LCP Array Construction in External Memory*

Juha Kärkkäinen and Dominik Kempa

Department of Computer Science, University of Helsinki, and
Helsinki Institute for Information Technology HIIT, Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

Abstract. One of the most important data structures for string processing, the suffix array, needs to be augmented with the longest-common-prefix (LCP) array in numerous applications. We describe the first external memory algorithm for constructing the LCP array given the suffix array as input. The only previous way to compute the LCP array for data that is bigger than the RAM is to use a suffix array construction algorithm with complex modifications to produce the LCP array as a by-product. Compared to the best prior method, our algorithm needs much less disk space (by more than a factor of three) and is significantly faster. Furthermore, our algorithm can be combined with any suffix array construction algorithm including a better one developed in the future.

1 Introduction

The suffix array [16,8], a lexicographically sorted array of the suffixes of a text, is the most important data structure in modern string processing. It is the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [18]. Modern text books spend dozens of pages in describing applications of suffix arrays, see e.g. [20]. In many of those applications, the suffix array needs to be augmented with the longest-common-prefix (LCP) array, which stores the lengths of the longest common prefixes between lexicographically adjacent suffixes (see e.g. [1,20]).

The construction of these data structures is a bottleneck in many of the applications. There are numerous suffix array construction algorithms (SACAs) [21] including linear time internal memory SACAs [13,19] as well as external memory SACAs with the optimal I/O complexity [4,3]. There are also simple, linear time internal memory LCP array construction algorithms (LACAs) [14,11] that take the suffix array and the text as input, but external memory LCP array construction remains a problem. In this paper, we describe the first external memory LACA.

Related Work. The first LACA by Kasai et al. [14] is simple and runs in linear time but needs a lot a space (the text plus $3n$ integers). Several later algorithms

* This research is partially supported by the Academy of Finland through grant 118653 (ALGODAN).

J. Gudmundsson and J. Katajainen (Eds.): SEA 2014, LNCS 8504, pp. 412–423, 2014.

aimed at reducing the space [15,17,22,11,7,2]. Some of the algorithms can even be made semi-external, i.e., they keep most of the data structures on disk but need to have at least the full text in RAM [22,11].

When the text size exceeds the RAM size, the only prior option for constructing the LCP array is to use an external memory SACA modified to compute the LCP array too during the construction [12,3] — we call these SLACAs — but there are several drawbacks to this approach. First, they add a substantial amount of complication to already complicated algorithms, and this complication is repeated for each SLACA, whereas a proper LACA can be combined with any SACA without a modification. Second, while theoretically the complexity of the algorithms does not change, adding the LCP computation increases the running time significantly in practice. Third, the SLACAs need a lot of disk space, so much in fact that the disk space is likely to be the biggest problem in scaling the algorithms for bigger data. For a text of length n, the best SLACA implementation, eSAIS [3], needs $54n$ bytes of disk space when computing the suffix and LCP arrays compared to only $28n$ bytes of disk space when computing only the suffix array.

Our Contribution. The new LACA, called LCPscan, is the first external memory LACA that is independent of any suffix array construction algorithm. The algorithm combines elements from several internal memory LACAs such as the original LACA [14], the Φ algorithm [11] and the irreducible LCP algorithm [11], adds some new twists such as a new method for identifying irreducible LCP values, and implements everything using external memory scanning and sorting. The main new idea, however, is to divide the text into blocks that are small enough to fit in RAM and then scan the rest of the text once for each block. A similar approach has been recently applied to computing the Burrows–Wheeler transform [6], the Lempel–Ziv factorization [10], and the suffix array [9]. This approach leads to a quadratic complexity in theory: $\mathcal{O}\left(\frac{n^2}{M \log_\sigma n} + n \log_{\frac{M}{B}} \frac{n}{B}\right)$ time and $\mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$ I/Os. However, in practice the size of the text would have to be more than about 100 times the size of the RAM, before the quadratic part of the computation would start to dominate. Up to that point, LCPscan is the fastest way to construct the LCP array in external memory as shown by our experiments. Furthermore, LCPscan needs just $16n$ bytes of disk space, which is less than a third of the disk usage of the best previous method [3].

2 Preliminaries

Strings. 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 the alphabet $[0..\sigma)$. 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". Similarly, we write $\mathsf{X}[0..i)$ to denote the *prefix* of X of length i. $\mathsf{X}[i..j)$ is the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j-1]$ of X that starts at position i and ends at position $j-1$.

