Scalable finite K-order LCP Array Construction for Large-Scale Data Sets

Yi Wu and Ge Nong*

Department of Computer Science
Sun Yat-sen University, Guangzhou, China
*Corresponding Author, Email: issng@mail.sysu.edu.cn

Abstract. We present in this paper a novel method to compute the finite K-order longest common prefix array for an input string T of size n. Theoretical analysis indicates that the LCP array can be constructed using $\mathcal{O}(n \log K)$ time and $\mathcal{O}(n)$ space in external memory. The method can also be deployed in a distributed system composed of d computing nodes, where the time complexity is linear proportional to n when $d = \mathcal{O}(\log K)$.

1 Introduction

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

As far as we know, the most time- and space-efficient SA and LCPA construction algorithms in random access memory (RAM) model are based on the induced sorting principle [3, 4]. However, great improvements in data sampling techniques have brought about new challenges as the ever-increasing data sets can no longer be processed internally. To meet the gap, recently, three novel algorithms [**?**, 5, 6] have been designed to adapt the internal SA construction algorithm SA-IS [3] for sorting suffixes in external memory (EM). As reported, all the three algorithms achieved a remarkable performance gain in terms of time, space and I/O efficiency against the state-of-the-arts [7].

As for constructing LCPA in external memory, [**?**] reformulated the RAM method formerly presented in [4] and made it applicable to running externally. Besides, [8] proposed a lightweight construction algorithm for very large collections of sequences, where the Burrow-Wheeler Transform (BWT) must be calculated simultaneously to facilitate the computation. In addition, [9] adopted the technique of multi-way merge-sort to produce generalized suffix and LCP arrays for a data set composed of multiple strings of variable-lengths. Moreover, [10] reported a method that builds the array according to the permuted LCPA and inverse SA obtained in advance.

It has been observed from [9] that the average longest common prefix (LCP) of corpus are typically small, especially for genome data sets, such as protein

and DNA. This enlightened us to design a practical method for computing the finite K-order LCP array of input string T, where the LCP length of any two suffixes in T is assumed to be not greater than $K << |T|$ (eg. $K = 8192$).

Contributions in this paper:

1. Our first work is to design and implement a K-order LCPA construction method applicable in typical RAM and EM models, which is capable of accomplishing the construction in $\mathcal{O}(n \log K)$ time and $\mathcal{O}(n)$ space using the LCP batch querying technique (LCP-BQT) [11] previously proposed for sparse SA construction.
2. The existing parallel algorithms for LCPA construction can only apply to shared-memory models like Bulk Synchronous Parallel (BSP) and Parallel Random Access Machine (PRAM) [12,13]. Different from the prior arts, our second work is to parallelize the proposed method in a distributed system consisting of d computing nodes.

2 K-order LCPA Construction in RAM

2.1 Basic Notations

Consider an input text $T[0, n-1] = T[0]T[1]...T[n-1]$ of size n drawn 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 LCP array of T, denoted as $LCPA_T$, consists of n integers taken from $[0, n)$, where $LCPA_T[i] = \mathsf{lcp}(SA_T[i], SA_T[i-1])$.

Additionally, we use Δ_k to denote $2^{\log n - k - 1}$ in the rest of the article.

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 rightward for 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, [11] introduced a finger-printing function [14] for transforming string comparisons to their integer counterparts that can be finished 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 and δ are very large.

Following the above description, we show the 3-step algorithmic framework for round k as 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 $FP[i_k, i_k + \Delta_k - 1]$ and $FP[j_k, j_k + \Delta_k - 1]$. 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, $lcp(i_k, j_k) \leq \frac{1}{2} \cdot \Delta_k$ for each pair $(i_k, j_k) \in P_k$.
- At the end of round k, $lcp(i_{k+1}, j_{k+1}) \leq \Delta_k$ for each pair $(i_{k+1}, j_{k+1}) \in P_{k+1}$.

