Scalable K-Order LCP Array Construction for Massive Data

Yi Wu[1], Ling Bo Han[1], Wai Hong Chan[2,*] and Ge Nong[1,3,*]

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

Abstract. Given a size-n input text T and its suffix array, a new method is proposed to compute the K-order longest common prefix (LCP) array for T, where the maximum LCP of two suffixes of T is truncated to be at most K. The method uses a fingerprint function to convert the comparison of two variable-length strings to the comparison of two fixed-size integers, where each integer is the fingerprint of a string. This method can be easily applied on the internal memory, external memory and distributed models, and the time and space complexities are universal as $\mathcal{O}(n \log K)$ and $\mathcal{O}(n)$, respectively. This method is scalable for a typical distributed model of a cluster 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. An experimental study has been conducted for performance evaluation of this method running on the external memory and distributed models. A cluster of computers in a local area network are commonly available in practice, but there is currently a lack of scalable LCP-array construction algorithms for such a distributed model. Our algorithms provide a candidate solution to meet the demand.

1 Introduction

Suffix array (SA) [1] is a fundamental data structure for many string processing applications. An SA together with the longest common prefix (LCP) array can be applied to emulating a bottom-up or top-down traversal of the corresponding suffix tree [2]. The SA construction problem has been intensively investigated in the past two decades, resulting in various algorithms proposed for the internal memory model [3–9], see [10] for a comprehensive survey. Among them, the SA-IS algorithm [9] is fast in practice and has a linear time complexity in the worst case, and it has been recently extended by Fischer, Bingmann and Osipov [11,12] for computing both the suffix and LCP arrays simultaneously on the internal memory and the external memory models.

The LCP-array is an important auxiliary data structure for using an SA to replace a suffix tree, hence its construction has attracted much research attention

since its appearance [1]. Kasai et al. proposed the first linear time LCP-array construction method, which builds the array very fast using T, the SA and the inverse SA [13]. Manzini et al. adapted the method to reduce the space requirement at the cost of an increase on the running time [14]. Kärkkäinen et al. presented an alternative to construct the Permuted LCP-array and then transform it to the LCP-array in linear time [15]. As recognized by so far [11,12], the most time and space efficient algorithms for simultaneously constructing the suffix and LCP arrays on both the internal memory and the external memory models are based on the induced sorting principle.

However, emerging applications of ever-increasing massive data have created new challenges for constructing the suffix and LCP arrays time and space efficiently. To attempt this problem, recently, three novel SA construction algorithms eSAIS [12], EM-SA-DS [16] and EM-SA-IS [17] have been designed to adapt SA-IS for sorting suffixes in external memory and achieved remarkable performance gains against the previous state-of-the-art [18]. In particular, the eSAIS algorithm [12] can produce the LCP-array together with the construction of SA, where the overheads of time and I/O volume are around twice as that of the plain SA construction. Given the SA, the LCPscan algorithm [19] can build the LCP-array from the permuted LCP-array and the inverse SA computed in advance. Compared with eSAIS, LCPscan requires less disk space, running time and I/O volume. By relaxing T from a single string to a collection of strings, algorithms with further performance improvements can be designed for constructing the suffix and LCP arrays, e.g. the algorithms eGSA [20] and exLCP [21]. The eGSA algorithm can compute both the suffix and LCP arrays for T consisting of many variable-length strings by using a multi-way merge-sort, and the experimental results show that it can run much faster than eSAIS. The exLCP algorithm is a lightweight LCP-array construction algorithm for a collection of sequences, where the Burrows-Wheeler transform is calculated at the same time to facilitate the computation. While these algorithms achieve remarkable time and space performance, their designs are quite sophisticated due to the poor locality of memory accesses, and thus not trivial to be extended for parallel and distributed models to scale the performance by a cluster of computers, for example.

It has been observed from [20] that the average LCPs are typically small for realistic data, or the string T consists of many short strings so that the LCP of any two short strings is upper bounded by the longest size of a short string (e.g. T is a genome database). In these cases, the original full-order LCP-array is actually a K-order LCP-array, where K is the maximum LCP value which is typically far less than the size of T. This motivated us to design a practical algorithm for computing the K-order LCP-array of T, which is defined as: given any two neighboring suffixes in the SA of T, the K-order LCP of them is the longest common prefix of their first K characters, where K is far less than $|T|$ (e.g., $K = 2^{13}$ and $|T|$ is usually beyond 2^{30} for massive data). The main contributions of this work are the following two aspects.

The first is to design and implement a K-order LCP-array construction method applicable to the typical internal and external memory models, which can build the LCP-array in $\mathcal{O}(n \log K)$ time and $\mathcal{O}(n)$ space using the LCP batch querying technique (LCP-BQT) [22] previously proposed for the sparse SA construction. This method can be easily applied on both the internal and external models. The program we developed for the external memory model is composed of less than 600 lines in C++.