Suffix Array. The *suffix array* [16] SA of X is an array $\mathsf{SA}[0..n]$ which contains a permutation of the integers $[0..n]$ such that $\mathsf{X}[\mathsf{SA}[0]..n) < \mathsf{X}[\mathsf{SA}[1]..n) < \cdots < \mathsf{X}[\mathsf{SA}[n]..n)$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n)$ is the $(j+1)^{\text{th}}$ suffix of X in ascending lexicographical order. The *inverse suffix array* ISA is the inverse permutation of SA, that is $\mathsf{ISA}[i] = j$ iff $\mathsf{SA}[j] = i$. Conceptually, $\mathsf{ISA}[i]$ tells us the position of suffix i in SA. Another representation of the permutation is the Φ *array* [11] $\Phi[0..n)$ defined by $\Phi[\mathsf{SA}[j]] = \mathsf{SA}[j-1]$ for $j \in [1..n]$. In other words, the suffix $\Phi[i]$ is the immediate lexicographical predecessor of the suffix i.

LCP Array. Let $\mathsf{lcp}(i,j)$ denote the length of the longest-common-prefix (LCP) of suffix i and suffix j. For example, in the string $\mathsf{X} = \mathit{ccccatcat}$, $\mathsf{lcp}(0,3) = 2 = |\mathit{cc}|$, and $\mathsf{lcp}(4,7) = 3 = |\mathit{cat}|$. The *longest-common-prefix array* [14], $\mathrm{LCP}[1..n]$, is defined such that $\mathrm{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i-1])$ for $i \in [1..n]$. The *permuted LCP array* [11] $\mathrm{PLCP}[0..n)$ is the LCP array permuted from the lexicographical order into the text order, i.e., $\mathrm{PLCP}[\mathsf{SA}[j]] = \mathrm{LCP}[j]$ for $j \in [1..n]$. Then $\mathrm{PLCP}[i] = \mathsf{lcp}(i, \Phi[i])$ for all $i \in [0..n)$. The following result is the basis of all efficient LACAs.

Lemma 1. *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 $\Phi[i] + \mathrm{PLCP}[i] \le \Phi[j] + \mathrm{PLCP}[j]$.*

Proof. As shown in [14,11], $i + \mathrm{PLCP}[i] \le (i+1) + \mathrm{PLCP}[i+1]$ for all $i \in [0..n-2]$, an iterative application of which results the first part of the claim. The second part follows by symmetry. □

3 Basic Algorithm

In this section, we describe the basic $\mathsf{LCPscan}$ algorithm for computing the LCP array of a string X given X and the suffix array SA of X. In the next two sections, we describe further optimizations and analyze the theoretical and practical properties of the algorithm.

The basic approach of the algorithm is similar to the original linear time LACA by Kasai et al. [14] and to the Φ algorithm introduced in [11]. The main steps in the computation are:

1. Compute ISA and Φ from SA.
2. Compute PLCP from X and Φ.
3. Compute LCP from ISA and PLCP.

The first and the third step are easy to implement in external memory using sorting. In step 1, we scan the suffix array creating a triple $(i, \mathsf{SA}[i], \mathsf{SA}[i-1])$ for each $i \in [1..n]$. When the triples are sorted by the middle component, the sequence of first components forms $\mathsf{ISA}[0..n)$ and the sequence of third components forms $\Phi[0..n)$. In the third step, we similarly sort the pairs $(\mathsf{ISA}[i], \mathrm{PLCP}[i])$, $i \in [0..n)$, by the first component obtaining $\mathrm{LCP}[1..n]$ as the sequence of the second components.

In the middle step, we partition the text into $\mathcal{O}(n/m)$ blocks of size at most m and process them one at a time. The block size m is chosen so that one block

of text fits in RAM together with a constant number of disk buffers. For each block $\mathsf{X}[s..e)$, $0 \le s \le e < n$, we want to compute $\mathrm{PLCP}[s..e)$ from $\Phi[s..e)$ and X. Only the block $\mathsf{X}[s..e)$ is kept in RAM and the rest of X is scanned once. During the computation, we maintain triples (i, j, ℓ), where $i \in [s..e)$, $j = \Phi[i]$ and $\ell \in [0..\mathsf{lcp}(i, j)]$. Each triple starts with $\ell = 0$ and ends with $\ell = \mathsf{lcp}(i, j)$ allowing us to set $\mathrm{PLCP}[i] = \ell$. The main idea is to process the triples in the order of the second component so that we can complete all the lcp computations during a single scan of X.

