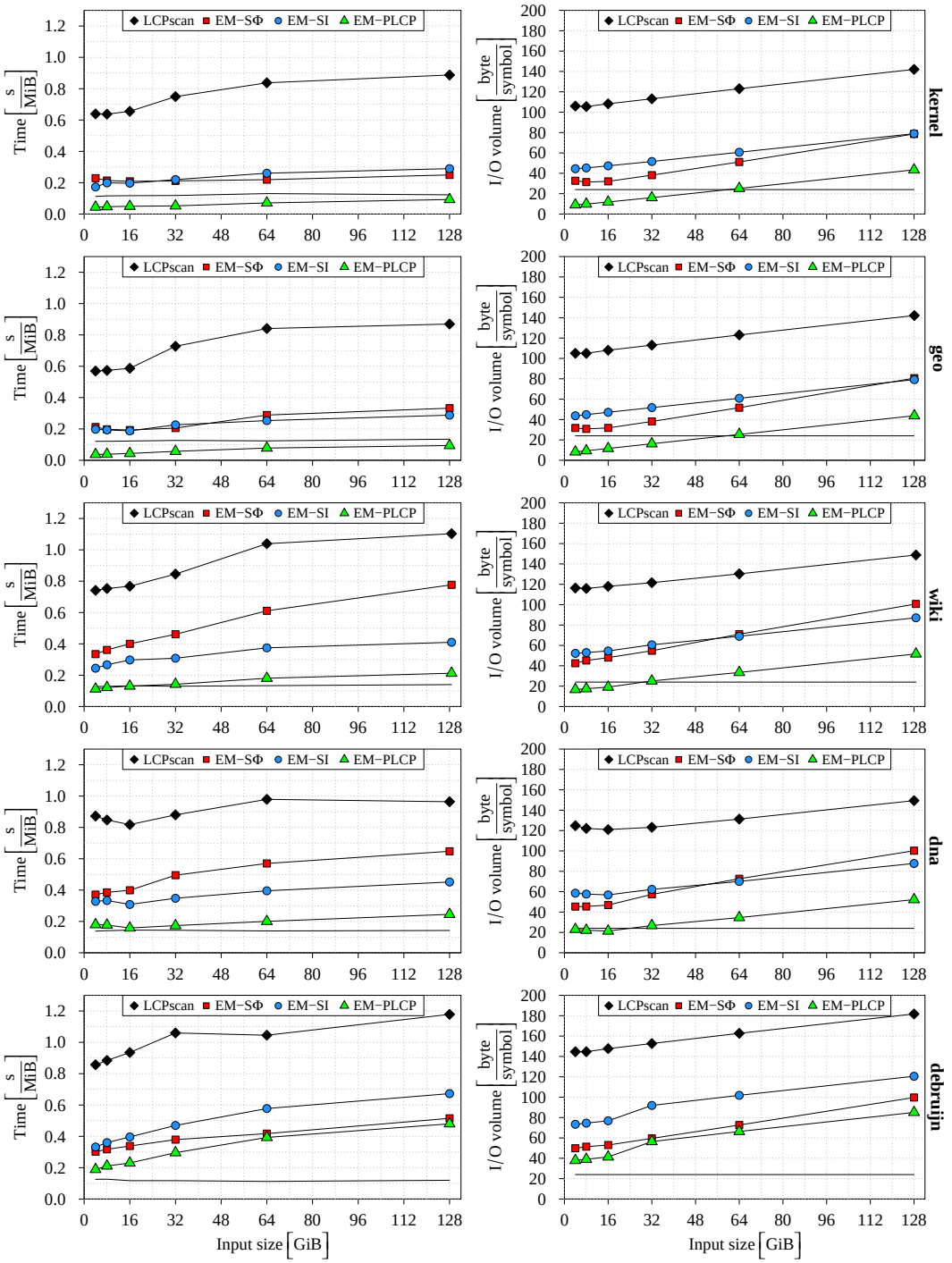

Figure 2 Comparison of the runtime (left; in seconds per MiB of input text) and I/O volume (right; in bytes per input symbol) of the new external-memory LCP array construction algorithms (EM-SΦ, EM-SI) to LCPscan. All algorithms were allowed to use 3.5 GiB of RAM. The unlabeled curve shows the runtime and I/O volume of the standalone external-memory construction of BWT from SA.

