Scalable K-Order LCP Array Construction for Massive Data

Yi Wu[1], Ge Nong[1,2], Wai Hong Chan[3] and Wei Jun Liu[1]

[1]Department of Computer Science, Sun Yat-sen University, Guangzhou, China.
[2]SYSU-CMU Shunde International Joint Research Institute, Shunde, China.
[3]Department of Mathematics and Information Technology, Hong Kong Institute of Education, Hong Kong.
Email: {wu.yi.621@gmail.com, issng@mail.sysu.edu.cn, waihchan@ied.edu.hk, lwjun312@163.com}

Abstract. A new method is proposed to compute the K-order longest common prefix (LCP) array for a size-n input string T, where the maximum LCP of a pair of suffixes of T is truncated to be at most K characters. By employing a finger-printing function, string comparisons in this method are translated into integer comparisons. This method can be applied on the internal memory, external memory and distributed computation models. For each of these models, the total time and space complexities are $\mathcal{O}(n \log K)$ and $\mathcal{O}(n)$, respectively. In particular, this method is scalable for a distributed model of d computing nodes, where the time and space complexities are evenly divided onto each node as $\mathcal{O}((n \log K)/d)$ and $\mathcal{O}(n/d)$, respectively.

1 Introduction

Suffix array (SA) [1] is a succinct data structure for full-text indexing. In conjunction with the longest common prefix array (LCPA), SA can emulate a bottom-up or top-down traversal of the corresponding suffix tree and thus becomes popular for a variety of string processing tasks previously tackled by suffix tree [2].

As far as we know, the most time and space efficient SA and LCPA construction algorithms designed for random access memory (RAM) model are based on the induced sorting principle [3, 4]. However, great improvements in data sampling techniques have created new challenges as the ever-increasing data sets can no longer be processed internally. To close the gap, recently, three novel algorithms [5–7] have been designed to adapt the internal SA construction algorithm SA-IS [3] for sorting suffixes in external memory and achieved remarkable performance gains against the previous state-of-the-art [8]. In particular, the eSAIS algorithm [6] can also produce the LCPA together with the construction of suffix array, where the overhead of time and I/O volume is twice as that of the plain SA construction. The eGSA algorithm [9] enables the simultaneous computation of SA and LCPA for data sets composed of multiple strings of variable-lengths by using multi-way merge-sort, where the time spent in construction is reduced to one-third of eSAIS. The exLCP algorithm [10] is a lightweight LCPA construction algorithm for very large collections of sequences, where the Burrow-Wheeler transform is calculated at the same time to facilitate the computation. Given the suffix array as input, the LCPscan algorithm [11] builds the LCPA from the permuted LCPA and inverse SA computed in advance. Compared with eSAIS, LCPscan requires less disk space and performs better in terms of the running time and I/O efficiency. While these algorithms achieve remarkable time and space performance, however, their designs are quite sophisticated and not trivial to be extended for parallel and distributed models so as to scale the performance by a cluster of computers, for example.

It has been observed from [9] that the average LCPs are typically small for realistic data, especially in genome data sets such as protein and DNA. This motivated us to design a practical algorithm for computing the K-order LCPA of input string T, where the maximum LCP of any two suffixes in T is assumed to be no more than $K << |T|$ (e.g. $K = 8192$).

The contributions of this work are mainly two aspects:

1. Our first contribution is to design and implement a K-order LCPA construction method applicable to typical internal and external memory models, which can build the LCPA in $\mathcal{O}(n \log K)$ time and $\mathcal{O}(n)$ space using the LCP batch querying technique (LCP-BQT) [12] previously proposed for sparse SA construction. This method can be easily applied on both the internal and the external models. The program that we developed for the external memory model is composed of less than 600 lines in C++.
2. The existing parallel construction algorithms are mainly designed for shared memory models such as bulk synchronous parallel and parallel random access machine [13, 14]. Our second contribution is to parallelize the method in a distributed system consisting of d computing nodes.

The rest of the paper is organized as below. Section 2 introduces LCP-BQT and describes the algorithmic framework of the proposed method in the RAM model. Section 3 extends the method to the external memory and distributed models. Finally, we give the performance evaluation in Section 4 and the conclusion in Section 5.