A small complication in the computation is dealing with the block boundaries. The triple (i, j, ℓ) is first created when processing the block that contains i but its computation may be finished when processing a different block, the one that contains $i + \mathsf{lcp}(i, j)$. We keep a set R_y to hold triples that cross a block boundary y. Thus, the processing of a block $\mathsf{X}[s..e)$ has R_s as an extra input and R_e as an extra output. Furthermore, the main output is not necessarily $\mathrm{PLCP}[s..e)$ but $\mathrm{PLCP}[s'..e')$ for some $s' \le s$ and $e' \le e$. The full algorithm for processing a block is given in Fig. 1.

ProcessBlock$(s, e, \mathsf{X}, \Phi[s..e), \mathsf{R}_s)$
1: $\mathsf{Q} \leftarrow \mathsf{R}_s \cup \{(i, \Phi[i], 0) : i \in [s..e)\}$
2: sort Q by the second component
3: $j_{\mathrm{prev}} \leftarrow s$, $\ell_{\mathrm{prev}} \leftarrow 0$
4: $\mathsf{L} \leftarrow \mathsf{R}_e \leftarrow \emptyset$
5: **for** $(i, j, \ell) \in \mathsf{Q}$ **do**
6: $\quad \ell \leftarrow \max(\ell, \ell_{\mathrm{prev}} + j_{\mathrm{prev}} - j)$
7: $\quad$ **while** $i + \ell < e$ **and** $j + \ell < n$ **and** $\mathsf{X}[i + \ell] = \mathsf{X}[j + \ell]$ **do** $\ell \leftarrow \ell + 1$
8: $\quad$ **if** $i + \ell \ge e$ **and** $j + \ell < n$ **then** $\mathsf{R}_e \leftarrow \mathsf{R}_e \cup \{(i, j, \ell)\}$
9: $\quad$ **else** $\mathsf{L} \leftarrow \mathsf{L} \cup \{(i, \ell)\}$
10: $\quad j_{\mathrm{prev}} \leftarrow j$, $\ell_{\mathrm{prev}} \leftarrow \ell$
11: sort L by the first component
12: $e' \leftarrow \min(\{e\} \cup \{i : (i, j, \ell) \in \mathsf{R}_e\})$
13: $s' \leftarrow \min(\{e'\} \cup \{i : (i, \ell) \in \mathsf{L}\})$
14: $\mathrm{PLCP}[s'..e') \leftarrow \{\ell : (i, \ell) \in \mathsf{L}\}$
15: **return** $\mathrm{PLCP}[s'..e')$, R_e

Fig. 1. Process a text block $\mathsf{X}[s..e)$

A subtle point in the algorithm is line 6, where we may set $\ell = \ell_{\mathrm{prev}} + j_{\mathrm{prev}} - j$. First, this is safe because $j_{\mathrm{prev}} + \ell_{\mathrm{prev}} \le j + \mathsf{lcp}(i, j)$ by Lemma 1. Second, this ensures that $j + \ell$ never decreases during the algorithm and thus the accesses to $\mathsf{X}[j + \ell]$ are purely sequential. The other access to the text, $\mathsf{X}[i + \ell]$, can be non-sequential, but we always have that $i + \ell \in [s..e)$. Everything else in the algorithm can be done by scanning and sorting and can thus be implemented efficiently in external memory.

4 Irreducible LCP Values

For highly repetitive texts, the longest common prefixes can be very long and cross several block boundaries, and conversely, a single block boundary may be crossed by LCPs starting from several blocks. Thus the sets R_s and R_e in the algorithm can grow big and the sum of their sizes over the whole algorithm could be as large as $\Theta(n^2/m)$. To prevent this, we will employ the irreducible LCP technique introduced in [11].

An LCP value PLCP[i] is said to be *reducible* if $\mathsf{X}[i-1] = \mathsf{X}[\Phi[i]-1]$. Otherwise, in particular when $i = 0$ or $\Phi[i] = 0$, PLCP[i] is *irreducible*. The following key properties of (ir)reducible LCP values were proved in [11].

Lemma 2 ([11]). *If* PLCP[i] *is reducible, then* PLCP[i] = PLCP[$i-1$] $- 1$.

Lemma 3 ([11]). *The sum of all irreducible lcp values is* $\leq 2n \log n$.

We will modify the procedure in Fig. 1 to compute only the irreducible LCP values in the main loop. The first lemma above shows that the reducible values are easy to obtain afterwards. The second lemma above ensures that the total number of boundary crossings of irreducible LCP values is $\mathcal{O}(n + (n \log n)/m)$, which is $\mathcal{O}(n)$ under the reasonable assumption that $m = \Omega(\log n)$.