After the loop, we have $lcp(i_{\log b}, j_{\log b}) \leq n/b$ for each pair $(i_{\log b}, j_{\log b}) \in P_{\log b}$, and can compute $lcp(i_{\log b}, j_{\log b})$ in $\mathcal{O}(n/b)$ 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 $i_{fin} = i_{\log b} + lcp(i_{\log b}, j_{\log b})$ and $j_{fin} = j_{\log b} + lcp(i_{\log b}, j_{\log b})$, i_{fin} and j_{fin} point to the right side of i and j such that $T[i, i_{fin} - 1] = T[j, j_{fin} - 1]$ and $T[i_{fin}] \neq T[j_{fin}]$. Thus we have $lcp(i, j) = i_{fin} - i$ for each pair $(i, j) \in P$.

From the above description, we can come to the following theorem.

Theorem 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 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 strings by their finger-prints. The space in need is limited to $O(b)$ words by employing a hash table to store and retrieve the finger-prints.

2.3 Implementation in RAM

Following the ideas in LCP-BQT, Algorithm 1 converts the K-order LCP array construction problem to the computation of $lcp(i, j)$ for all pairs of indices in $\{(SA_T[1], SA_T[0]), (SA_T[2], SA_T[1]) \dots (SA_T[n], SA_T[n-1])\}$. The parameters and notations listed below are introduced for presentation clarity, where $i \in [0, n)$ and $k \in [0, \log K)$.

- CP_k and PP_k: arrays of size $2n$. $T[CP_k[2i] + 1, CP_k[2i + 1]]$ is compared with $T[PP_k[2i]+1, PP_k[2i+1]]$ in round k by their finger-prints to generate CP_{k+1} and PP_{k+1}, for $i \in [0, n)$.
- ICP_k and IPP_k: 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 $\mathsf{pre}(\mathsf{T}, \mathsf{i})$, where $i \in \{CP_k[j] \cup PP_k[j], j \in [0, 2n)\}$.

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 $\Upsilon_i = lcp(CP_{\log K}[2i], PP_{\log K}[2i])$.
11 For $i \in [0, n)$, let $lcp(SA_T[i], SA_T[i-1]) = CP_{\log K}[2i] + \Upsilon_i - SA_T[i]$.

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, where the key operation of round k is to compute the finger-prints of $\mathsf{pre}(T, l)$ for $i \in \{CP_k[j] \cup PP_k[j], j \in [0, 2n)\}$. The algorithm uses the values to compute $FP[CP_k[2i]+1, CP_k[2i+1]]$ and $FP[PP_k[2i]+1, PP_k[2i+1]]$, and thus to compare them for producing CP_{k+1} and PP_{k+1} at the end of round k. Specifically, it increase $CP_k[2i]$, $CP_k[2i + 1]$, $PP_{k+1}[2i]$ and $PP_{k+1}[2i + 1]$ by Δ_{k+1} if the two finger-prints are identical; otherwise, it decreases $CP_{k+1}[2i + 1]$ and $PP_{k+1}[2i+1]$ by Δ_{k+1}. Then ECP_{k+1} and EPP_{k+1} are set to ECP_k and EPP_k.

After the while-loop, it takes $\mathcal{O}(n)$ to compute the LCP array in lines 10-11 as $lcp(CP_{\log K}[2i], PP_{\log K}[2i]) \leq 1$ holds for any $i \in [0, n)$. This leads us to the following statement.

Theorem 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.*

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 and other data structures.