2 *K*-Order LCPA Construction in RAM

2.1 Notation

Consider an input text $T[0, n-1] = T[0]T[1]...T[n-1]$ of n characters from an ordered alphabet Σ. We assume $T[n-1]$ to be a unique character alphabetically smaller than any characters in $T[0, n-2]$ and introduce the following notations for description clarity.

– $\mathsf{pre}(T, i)$ and $\mathsf{suf}(T, i)$: We write $\mathsf{pre}(T, i)$ to be the prefix of T running from $T[0]$ to $T[i]$ and write $\mathsf{suf}(T, i)$ to be the suffix of T running from $T[i]$ to $T[n-1]$.

- SA_T: The suffix array of T, denoted by SA_T, is a permutation of integers $[0, n)$ such that $\mathsf{suf}(T, SA_T[0]) < \mathsf{suf}(T, SA_T[1]) < ... < \mathsf{suf}(T, SA_T[n-1])$ in their lexicographic order.
- $\mathsf{lcp}(i, j)$ and $LCPA_T$: We write $\mathsf{lcp}(i, j)$ to be the LCP length of $\mathsf{suf}(T, i)$ and $\mathsf{suf}(T, j)$. The LCPA of T, denoted by $LCPA_T$, consists of n integers taken from $[0, n)$, where $LCPA_T[i] = \mathsf{lcp}(SA_T[i], SA_T[i-1])$.
- Δ_k: We denote $2^{\log n - k - 1}$ by Δ_k.

2.2 LCP Batch Querying Technique

Given T and a set of b pairs of indices P, LCP-BQT computes $\mathsf{lcp}(i, j)$ for all pairs $(i, j) \in P$ in $\mathcal{O}(n \log b)$ time using $\mathcal{O}(n)$ RAM space. The main idea behind the technique is to find the indices (i_{fin}, j_{fin}) for (i, j) such that $T[i, i_{fin} - 1] = T[j, j_{fin} - 1]$ and $T[i_{fin}] \neq T[j_{fin}]$. To do this, it initially sets P_0 to P and performs a loop of $\log b$ rounds, where the goal of round $k \in [0, \log b)$ is to decide whether $\mathsf{lcp}(i_k, j_k) \leq \Delta_k$ or not for each pair $(i_k, j_k) \in P_k$ and generate P_{k+1} as the input for round $k + 1$ as following: if $T[i_k, i_k + \Delta_k - 1] \neq T[j_k, j_k + \Delta_k - 1]$, then insert (i_k, j_k) into P_{k+1}; otherwise, insert $(i_k + \Delta_k, j_k + \Delta_k)$ into P_{k+1}. The string comparison in concern can be carried out by scanning the two strings from left to right and literally comparing the characters in their lexicographic order. However, this operation takes $O(2^{\log n})$ time at worst and thus becomes a performance bottleneck.

As a solution to relieving the time overhead, the author in [12] introduced a finger-printing function [15] for transforming string comparisons to their integer counterparts that can be done in constant time. The finger-print (FP) of $T[i, j]$, namely $FP[i, j]$, is calculated by the formula $FP[i, j] = \sum_{p=i}^{j} \delta^{j-p} \cdot T[p] \, mod \, L$, in which L is a prime and δ is an integer randomly chosen from $[1, L)$. Obviously, two identical strings always have a common finger-print, while the converse is not true. Fortunately, it has been proved that the error probability of two different strings having a same finger-print can be ignored when L is very large.

Following the above description, the 3-step algorithmic framework for round k is given below.