In the procedure ProcessBlock, we need to discard a triple (i, j, ℓ) if PLCP[i] is reducible, i.e., if $\mathsf{X}[i-1] = \mathsf{X}[\Phi[i]-1]$, which could be done in the main loop when the text scan reaches $\mathsf{X}[j-1]$. However, we can do it already earlier using the following alternative characterization of the irreducible LCP values.

Lemma 4. PLCP[i] *is a reducible value iff* $i > 0$ *and* $\Phi[i-1] = \Phi[i] - 1$ *and* PLCP[$i-1$] > 0.

Proof. If $i > 0$ and $\Phi[i-1] = \Phi[i] - 1$ and PLCP[$i-1$] $= \mathsf{lcp}(i-1, \Phi[i-1]) > 0$, then $\mathsf{X}[i-1] = \mathsf{X}[\Phi[i-1]] = \mathsf{X}[\Phi[i]-1]$ and thus PLCP[i] is reducible. PLCP[i] is irreducible if $i = 0$ (by definition) or if $\Phi[i-1] = \Phi[i] - 1$ and PLCP[$i-1$] $= 0$ (since then $\mathsf{X}[i-1] \neq \mathsf{X}[\Phi[i-1]] = \mathsf{X}[\Phi[i]-1]$). The only thing left to prove is that if PLCP[i] is reducible then $\Phi[i-1] = \Phi[i] - 1$.

Assume by contradiction that PLCP[i] is reducible but $\Phi[i-1] \neq \Phi[i]-1$. Since $\mathsf{X}[i-1] = \mathsf{X}[\Phi[i]-1]$ and $\mathsf{X}[\Phi[i]..n) < \mathsf{X}[i..n)$, we have that $\mathsf{X}[\Phi[i]-1..n) < \mathsf{X}[i-1..n)$. Since the suffix $\Phi[i-1]$ is the immediate lexicographical predecessor of the suffix $i-1$, we must have have $\mathsf{X}[\Phi[i]-1..n) < \mathsf{X}[\Phi[i-1]..n) < \mathsf{X}[i-1..n)$. But this implies that $\mathsf{X}[\Phi[i-1]] = \mathsf{X}[i-1]$ and that $\mathsf{X}[\Phi[i]..n) < \mathsf{X}[\Phi[i-1]+1..n) < \mathsf{X}[i..n)$, which contradicts the suffix $\Phi[i]$ being the immediate lexicographical predecessor of the suffix i. □

Note that PLCP[$i-1$] $= 0$ implies that the suffix $i-1$ is the lexicographically smallest suffix starting with the character $\mathsf{X}[i-1]$. Thus there are at most σ such positions and they can be easily computed from the text and the suffix array. We scan (if σ is small) or sort (if σ is large) the text to compute the character frequencies, which then identify the positions of the relevant suffixes in SA. The other reducible positions can be recognized on line 1 of ProcessBlock

while scanning Φ and only the irreducible positions are added into Q. The only other modification to Algorithm ProcessBlock is on line 14. Since L contains only irreducible LCP values, the missing values in PLCP are filled using Lemma 2.

We are now ready to analyze the complexity of the algorithm in the standard external memory model (see [23]) with RAM size M and disk block size B, both measured in units of $\Theta(\log n)$-bit words. We assume that $M = \Omega(\log n)$, $M = \mathcal{O}(n)$ and $\sigma = \mathcal{O}(n)$.

Theorem 1. *Given a text of length n over an alphabet of size σ and its suffix array, the associated LCP array can be computed with the algorithm described above in*

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

and

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

Proof. The text is divided into $\mathcal{O}(n/m)$ blocks, where m is chosen so that m characters fit in RAM. The size of the RAM is $M \log n$ bits and we can fit $m = \Theta(M \log_\sigma n)$ characters into that space. For each block, we scan the text once with each scan requiring $\mathcal{O}(n)$ time and $\mathcal{O}(n/(B \log_\sigma n))$ I/Os. This gives the first terms in the complexities.

Everything else in the algorithm involves scanning and sorting (tuples of) integers and the total number of integers involved in these scans and sorts is $\mathcal{O}(n)$. This gives the second terms in the complexities. □

5 Practical Improvements

In this section, we describe some practical improvements and analyze the practical properties of LCPscan. In the analysis we assume that the implementation uses one byte for each character and five bytes for each integer.