3 *K*-Order LCPA Construction in EM

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

3.1 Preliminaries

In our EM 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, where 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 $\mathsf{pre}(0, pos)$, where $FP[ECP_k[2i].pos+1, ECP_k[2i+1].pos]$ is compared with $FP[EPP_k[2i].pos+1, EPP_k.pos[2i+1]]$ to generate ECP_{k+1} and EPP_{k+1} in round k.
- 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 $ECP_k[2i + 1].fp - ECP_k[2i].fp \cdot \delta^{\Delta_k} \bmod L$, 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-em* performs two runs of external memory sort to arrange fixed-size items by their integer keys of $\log n$ bits during a loop 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 the RAM capacity

$M = \Omega(nW)^{0.5}$, where W is the size of I/O buffers large enough to amortize the overhead of accesses to disks, each run of sort can be done in $\mathcal{O}(n)$ time and space by adopting a multi-way external memory radix-sort of two passes, where the first pass sorts the lowest $0.5 \log n$ bits and the second pass sorts the highest $0.5 \log n$ bits. This leads us to the following conclusion.

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

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

1 *lcpa-em*(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 form $IECP_k$ 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 set $IECP_k[j].fp$ or $IEPP_k[j].fp$ to $FP[0, i]$ if $IECP_k[j].pos = i$ or $IEPP_k[j].pos = i$.
7 Radix-sort $IECP_k$ and $IEPP_k$ back to ECP_k and EPP_k by idx.
8 For $i \in [0, n)$, scan ECP_k and EPP_k rightward to compute each pair of $(FP[ECP_k[2i].pos + 1, ECP_k[2i + 1].pos], FP[EPP_k[2i].pos + 1, EPP_k[2i + 1].pos])$ and compare them to generate 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 literally compute $\Delta_i = lcp(ECP_{\log K}[2i].pos, EPP_{\log K}[2i].pos)$.
12 For $i \in [0, n)$, let $lcp(SA_T[i], SA_T[i-1]) = ECP_{\log K}[2i].pos + \Delta_i - SA_T[i]$.

3.3 Optimization

As the while-loop is time-consuming in lcpa-em, we adapt lines 8-9 to generate ECP_{k+2}/EPP_{k+2} directly from ECP_k/EPP_k, where Δ_{k+1} and Δ_{k+2} are set to $2^{\log n-(k+1)-1}$ and $2^{\log n-(k+2)-1}$, respectively.

S1. 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 step 2; otherwise, go to step 3.
S2. Compare the finger-prints of $T[ECP_k[2i+1].pos+1, ECP_k[2i+1].pos+\Delta_{k+2}]$ and $T[EPP_k[2i + 1].pos + 1, EPP_k[2i + 1].pos + \Delta_{k+2}]$. If equal, increase $ECP_k[2i].pos$, $ECP_k[2i + 1].pos$, $EPP_k[2i].pos$ and $EPP_k[2i + 1].pos$ by $\Delta_{k+1} + \Delta_{k+2}$; otherwise, increase them by Δ_{k+1}.
S3. Compare the finger-prints of $T[ECP_k[2i].pos+1, ECP_k[2i].pos+\Delta_{k+2}]$ and $T[EPP_k[2i].pos+1, EPP_k[2i].pos+\Delta_{k+2}]$. If equal, increase $ECP_k[2i].pos$, $ECP_k[2i + 1].pos$, $EPP_k[2i].pos$ and $EPP_k[2i + 1].pos$ by Δ_{k+2}.

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

In this way, the number of loop round is reduced to half where the required finger-prints can be obtained using the approach described in lcpa-em. As shown in Section 4, the revised algorithm, namely lcpa-em-m, achieves substantial improvements against the prototype.

3.4 Discussions

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 and the internal and external memory capacities of each node are M and E, respectively.

For $i \in [0, d-1]$ and $j \in [0, e-1]$, 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 residing on node N_i, where $T_i[j] = T[ie + j]$.
- SA_{T_i}: the i-th block of SA_T residing on node N_i, where $SA_{T_i}[j] = SA_T[ie + j]$.

Recall that *lcpa-em* computes $ECP_{\log K}$ and $EPP_{\log K}$ through 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, and we can simply concatenate the local result on each node to form the global one.

Theorem 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 $\mathcal{O}(\frac{n}{d} \log K)$ EM space on each computing node with a high probability.*

Proof. Algorithm 3 introduces a communication phase to perform radix-sorts among the computing nodes, where the overhead 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 of constant size in lines 9-10 and 14-15.