S1. Scan T rightward to iteratively compute the finger-print of $\mathsf{pre}(0, l)$ by the formula $FP[0, l] = FP[0, l-1] \cdot \delta + T[l] \; mod \; L$ and store $FP[0, l]$ in the hash table if $l \in \{\{i_k - 1\} \cup \{j_k - 1\} \cup \{i_k + \Delta_k - 1\} \cup \{j_k + \Delta_k - 1\}, (i_k, j_k) \in P_k\}$.
S2. For each $l \in \{\{i_k\} \cup \{j_k\}, (i_k, j_k) \in P_k\}$, compute the finger-print of $T[l, l + \Delta_k - 1]$ by the formula $FP[l, l + \Delta_k - 1] = FP[0, l + \Delta_k - 1] - FP[0, l - 1] \cdot \delta^{\Delta_k} \, mod \, L$.
S3. For each pair $(i_k, j_k) \in P_k$, compare $FP[i_k, i_k + \Delta_k - 1]$ with $FP[j_k, j_k + \Delta_k - 1]$. If equal, insert $(i_k + \Delta_k, j_k + \Delta_k)$ into P_{k+1}; otherwise, insert (i_k, j_k) into P_{k+1}.

The following two properties remain invariant during the whole loop.

- At the beginning of round k, $\mathsf{lcp}(i_k, j_k) \leq 2 \cdot \Delta_k$ for each pair $(i_k, j_k) \in P_k$.

– At the end of round k, $\mathsf{lcp}(i_{k+1}, j_{k+1}) \leq \Delta_k$ for each pair $(i_{k+1}, j_{k+1}) \in P_{k+1}$.

After the loop, we have $\mathsf{lcp}(i_{\log b}, j_{\log b}) \leq \frac{n}{b}$ for each pair $(i_{\log b}, j_{\log b}) \in P_{\log b}$, and thus can compute $\mathsf{lcp}(i_{\log b}, j_{\log b})$ in $\mathcal{O}(\frac{n}{b})$ time by literally compare the characters in $\mathsf{suf}(T, i_{\log b})$ and $\mathsf{suf}(T, j_{\log b})$ from left to right. Let $i_{fin} = i_{\log b} + \mathsf{lcp}(i_{\log b}, j_{\log b})$ and $j_{fin} = j_{\log b} + \mathsf{lcp}(i_{\log b}, j_{\log b})$, then i_{fin} and j_{fin} are the position indices to the right side of i and j such that $T[i, i_{fin} - 1] = T[j, j_{fin} - 1]$ and $T[i_{fin}] \neq T[j_{fin}]$.

Lemma 1 *The LCP of any b pairs of suffixes in T can be correctly computed in $\mathcal{O}(n \log b)$ time using $\mathcal{O}(b)$ RAM space with a high probability.*

Proof. The time complexity of LCP-BQT is dominated by the loop of $\log b$ rounds, where each round takes $\mathcal{O}(n)$ time for step 1 to iteratively compute the finger-prints of $\mathsf{pre}(T, l)$, and $\mathcal{O}(b)$ time for steps 2-3 to compute and compare the finger-prints of strings. The space in need is limited to $\mathcal{O}(b)$ words by employing a hash table for storing and retrieving the finger-prints.

2.3 Details

We convert the construction problem to the computation of $\mathsf{lcp}(i, j)$ for all pairs of indices in $\{(SA_T[1], SA_T[0]), (SA_T[2], SA_T[1]) \ldots (SA_T[n], SA_T[n-1])\}$. Parameters and notations listed below are used for presentation clarity, where $i \in [0, n)$ and $k \in [0, \log K)$.

- CP_k and PP_k: integer arrays of size $2n$.
- ICP_k and IPP_k: integer arrays of size $2n$, which are produced by radix-sorting CP_k and PP_k, respectively.
- HT: a hash table for storing and retrieving finger-prints of $\mathsf{pre}(T, i)$, where $i \in \{CP_k[j] \cup PP_k[j], j \in [0, 2n)\}$.