As championed by the STXXL library [5], pipelining is an important technique for external memory algorithms. That is, instead of writing the output of one stage to disk and then reading it from the disk in the next stage, we execute the two stages simultaneously so that the input of the first stage is fed directly to the next stage. For example, when in the first step of the algorithm we obtain Φ as the third components of the sorted triples, we do not write the Φ values directly to disk. Instead, we identify the reducible positions and discard the corresponding entries of Φ already at this stage. For each irreducible position i, we form the pair $(i, \Phi[i])$ and feed these pairs to a sorter. The sorter uses the block number $\lfloor i/m \rfloor$ as the primary key and the value $\Phi[i]$ as the secondary key. This accomplishes most of the work on lines 1 and 2 in procedure ProcessBlock. The beginning of the actual ProcessBlock procedure will then just read the sorted $(i, \Phi[i])$ pairs, add the third components (0) and merge with R_s to obtain the (i, j, ℓ) triples, which are then immediately processed in the loop on lines 5–10.

Using pipelining throughout and assuming that n elements in the algorithm can be sorted in one pass of multiway mergesort,[1] the total I/O volume of LCPscan is $71n + 40r + \lceil n/m \rceil n$ bytes, where r is the number of irreducible LCP values.

The peak disk usage of the algorithm occurs during the stage described above, where we read the sorted $(i, \mathsf{SA}[i], \mathsf{SA}[i-1])$ triples and output ISA and the irreducible $(i, \Phi[i])$ pairs. All that data requires $20n + 10r$ bytes of disk space, which is $30n$ bytes in the worst case. We also have the suffix array and the text on disk occupying a further $6n$ bytes for a total of $36n$ bytes.

To reduce the disk usage, we divide the text into four superblocks of sizes $0.31n$, $0.27n$, $0.23n$ and $0.19n$, and run the algorithm separately for each superblock. That is, in step 1, we scan SA forming the triple $(i, \mathsf{SA}[i], \mathsf{SA}[i-1])$ only when $\mathsf{SA}[i]$ belongs to the current superblock. The output of the superblock computation is a subsequence of LCP containing only the entries $\mathrm{LCP}[i]$ such that $\mathsf{SA}[i]$ belongs to the current superblock. Once all superblocks have been processed, the LCP subsequences are merged using SA to determine the merging order. In the rest of the algorithm, processing a superblock instead of the full text changes little as each superblock is a contiguous segment in the main data structures Φ, ISA and PLCP. With the superblock division, the peak disk usage is reduced to $16n$ bytes. For example, when processing the third superblock, we need $6n$ bytes for the text and the suffix array, $0.23 \times 30n = 6.9n$ bytes for the $(i, \mathsf{SA}[i], \mathsf{SA}[i-1])$ triples, $(i, \Phi[i])$ pairs and ISA for the superblock, and $(0.31 + 0.27) \times 5n = 2.9n$ bytes for the LCP subsequences for the first two superblocks, which sums up to $15.8n$ bytes. The full $16n$ bytes is needed when merging the LCP subsequences into the final LCP array.

With the division into superblocks, the extra scans of the suffix array and the merging of the LCP subsequences add $30n$ bytes to the I/O volume for a total of $101n + 40r + \lceil n/m \rceil n$ bytes. We are willing to accept the slightly increased running time because the lack of disk space is likely to be a more serious limitation than time.

6 Experimental Results

We have implemented LCPscan as described above using the STXXL library [5] for external memory sorting. As there are no previous external memory LACAs, we compare it to eSAIS [3], the fastest external memory SLACA in previous studies. eSAIS can compute either SA and LCP arrays or only SA, and it is the difference between the two modes that we compare LCPscan against. This reveals if the combination of eSAIS as SACA plus LCPscan is better than eSAIS as SLACA. Combining LCPscan with another SACA such as the recent SAscan [9] might be an even better combination, but we want to focus on the LCP computation alone. The C++ implementations are available at `http://www.cs.helsinki.fi/group/pads/`.

[1] Considering the amount of RAM on modern computers, one pass of merging is a reasonable assumption in practice.

Table 1. 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
countries	64	210	0.14	1294.0
wiki	120	213	17.9	27.1
dna	64	6	20.7	22.0
debruijn	64	2	99.2	34.0

Data Set. For the experiments we selected a range of testfiles varying in the number and length of irreducible LCP values:

- countries: a concatenation of all versions (edit history) of Wikipedia articles about 78 largest countries in the XML format[2]. It contains a small number of large irreducible LCP values,
- wiki: latest English, German and French Wikipedia dumps[3] in the XML format concatenated and truncated to 120GiB. It represents a natural text,
- dna: a collection of DNA reads from a human genome[4] filtered from symbols other than {A, C, G, T, N} and newline. The irreducible LCP values are very short but relatively frequent ($\sim 30\%$),
- 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 $\Theta(n)$ irreducible LCPs of maximal possible total length $\Theta(n \log n)$ [11]. It represents the worst case for LCPscan.