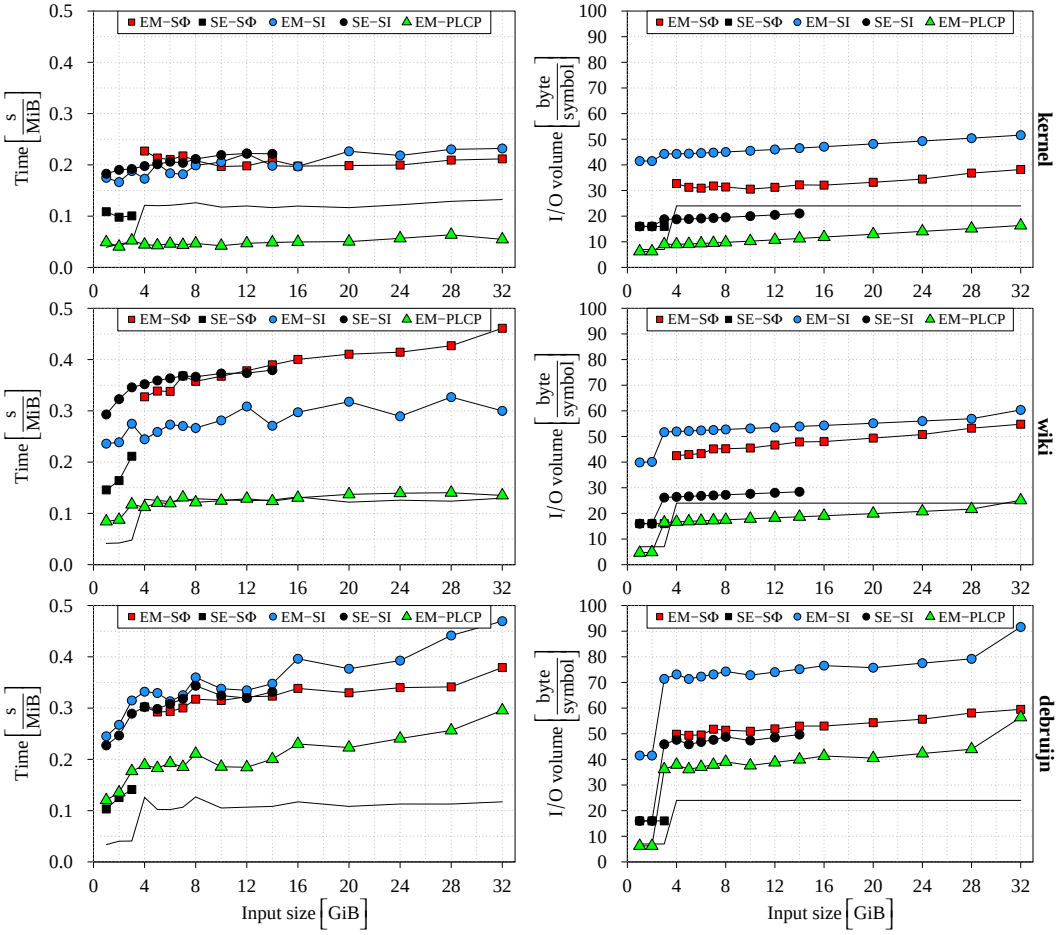

Figure 3 Comparison of the runtime and I/O volume of the new external-memory LCP array construction algorithms (EM-SΦ, EM-SI), to their original semi-external counterparts (SE-SΦ, SE-SI) [16]. The setup is analogous to Figure 2, i.e., all algorithms are using 3.5 GiB of RAM. The unlabeled curve shows the runtime and I/O volume of the standalone external-memory construction of BWT from SA (when the text fits in RAM, we run a semi-external version that needs less I/O).

In the second experiment we compare the EM-SΦ and EM-SI algorithms to their semi-external counterparts SE-SΦ and SE-SI. More precisely, we analyse the transition between semi-external and fully-external algorithms in terms of runtime and I/O volume.

The results are given in Figure 3. First, observe that at the point where we no longer can accommodate the text and bitvectors $\text{PLCP}_{\text{succ}}$ and R in RAM (i.e., between 2 and 3 GiB prefixes) the I/O volume of EM-PLCP (and thus also SE-SI and EM-SI) increases proportionally to the number of irreducible lcp values (up to $30n$ bytes for debruijn). The extra I/O accounts mostly for scanning and does not significantly affect the runtime. Second, note the difference of $25n$ bytes in the I/O volume of SE-SI and EM-SI. While SE-SI uses the original semi-external method that performs select-queries over $\text{PLCP}_{\text{succ}}$ bitvector (kept in RAM) to convert $\text{PLCP}_{\text{succ}}$ into LCP array [16], EM-SI uses the fully external-memory (and thus more I/O-demanding) method that does not require $\text{PLCP}_{\text{succ}}$ to fit in RAM, and furthermore, replaces the select-queries with simple lookups (see Section 5). The I/O increase is however compensated by faster computation and the fully-external method is either comparable (kernel, debruijn) or faster (wiki) than the semi-external method.