At the very beginning, Algorithm 1 computes CP_0 and PP_0 as following: 1) $CP_0[2i] = SA_T[i] - 1$ and $CP_0[2i + 1] = SA_T[i] + \Delta_0 - 1$; and 2) $PP_0[2i] = SA_T[i - 1] - 1$ and $PP_0[2i + 1] = SA_T[i - 1] + \Delta_0 - 1$. When finishing the computation, it proceeds on to performing a loop of $\log K$ rounds in lines 4-9. A key operation in round k is to iteratively compute the finger-prints of $\mathsf{pre}(T, l)$, where the values are used to calculate $FP[CP_k[2i]+1, CP_k[2i+1]]$ and $FP[PP_k[2i]+1, PP_k[2i+1]]$ for producing CP_{k+1} and PP_{k+1}. More specifically, increase $CP_k[2i]$ and $PP_k[2i]$ by Δ_k and $CP_k[2i+1]$ and $PP_k[2i+1]$ by Δ_{k+1} if the two finger-prints are identical; otherwise, decrease $CP_k[2i+1]$ and $PP_k[2i+1]$ by Δ_{k+1}. Then the algorithm assigns CP_k and PP_k to CP_{k+1} and PP_{k+1}, respectively. After the while-loop, it takes $\mathcal{O}(n)$ time to compute the LCP array in lines 10-11 as $\mathsf{lcp}(CP_{\log K}[2i], PP_{\log K}[2i]) \leq 1$ holds for any $i \in [0, n)$. This leads us to the conclusion stated below.

Lemma 2 *Given T and SA_T, the K-order $LCPA_T$ can be correctly computed in $\mathcal{O}(n \log K)$ time using $\mathcal{O}(n)$ RAM space with a high probability.*

Algorithm 1: Compute K-Order $LCPA_T$ in RAM

```
1  lcpa-ram($T$, $SA_T$, $n$, $K$, $HT$)
2  Scan $SA_T$ rightward to produce $CP_0$ and $PP_0$.
3  Let $k = 0$.
4  while $k < \log K$ do
5     Radix-sort $CP_k$ and $PP_k$ to produce $ICP_k$ and $IPP_k$.
6     For $i \in [0, n)$, scan $T$ rightward to compute the finger-print of $\mathsf{pre}(T, i)$ and let
      $FP[0, i] = HT[i]$ if $i \in \{ICP_k[j] \cup IPP_k[j], j \in [0, 2n)\}$.
7     For $i \in [0, n)$, scan $CP_k$ and $PP_k$ rightward to compute and compare
      $FP[CP_k[2i] + 1, CP_k[2i + 1]]$ and $FP[PP_k[2i] + 1, PP_k[2i + 1]]$ for generating
      $CP_{k+1}$ and $PP_{k+1}$.
8     Let $k = k + 1$ and clear $HT$.
9  end
10 For $i \in [0, n)$, scan $T$, $CP_{\log K}$ and $PP_{\log K}$ rightward to compute
   $\Upsilon_i = \mathsf{lcp}(CP_{\log K}[2i], PP_{\log K}[2i])$.
11 For $i \in [0, n)$, let $\mathsf{lcp}(SA_T[i], SA_T[i-1]) = CP_{\log K}[2i] + \Upsilon_i - SA_T[i]$.
```

Proof. With respect to time complexity, the while-loop spends $\mathcal{O}(n \log K)$ time computing $CP_{\log K}$ and $PP_{\log K}$. On the other respect, it demands $\mathcal{O}(n)$ internal space to maintain HT, CP_k and PP_k.

3 *K*-Order LCPA Construction in Disk

A hash table is employed in Algorithm 1 to store the finger-prints for quick lookups. This is feasible in the RAM model, but not practical when n becomes larger as the table can not be wholly afforded internally anymore. For the purpose, we reformulate lcpa-ram to design a disk-friendly external memory algorithm, called lcpa-disk, where the finger-prints in use are sequentially retrieved from the external memory.

3.1 Preliminaries

Given the RAM capacity M in our external memory model, T and SA_T are evenly partitioned into $d = n/m$ blocks, where each block is of size $m = O(M)$ and thus can be processed as a whole in RAM. We extend CP_k and PP_k previously exploited in lcpa-ram to define their counterparts ECP_k and EPP_k. Both of them consist of $2n$ entries and each entry is a tuple of $\langle idx, pos, fp\rangle$ as described below.

- idx: the index of the entry, where $ECP[i].idx = EPP[i].idx = i$.
- pos: the ending position of $\mathsf{pre}(0, pos)$.
- fp: the finger-print of $\mathsf{pre}(0, pos)$.