Table 1 gives detailed statistics about the data.

Experiments Setup. We performed experiments on 2 different machines referred to as Platform S (small) and Platform L (large).

Platform S was equipped with a 3.16GHz Intel Core 2 Duo CPU with 6MiB L2 cache and two 320GiB hard drives with a total of 480GiB of usable disk space. For experiments, we artificially restricted the RAM size to 2GiB using the Linux boot option `mem`, and the algorithms were allowed to use at most 1.5GiB.

Platform L was equipped with 1.9GHz Intel Xeon E5-2420 CPU with 15MiB L2 cache and 7.2TiB of disk space striped with RAID0 across 4 local disks of size 1.8TiB. For experiments we restricted the RAM size to 4GiB, and the algorithms were allowed to use at most 3.5GiB.

On both platforms we used a disk block of size 1MiB. The OS was Linux (Ubuntu 12.04, 64bit). All programs were compiled using `g++` version 4.6.4 with `-O3 -DNDEBUG` options. In all experiments we used only a single thread of execution. All reported runtimes are wallclock (real) times.

[2] `http://www.mediawiki.org/wiki/Parameters_to_Special:Export`
[3] `http://dumps.wikimedia.org/`
[4] `http://www.1000genomes.org/`

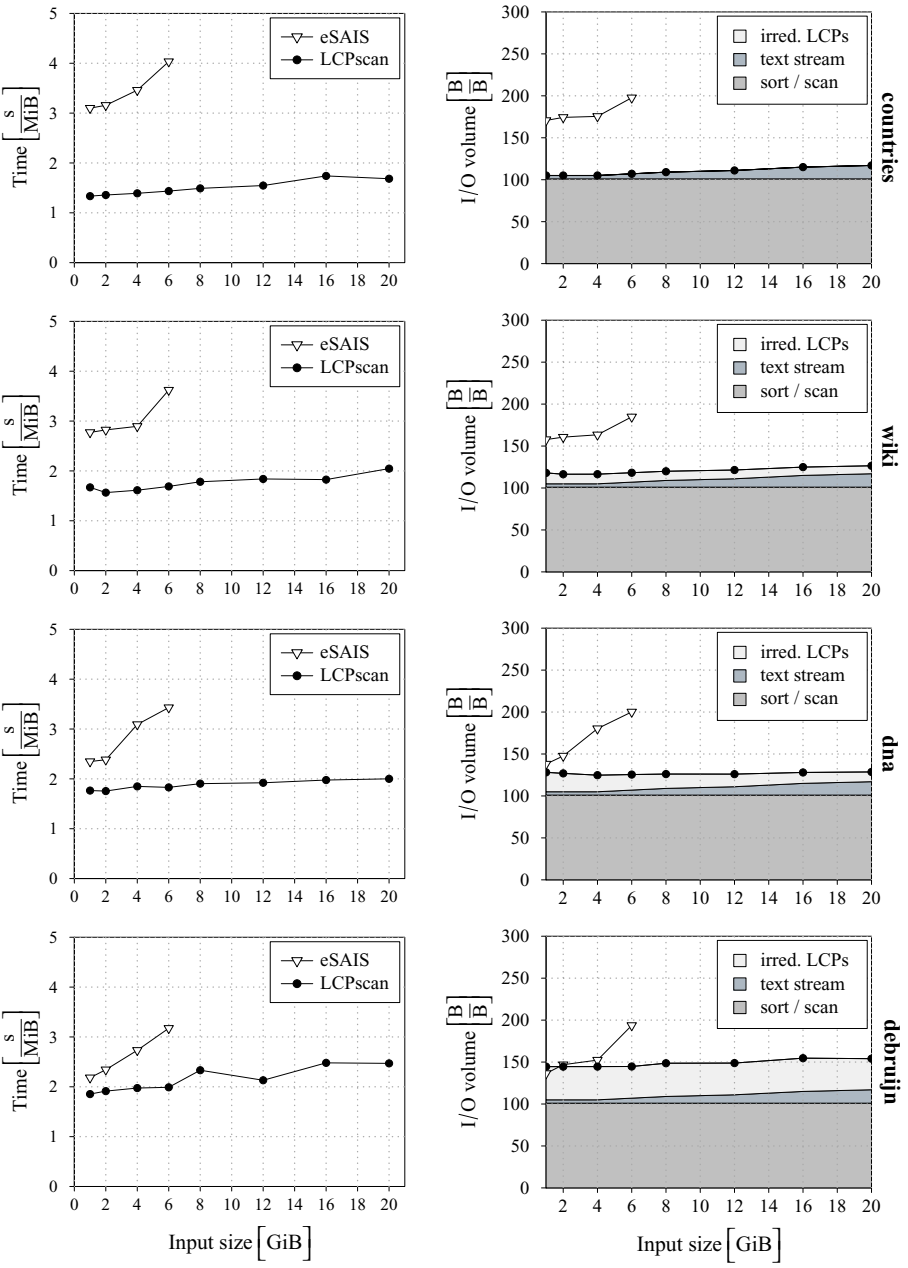

Fig. 2. Experimental results on platform S: comparison of the runtime (left) and normalized I/O volume (right) of LCPscan and eSAIS. For eSAIS, the values are the difference between constructing both SA and LCP arrays and constructing SA only. For the LCPscan I/O volume we show a detailed breakdown into: (bottom) I/Os that involve sorting/scanning of $\Theta(n)$ elements, (middle) I/Os resulting from text scans of which there are $\Theta(n/m)$ and (top) I/Os that only involve processing irreducible values.

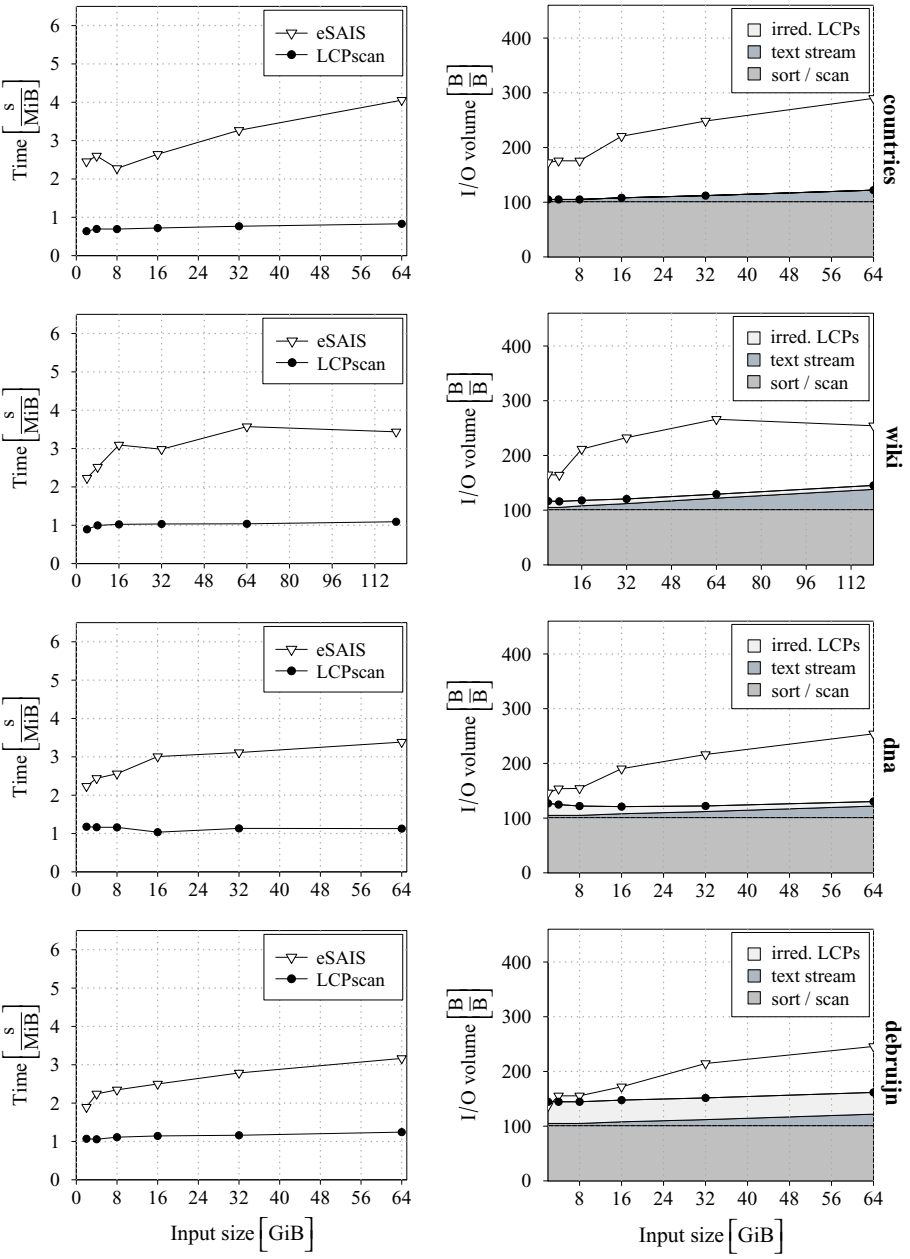

Fig. 3. Experimental results on platform L (analogous to Fig. 2)

Table 2. Summary of experiments on the 120GiB wiki test file. Disk space usage includes input and output.