Table 4 Experimental results on the 512 GiB instance of dna testfile using 120 GiB of RAM. The disk usage column gives a peak disk space usage including the input and output of the given algorithm. pEM-BWT is a simple external-memory algorithm constructing BWT from SA. For comparison with pSAscan we also parallelized the computation in pEM-BWT.

Algorithm	Runtime	I/O volume	Disk usage
pSAscan	2.51 days	16.63 TiB	3.75 TiB
pEM-BWT	0.54 days	12.00 TiB	5.50 TiB
LCPscan	5.04 days	62.22 TiB	6.33 TiB
EM-SI	2.16 days	25.89 TiB	6.12 TiB
EM-SΦ	2.69 days	20.77 TiB	5.50 TiB
EM-PLCP	0.77 days	8.27 TiB	4.36 TiB

The transition between the SE-SΦ algorithm and EM-SΦ shows an increase in I/O by a factor 2–3 at the point where the text no longer fits in RAM. This is due to the fact that SE-SΦ reads the text once while EM-SΦ reads it in steps 2.2 and 3.2. Similarly, while SE-SΦ reads SA once in Step 3, EM-SΦ needs two scans (steps 3.1 and 3.3). Moreover EM-SΦ computes the values $\ell_{\min}$ and $\ell_{\max}$ twice for every processed pair (step 3.1 and 3.3), whereas SE-SΦ does it only once. This causes a slowdown by a factor 1.6–2.3 at the transition point, since these computations involve random accesses to the PLCP_q array and thus attract multiple cache misses. Note however that the increase in runtime and I/O volume is much bigger if we consider the transition from SE-SΦ to LCPscan.

In the last experiment we compare the performance of the new external-memory algorithms EM-SI and EM-SΦ to LCPscan on a full 512 GiB instance of the dna file using all 120 GiB of RAM available on our test machine. Unlike in previous experiments, here we use LCPscan with six rounds of processing since the four-round version ran out of disk space. The performance of both modes is very similar.

The results are given in Table 4. For comparison we also present the resources used by pSAscan [15] – currently the fastest practical way to construct suffix arrays in external memory, and pEM-BWT – a simple parallel BWT-from-SA construction. Note that in all previous experiments the algorithm for computing BWT from SA was sequential. Here we decided to use the parallel version to make it comparable to pSAscan. The results are consistent with previous experiments, i.e., both EM-SI and EM-SΦ are about two times faster than LCPscan. When BWT is available alongside the suffix array, the processing time is smaller for EM-SI. Otherwise, we need to additionally run a separate BWT construction and as a result EM-SΦ has a slight edge over EM-SI. Finally, if one wishes to only compute the PLCP array the processing time of EM-SI is reduced by about 65%.

Lastly, note that the new algorithms use a fairly moderate disk space. For EM-PLCP a peak disk space usage is either achieved after step 1.1 where in addition to input, we have $2r$ integers and the R bitvector stored on disk (recall that r is the number of irreducible lcp values) or after step 2.3 where in addition to input we store the $\mathrm{PLCP}_{\mathrm{succ}}$ bitvector on disk. The resulting disk usage is $7.125n + \max(10r, 0.125n)$ bytes, assuming 40-bit integers. Given that for the input instance we have $10r = 1.61n$ (see Table 3), the peak disk usage is $8.735n$, i.e., 4.36 TiB. For EM-SI we obtain the full LCP array in the last stage and thus the disk usage increases to $7.125n + \max(10r, 5.125n)$ bytes, i.e., $12.25n$ for the tested input. For EM-SΦ the disk usage is either maximized after step 3.1 or at the end of computation and is equal to at most $16n$ bytes (we ignore the space needed for PLCP_q). In practice it can be less due to skipped $(i, \Phi[i])$ pairs (see the discussion above) but it cannot be easily expressed

in terms and n and r. In our experiment 55% of all pairs were skipped resulting in a peak disk usage of about $11n$ bytes or 5.5 TiB. We leave details for the full version of the paper.