For $i \in [0, n)$, the fp values of $ECP_k[2i]$ and $ECP_k[2i + 1]$ are employed to figure out $FP[ECP_k[2i].pos + 1, ECP_k[2i + 1].pos]$ by the formula $ECP_k[2i + 1].fp - ECP_k[2i].fp \cdot \delta^{\Delta_k} \, mod \, L$, where $FP[EPP_k[2i].pos + 1, EPP_k[2i + 1].pos]$ can be obtained in a similar way.

3.2 Details

Different from Algorithm 1, lcpa-disk performs two runs of external memory sort to arrange fixed-size entries by their integer keys of $\log n$ bits during each round. In line 5, entries of ECP_k and EPP_k are sorted by *pos* to form $IECP_k$ and $IEPP_k$ for iteratively computing the required finger-prints, and sorted back by *idx* to compute ECP_{k+1} and EPP_{k+1} in lines 7-8. Given $M = \Omega(nW)^{0.5}$, where W is the size of I/O buffers large enough to amortize the overhead of accesses to disks, each run can be done in $\mathcal{O}(n)$ time and space by adopting a multi-way external memory radix-sort of two passes, of which the first pass sorts the lowest $0.5 \log n$ bits and the second pass sorts the highest $0.5 \log n$ bits. Thus we come to the following statement.

Lemma 3 *Given T and SA_T, the K-order $LCPA_T$ can be correctly computed in $O(n \log K)$ time using $O(n)$ disk space with a high probability.*

Algorithm 2: Compute K-Order $LCPA_T$ in Disk

```
1  lcpa-disk($T$, $SA_T$, $n$, $K$)
2  Scan $SA_T$ rightward to produce $ECP_0$ and $EPP_0$.
3  Let $k = 0$.
4  while $k < \log K$ do
5      Radix-sort $ECP_k$ and $EPP_k$ by $pos$ to produce $IECP_k$ and $IEPP_k$.
6      For $i \in [0, n)$ and $j \in [0, 2n)$, scan $T$ rightward to iteratively compute the
       finger-print of $\mathsf{pre}(T, i)$ and assign $FP[0, i]$ to $IECP_k[j].fp$ or $IEPP_k[j].fp$ if
       $IECP_k[j].pos = i$ or $IEPP_k[j].pos = i$.
7      Radix-sort $IECP_k$ and $IEPP_k$ by $idx$ to reproduce $ECP_k$ and $EPP_k$.
8      For $i \in [0, n)$, scan $ECP_k$ and $EPP_k$ rightward to compute and compare each
       pair of $(FP[ECP_k[2i].pos + 1, ECP_k[2i+1].pos], FP[EPP_k[2i].pos +$
       $1, EPP_k[2i+1].pos])$ for generating $ECP_{k+1}$ and $EPP_{k+1}$.
9      Let $k = k + 1$.
10 end
11 For $i \in [0, n)$, scan $T$, $ECP_{\log K}$ and $EPP_{\log K}$ rightward to compute
   $\Upsilon_i = \mathsf{lcp}(ECP_{\log K}[2i].pos, EPP_{\log K}[2i].pos)$.
12 For $i \in [0, n)$, let $\mathsf{lcp}(SA_T[i], SA_T[i-1]) = ECP_{\log K}[2i].pos + \Upsilon_i - SA_T[i]$.
```

3.3 Optimization

The while-loop is time-consuming in lcpa-disk, therefore we adapt the algorithm to generate $ECP_{k+2} \backslash EPP_{k+2}$ directly from $ECP_k \backslash EPP_k$ for reducing the loop rounds by half, where the procedure in lines 8-9 is revised as below.

S1. Compute and compare the finger prints of $T[ECP_k[2i].pos + 1, ECP_k[2i + 1].pos]$ and $T[EPP_k[2i].pos + 1, EPP_k[2i + 1].pos]$. If equal, perform S2; otherwise, go to S3.