4 Experimental Results

We evaluate the performance of *lcpa-em* and *lcpa-em-m* on a real data set collected from the web site http://download.wikimedia.org/enwiki/. The simulation experiments are conducted on a platform equipped with Intel Cuo i5 4200M CPU, 4GiB RAM and a 500GiB hard disk. We use a third-party C++ STL library STXXL [15] to perform external memory sorts. Benefit from the simplicity

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

lcpa-ds(T_i, SA_{T_i}, e, K)
Scan T_i rightward to compute $ECP_{i,0}$ and $EPP_{i,0}$.
Let $k = 0$.
For $j \in [0, d)$, create sending buffers $SB1_{i,j}$ and $SB2_{i,j}$.
Create receiving buffers $RB1_i$ and $RB2_i$.
while $k < \log K$ **do**
  Scan SA_{T_i} rightward to compute $ECP_{i,k}$ and $EPP_{i,k}$.
  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 $EPP_{i,k}[p].pos$ belong to $[je, je+e)$.
  For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to N_j.
  For $j \in [0, d)$, cache entries from $SB1_{i,j}$ and $SB2_{i,j}$ in $RB1_i$ and $RB2_i$.
  Radix-sort entries in $RB1_i$ and $RB2_i$ by pos to produce $IECP_{i,k}$ and $IEPP_{i,k}$.
  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 $IEPP_{i,k}[q].pos = ie+p$.
  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 $IEPP_{i,k}[p].pos$ belong to $[je, je+e)$.
  For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to N_j.
  For $j \in [0, d)$, cache entries from $SB1_{i,j}$ and $SB2_{i,j}$ in $RB1_i$ and $RB2_i$.
  Radix-sort entries in $RB1_i$ and $RB2_i$ by idx back to $ECP_{i,k}$ and $EPP_{i,k}$.
  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 $EPP_{i,k+1}$.
  Let $k = k + 1$.
end
For $p \in [0, e)$, scan T_i, $ECP_{i,\log K}$ and $EPP_{i,\log K}$ rightward to literally compute $\Delta_{i,p} = lcp(ECP_{i,\log K}[2p].pos, EPP_{i,\log K}[2p].pos)$.
For $p \in [0, e)$, let $lcp(SA_{T_i}[p], SA_{T,i}[p-1]) = ECP_{i,\log K}.pos + \Delta_{i,p} - SA_{T_i}[p]$.

of STXXL, the programs of two algorithms can be implemented in only 400 and 600 lines, respectively.

As shown in Table 1, the running time of lcpa-em-m is significantly faster than that of lcpa-em in all cases. We also observed that the average processing time of each character per round varies from $1.99\mu s$ to $4.18\mu s$ as the size of corpora increases from $200M$ to $2G$. This sub-linear phenomenon is due to the non-linear external memory sorts adopted by STXXL.

Table 1. Round number of the while-loop and running time in $\mu s/ch$ per round, where $K = 8192$.

Size	lcpa-em		lcpa-em-m	
	Round	Running time	Round	Running time
200M	13	3.18	7	1.99
400M	13	3.17	7	2.01
600M	13	3.32	7	2.27
800M	13	3.45	7	2.65
1G	13	5.76	7	3.30
2G	13	x	7	4.18

5 Conclusions

We present in this paper a practical K-order $LCPA$ construction method that can be adopted in internal and external memory. The code of the EM program takes at most 600 lines when using STXXL to implement sorting in external memory. We also show that the proposed method is extensible to be deployed in a typical distributed system composed of d computing nodes, where the time and space requirements for each node are $O(\frac{n}{d} \log K)$ time and $O(\frac{n}{d})$, respectively.

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. 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.
7. 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.
8. 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.
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. 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.
11. 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.
12. 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.
13. 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.
14. R. Karp and M. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.
15. R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL Data Sets. *Software: Practice and Experience*, 38(6):589–637, 2008.