7 Concluding Remarks

We have described two new external memory algorithms for LCP array construction. Our experiments show that the new algorithms are about two times faster than the state of the art. A common feature of the new algorithms is their avoidance of external-memory sorting.

One of the possible avenues for future work is reducing the disk space usage. In LCPscan this is accomplished by splitting and processing the input text in multiple parts. Similar partitioning techniques can be applied to reduce the disk space usage of the presented algorithms. Although the new algorithms are already quite disk space efficient, with partitioning their peak disk space usage can be guaranteed to be little more than what is needed for the input and the output.

Another possibility for improvement is to use parallelism. Both of the new algorithms are compute-bound rather than I/O-bound in some stages of the computation. Parallel computation in such stages can reduce the running time further.

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 M. J. Bauer, A. J. Cox, G. Rosone, and M. Sciortino. Lightweight LCP construction for next-generation sequencing datasets. In *Proceedings of the 12th Workshop on Algorithms in Bioinformatics (WABI 2012)*, volume 7534 of *LNCS*, pages 326–337. Springer, 2012. doi:10.1007/978-3-642-33122-0_26.

3 T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013. doi:10.1016/j.jda.2012.07.007.

4 T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proceedings of the 2013 Workshop on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 88–102. SIAM, 2013. doi:10.1137/1.9781611972931.8.

5 D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1998.

6 F. A. da Louza, G. P. Telles, and C. D. de Aguiar Ciferri. External memory generalized suffix and LCP arrays construction. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *LNCS*, pages 201–210. Springer, 2013. doi:10.1007/978-3-642-38905-4_20.

7 R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM J. Exp. Algor.*, 12:3.4:1–3.4:24, August 2008. doi:10.1145/1227161.1402296.

8 M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013)*, pages 197–206. ACM, 2013. doi:10.1145/2442516.2442536.

9 S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proceedings of the 2011 Workshop on Algorithm Engineering and Experiments (ALENEX 2011)*, pages 25–34. SIAM, 2011. doi:10.1137/1.9781611972917.3.

10 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.

11 J. Kärkkäinen and D. Kempa. Engineering a lightweight external memory suffix array construction algorithm. In *Proceedings of the 2nd International Conference on Algorithms for Big Data (ICABD 2014)*, volume 1146 of *CEUR Workshop Proceedings*, pages 53–60. CEUR-WS.org, 2014.

12 J. Kärkkäinen and D. Kempa. LCP array construction in external memory. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 412–423. Springer, 2014. doi:10.1007/978-3-319-07959-2_35.

13 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.

14 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.

15 J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Parallel external memory suffix sorting. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 329–342. Springer, 2015. doi:10.1007/978-3-319-19929-0_28.

16 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.

17 J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.

18 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.

19 W. Liu, G. Nong, W. H. Chan, and Y. Wu. Induced sorting suffixes in external memory with better design and less space. In *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, volume 9309 of *LNCS*, pages 83–94. Springer, 2015. doi:10.1007/978-3-319-23826-5_9.

20 V. Mäkinen. Compact suffix array – a space efficient full-text index. *Fund. Inform.*, 56(1–2):191–210, 2003.

21 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.

22 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.

23 G. Manzini. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 14th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2004)*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004. doi:10.1007/978-3-540-27810-8_32.

24 I. Munro. Tables. In *Proceedings of the 16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996. doi:10.1007/3-540-62034-6_35.

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

26 G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu. Induced sorting suffixes in external memory. *ACM Trans. Inf. Syst.*, 33(3), February 2015. doi:10.1145/2699665.

27 G. Nong, W. H. Chan, S. Zhang, and X. F. Guan. Suffix array construction in external memory using d-critical substrings. *ACM Trans. Inf. Syst.*, 32(1), January 2014. doi:10.1145/2518175.

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

29 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 2007 Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*. SIAM, 2007. doi:10.1137/1.9781611972870.6.

30 S. J. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008. doi:10.1007/978-3-540-92182-0_14.

31 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.

32 J. Shun. Fast parallel computation of longest common prefixes. In *Proceedings of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014)*, pages 387–398. IEEE, 2014. doi:10.1109/SC.2014.37.

33 J. Sirén. Sampled longest common prefix array. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010)*, volume 6129 of *LNCS*, pages 227–237. Springer, 2010. doi:10.1007/978-3-642-13509-5_21.

34 J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theoretical Computer Science*, 2(4):305–474, 2006. doi:10.1561/0400000014.