S2. Compute and compare the finger-prints of $T[ECP_k[2i{+}1].pos{+}1, ECP_k[2i{+}1].pos{+}\Delta_{k+1}]$ and $T[EPP_k[2i{+}1].pos{+}1, EPP_k[2i{+}1].pos{+}\Delta_{k+1}]$. If equal, increase $ECP_k[2i].pos$ and $EPP_k[2i].pos$ by $\Delta_k{+}\Delta_{k+1}$ and $ECP_k[2i{+}1].pos$ and $EPP_k[2i{+}1].pos$ by $\Delta_{k+1}{+}\Delta_{k+2}$; otherwise, increase $ECP_k[2i].pos$ and $EPP_k[2i].pos$ by Δ_k and $ECP_k[2i+1].pos$ and $EPP_k[2i+1].pos$ by Δ_{k+2}.

S3. Compute and compare the finger-prints of $T[ECP_k[2i].pos{+}1, ECP_k[2i].pos{+}\Delta_{k+1}]$ and $T[EPP_k[2i].pos + 1, EPP_k[2i].pos + \Delta_{k+1}]$. If equal, increase $ECP_k[2i].pos$ and $EPP_k[2i].pos$ by Δ_{k+1} and decrease $ECP_k[2i+1].pos$ and $EPP_k[2i+1].pos$ by Δ_{k+2}; otherwise, decrease $ECP_k[2i+1].pos$ and $EPP_k[2i+1].pos$ by $\Delta_{k+1} + \Delta_{k+2}$.

S4. Let $ECP_{k+2} = ECP_k$ and $EPP_{k+2} = EPP_k$.

S5. Let $k = k + 2$.

Notice that the required finger-prints in S1-S3 can be obtained following the same way described in lcpa-disk. As shown in Section 4, the revised algorithm, namely lcpa-disk-m, achieves a substantial improvement against lcpa-disk.

3.4 Discussion

In this part, we demonstrate how to parallelize Algorithm 2 in a distributed system consisting of d computing nodes $\{N_0, N_1, ...N_{d-1}\}$. All nodes are interconnected with each other by a gigabit switch operating in full duplex mode where the internal and external memory capacities of each node are M and E, respectively. For $i \in [0, d)$ and $j \in [0, e)$, we evenly partition T and SA_T into $d = n/e$ blocks of size $e = \mathcal{O}(E)$ and allocate them among the computing nodes in a manner of one block per node as below.

- T_i: the i-th block of T on node N_i, where $T_i[j] = T[ie + j]$.
- SA_{T_i}: the i-th block of SA_T on node N_i, where $SA_{T_i}[j] = SA_T[ie + j]$.

Recall that lcpa-disk computes $ECP_{\log K}$ and $EPP_{\log K}$ via two runs of radix-sort in external memory. This is emulated in Algorithm 3 by using a set of sending/receving buffers for data exchange among the computing nodes. After the while-loop, the LCP array of $T[ie, ie + e)$ can be figured out in linear time according to $ECP_{i,\log K}$, $EPP_{i,\log K}$ and SA_{T_i} residing on N_i. Finally, we can simply concatenate the local result on each node to form the global one.

Lemma 4 *Given T and SA_T, the K-order $LCPA_T$ can be correctly computed in $\mathcal{O}(\frac{n}{d}\log K)$ time using $\mathcal{O}(\frac{n}{d})$ disk space on each computing node with a high probability.*

Proof. Algorithm 3 performs radix-sorts among the computing nodes, where the overhead of time and space for data transmission sums up to $\mathcal{O}(e \log K)$ during the whole loop. This is because each node sends and receives $\mathcal{O}(e)$ entries per round.