Algorithm	Runtime	I/O volume	Disk space usage
eSAIS (SA only)	5.0 days	30.5 TiB	3.4 TiB
LCPscan	1.6 days	17.0 TiB	2.3 TiB
eSAIS (SA) + LCPscan	6.6 days	47.5 TiB	3.4 TiB
eSAIS (SA+LCP)	9.9 days	60.3 TiB	7.2 TiB

Discussion. Fig. 2 shows the running times and I/O volumes for LCP array construction on Platform S. In most cases, LCPscan is significantly faster than eSAIS. The type of the file has a noticeable effect on the relative performance. eSAIS probably benefits from the small alphabet size of some of the files. For LCPscan the difference is due to r, the number of irreducible LCP values, which can be seen clearly in the I/O volumes, where the I/O that depends on r is shown separately. Also shown separately is the I/O volume resulting from the text scans during the procedure ProcessBlock, which is the only quadratic component in the I/O volume. Even for the largest 20GiB files, the text scanning volume is only a small fraction of the total I/O volume. In fact, the text scanning volume is almost exactly n/m bytes per byte, where m is the available RAM in bytes, which is 1.5GiB in our case. The file size would have to be at least $100m$ before the quadratic text scanning time would become the dominant component and the asymptotic advantage of eSAIS would really start to show.

The disk space requirement of the implementations is $11n$ bytes for the input and the output plus the peak disk space needed for the intermediate data structures, which is $10n$ bytes for LCPscan and $54n$ bytes for eSAIS; thus the totals are $21n$ bytes and $65n$ bytes.[5] Because of the lack of disk space on Platform S, LCPscan failed for 24GiB files while eSAIS failed already for 8GiB files.

For Platform L we ran experiments up to size 64GiB for all files except wiki, for which we run experiments up to 120GiB. As can be seen in Fig. 3, the results are quite similar to Platform S. With the disk space limitation of eSAIS removed, the speed advantage of LCPscan is even clearer. Table 2 shows the key performance statistics for the largest 120GiB file.

Despite its theoretical disadvantage, LCPscan can be considered to be the more scalable algorithm in practice. For eSAIS to clearly dominate LCPscan, the file size would need to be more than 100 times the size of the RAM and the available disk space would need to be more than 6000 times the RAM size.

Acknowledgements. We thank Timo Bingmann for guiding us in the use of STXXL and for other useful discussions.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms 2(1), 53–86 (2004)

[5] This is a bit more than in the analysis earlier and in [3] because STXXL does not release disk space it has once allocated. In both algorithms, the peak intermediate disk usage occurs at a time when no disk space is needed for the output yet, but STXXL keeps occupying that space even when the output is written to disk.

2. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows-Wheeler transform. J. Discrete Algorithms 18, 22–31 (2013)
3. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and lcp arrays in external memory. In: Proc. ALENEX 2013, pp. 88–102. SIAM (2013)
4. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM J. Experimental Algorithmics 12 (2008)
5. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw., Pract. Exper. 38(6), 589–637 (2008)
6. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)
7. Gog, S., Ohlebusch, E.: Fast and lightweight lcp-array construction algorithms. In: Proc. ALENEX 2011, pp. 25–34. SIAM (2011)
8. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: Pat trees and Pat arrays. In: Frakes, W.B., Baeza-Yates, R. (eds.) Information Retrieval: Data Structures & Algorithms, pp. 66–82. Prentice–Hall (1992)
9. Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. In: Proc. ICABD 2014, pp. 53–60 (2014)
10. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-Ziv parsing in external memory. In: Proc. DCC 2014, pp. 153–162. IEEE CS (2014)
11. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
12. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
13. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
14. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
15. Mäkinen, V.: Compact suffix array — a space efficient full-text index. Fundamenta Informaticae 56(1-2), 191–210 (2003)
16. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
17. Manzini, G.: Two space saving tricks for linear time lcp array computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
18. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
19. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Computers 60(10), 1471–1484 (2011)
20. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag (2013)
21. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
22. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for Longest-Common-Prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
23. Vitter, J.S.: Algorithms and data structures for external memory. Foundations and Trends in Theoretical Computer Science 2(4), 305–474 (2006)