Algorithm 3: Compute K-Order $LCPA_T[ie, ie+e)$ on N_i

```
1  lcpa-ds($T_i$, $SA_{T_i}$, $e$, $K$)
2  Scan $T_i$ rightward to compute $ECP_{i,0}$ and $EPP_{i,0}$.
3  Let $k = 0$.
4  For $j \in [0, d)$, create sending buffers $SB1_{i,j}$ and $SB2_{i,j}$.
5  Create receiving buffers $RB1_i$ and $RB2_i$.
6  while $k < \log K$ do
7    Scan $SA_{T_i}$ rightward to compute $ECP_{i,k}$ and $EPP_{i,k}$.
8    For $j \in [0, d)$ and $p \in [0, 2e)$, scan $ECP_{i,k}$ and $EPP_{i,k}$ rightward to cache
       $ECP_{i,k}[p]$ in $SB1_{i,j}$ or $EPP_{i,k}[p]$ in $SB2_{i,j}$ if $ECP_{i,k}[p].pos$ or $EPP_{i,k}[p].pos$
       belongs to $[je, je+e)$.
9    For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to $N_j$.
10   For $j \in [0, d)$, cache entries from $SB1_{i,j}$ and $SB2_{i,j}$ in $RB1_i$ and $RB2_i$.
11   Radix-sort entries in $RB1_i$ and $RB2_i$ by $pos$ to produce $IECP_{i,k}$ and
       $IEPP_{i,k}$.
12   For $p \in [0, e)$ and $q \in [0, 2e)$, scan $T_i$ rightward to iteratively compute the
       finger-print of $\mathsf{pre}(T, ie+p)$ and assign $FP[0, ie+p]$ to $IECP_{i,k}[q].fp$ or
       $IEPP_{i,k}[q].fp$ if $IECP_{i,k}[q].pos = ie+p$ or $IEPP_{i,k}[q].pos = ie+p$.
13   For $j \in [0, d)$ and $p \in [0, 2e)$, scan $IECP_i$ and $IEPP_i$ rightward to cache
       $IECP_{i,k}[p]$ in $SB1_{i,j}$ or $IEPP_{i,k}[p]$ in $SB2_{i,j}$ if $IECP_{i,k}[p].pos$ or
       $IEPP_{i,k}[p].pos$ belongs to $[je, je+e)$.
14   For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to $N_j$.
15   For $j \in [0, d)$, cache entries from $SB1_{i,j}$ and $SB2_{i,j}$ in $RB1_i$ and $RB2_i$.
16   Radix-sort entries in $RB1_i$ and $RB2_i$ by $idx$ to reproduce $ECP_{i,k}$ and $EPP_{i,k}$.
17   For $p \in [0, e)$, scan $ECP_{i,k}$ and $EPP_{i,k}$ rightward to compute and compare
       the finger-prints of $T[ECP_{i,k}[2p].pos+1, ECP_{i,k}[2p+1].pos]$ and
       $T[EPP_{i,k}[2p].pos+1, EPP_{i,k}[2p+1].pos]$ for generating $ECP_{i,k+1}$ and
       $EPP_{i,k+1}$.
18   Let $k = k+1$.
19 end
20 For $p \in [0, e)$, scan $T_i$, $ECP_{i,\log K}$ and $EPP_{i,\log K}$ rightward to literally compute
     $\Upsilon_{i,p} = \mathsf{lcp}(ECP_{i,\log K}[2p].pos, EPP_{i,\log K}[2p].pos)$.
21 For $p \in [0, e)$, let $\mathsf{lcp}(SA_{T_i}[p], SA_{T,i}[p-1]) = ECP_{i,\log K}.pos + \Upsilon_{i,p} - SA_{T_i}[p]$.
```

4 Experimental Results

The performance of lcpa-disk and lcpa-disk-m is evaluated on a real data set collected from the web site http://download.wikimedia.org/enwiki/. We conduct the experiments on a laptop equipped with Intel Cuo i5 4200M CPU, 4 GiB RAM and a 500 GiB hard disk of 5400 rpm.

Instead of to develop a radix sorter specific for our purpose, we use STXXL [16] to perform the external memory sorts in our programs. STXXL is a C++ STL library designed for efficient computation in external memory, and freely available at http://stxxl.sourceforge.net/. Benefit from the powerful priority queues provided by STXXL, the codes for lcpa-disk and lcpa-disk-m are only 400 and 600 lines, respectively.

Each result in Table 1 is a mean of two runs of the programs, where the running time and peak disk use are monitored by the shell command "time" and "stxxl::block_manager", respectively. As can be seen, the running time of lcpa-disk-m is significantly faster than that of lcpa-disk in all cases, where the loop rounds of them are 7 and 13, respectively. We also observed that the processing time of each character per round varies from 1.99 μs to 4.18 μs as the size of corpora increases from 200 MiB to 2 GiB. This sub-linear phenomenon is partly due to the non-linear external memory sorts conducted by STXXL.

As reported in [11], the disk space requirements for eSAIS and LCPscan are $65n$ and $21n$, respectively, while the peak disk use of the implementation for lcpa-disk-m rises up to $61n$ for processing the 2 GiB file. This indicates that lcpa-disk-m is more space demanding in comparison with the current best methods. However, the proposed method is easy to be implemented and strongly scalable for parallel and distributed models where the communication overhead on each node is balanced as $\mathcal{O}(\frac{n}{d})$.

Table 1. Loop rounds, mean running time in $\mu s/ch$ per round and peak disk use in bytes per character, where $K = 8192$.

Size	lcpa-disk			lcpa-disk-m		
(MiB)	Round	Time	Disk	Round	Time	Disk
200	13	1.71	30.94	7	1.99	47.36
400	13	1.70	31.50	7	2.01	47.36
600	13	1.79	31.58	7	2.27	47.36
800	13	1.86	31.52	7	2.65	47.38
1024	13	3.10	31.52	7	3.30	47.38
2048	13	3.16	46.04	7	4.18	60.48

5 Conclusion

We present in this paper a practical K-order LCPA construction method that can be easily applied on both the internal memory and the external memory models. The program for lcpa-disk-m is less than 600 lines when using STXXL to implement the external sorts. We also show that the proposed method is straightforward to be extended for running on a typical distributed system of a cluster of d computing nodes, where the time and space complexities are evenly divided onto each node as $\mathcal{O}(\frac{n}{d}\log K)$ and $\mathcal{O}(\frac{n}{d})$, respectively. The proposed algorithms are simple in design and universal for the RAM, disk and distributed models. Its implementations on all these models are not difficult and easily to be deployed. A cluster of computers in a local area network are commonly available in practice, but there is currently lack of scalable LCPA construction algorithms for such a distributed model, in this sense, our algorithms provide a candidate solution to meet this demand. For further work, we now attempt to implement the distributed algorithm on a cluster of computers and speed up the computation of finger-prints by GPU.

References

1. U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
2. M. Abouelhodaa, S. Kurtzb, and E. Ohlebuscha. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, 2(1):53–86, November 2004.
3. G. Nong, S. Zhang, and W. H. Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*, 60(10):1471–1484, October 2011.
4. J. Fischer. Inducing the LCP-Array. In *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. 2011.
5. G. Nong, W. H. Chan, S. Q. Hu, and Y. Wu. Induced Sorting Suffixes in External Memory. *ACM Transactions on Information Systems*, 33(3):12:1–12:15, March 2015.
6. 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*, pages 88–102.
7. G. Nong, W. H. Chan, S. Zhang, and X. F. Guan. Suffix Array Construction in External Memory Using D-Critical Substrings. *ACM Transactions on Information Systems*, 32(1):1:1–1:15, January 2014.
8. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better External Memory Suffix Array Construction. *ACM Journal of Experimental Algorithmics*, 12:3.4:1–3.4:24, August 2008.
9. F. Louza, G. Telles, and C. Ciferri. External Memory Generalized Suffix and LCP Arrays Construction. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching*, pages 201–210, Bad Herrenalb, Germany, June 2013.
10. M. Bauer, A. C. G. Rosone, and M. Sciortino. Lightweight LCP Construction for Next-Generation Sequencing Datasets. In *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics*, Ljubljana, Slovenia, September 2012.
11. J. Kärkkäinen and D. Kempa. LCP Array Construction in External Memory. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 412–423, Copenhagen, Denmark, June 2014.

12. P. Bille, I. L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse Suffix Tree Construction in Small Space. pages 148–159, July 2013.
13. J. Shun. Fast Parallel Computation of Longest Common Prefixes. In *Proceedings of the 40th International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 387–398, New Orleans, LA, November 2014.
14. 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*, pages 197–206, New York ,USA, August 2013.
15. R. Karp and M. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.
16. R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL Data Sets. *Software: Practice and Experience*, 38(6):589–637, 2008.