The second is to parallelize our new K-order LCP-array construction method in a distributed system consisting of a cluster of d computing nodes. The existing parallel LCP-array construction algorithms are mainly designed for shared memory models such as bulk synchronous parallel and parallel random access machine [23, 24]. Our distributed model of clustered computers is popular in nowadays computation environments, and hence our method is easier to be deployed in practice.

The rest of this paper is organized as below. Section 2 introduces LCP-BQT and describes the algorithmic framework of our proposed method in random access memory (RAM). Section 3 and 4 extend the method to the external memory and distributed models, respectively. Finally, we give the performance evaluation in Section 5 and the conclusion in Section 6.

2 *K*-Order LCP Computation in RAM

2.1 Notation

Consider an input text $T[0, n-1] = T[0]T[1]...T[n-1]$ drawn from any full-ordered constant or integer alphabet Σ. We assume the ending character $T[n-1]$ to be a unique character alphabetically smaller than any characters in $T[0, n-2]$, and use these notations for description clarity:

- $\mathsf{pre}(T, i)$ and $\mathsf{suf}(T, i)$: The prefix of T running from $T[0]$ to $T[i]$ and the suffix of T running from $T[i]$ to $T[n-1]$, respectively.
- SA_T: The suffix array of T, which is a permutation of integers in $[0, n)$ 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 $LCPA_T$: $\mathsf{lcp}(i, j)$ denotes the LCP length of $\mathsf{suf}(T, i)$ and $\mathsf{suf}(T, j)$ and $LCPA_T$ denotes the LCP-array of T, which consists of n integers in $[0, n)$, where $LCPA_T[i] = \mathsf{lcp}(SA_T[i], SA_T[i-1])$.
- Δ_k: The shorthand notation for $2^{\log n - k - 1}$ (not otherwise specified, the base of log is assumed to be 2).

2.2 LCP Batch Querying Technique

The LCP-BQT was previously proposed for the sparse SA construction [22]. Given T and a set P consisting of b pairs of position indices, LCP-BQT computes $\mathsf{lcp}(i, j)$ for each pair $(i, j) \in P$ in $\mathcal{O}(n \log b)$ time using $\mathcal{O}(n)$ space. The underlying idea is to find the indices (i_f, j_f) for (i, j) such that $T[i, i_f - 1] = T[j, j_f - 1]$

and $T[i_f] \neq T[j_f]$. For this purpose, it initially assigns P to P_0 and performs a loop of $\log b$ rounds on the set, 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 follows: 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 naive method for computing the LCP of two strings is to sequentially scan the two strings to compare their characters one by one until a difference is detected. However, this operation takes $O(2^{\log n})$ time in the worst and thus becomes a performance bottleneck.

As a solution for reducing the time overhead, the fingerprint (FP) function presented in [25] is employed herein for transforming a string comparison to its integer comparison counterpart that can be done in $O(1)$ time. The fingerprint of $T[i, j]$, namely $\mathsf{fp}(i, j)$, is calculated by the formula $\mathsf{fp}(i, j) = \sum_{p=i}^{j} \delta^{j-p} \cdot T[p] \mod L$, where L is a prime and δ is an integer randomly chosen from $[1, L)$. Two identical strings always have a common fingerprint, while the inverse is not true. Fortunately, it has been analyzed that the error probability of two different strings having an identical fingerprint can be ignored when L is large enough. Exploiting the use of the fingerprint function, each loop's round in LCP-BQT can be executed by the 3-step algorithmic framework given below.

1. Scan T rightward to iteratively compute the fingerprint of $\mathsf{pre}(T, l)$ by the formula $\mathsf{fp}(0, l) = \mathsf{fp}(0, l-1) \cdot \delta + T[l] \mod L$ and store $\mathsf{fp}(0, l)$ in the hash table if $l \in \{\{i_k - 1\} \cup \{j_k - 1\} \cup \{i_k + \Delta_k - 1\} \cup \{j_k + \Delta_k - 1\}, (i_k, j_k) \in P_k\}$.
2. For each $l \in \{\{i_k\} \cup \{j_k\}, (i_k, j_k) \in P_k\}$, compute the fingerprint of $T[l, l + \Delta_k - 1]$ by the formula $\mathsf{fp}(l, l + \Delta_k - 1) = \mathsf{fp}(0, l + \Delta_k - 1) - \mathsf{fp}(0, l - 1) \cdot \delta^{\Delta_k} \mod L$, where $\mathsf{fp}(0, l + \Delta_k - 1)$ and $\mathsf{fp}(0, l - 1)$ can be retrieved from the hash table in amortized $\mathcal{O}(1)$ time.
3. For each pair $(i_k, j_k) \in P_k$, compare $\mathsf{fp}(i_k, i_k + \Delta_k - 1)$ with $\mathsf{fp}(j_k, j_k + \Delta_k - 1)$. 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}.

For any round k, the following two properties remain invariant:

- 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 comparing the characters in $\mathsf{suf}(T, i_{\log b})$ and $\mathsf{suf}(T, j_{\log b})$ from left to right. Let $i_f = i_{\log b} + \mathsf{lcp}(i_{\log b}, j_{\log b})$ and $j_f = j_{\log b} + \mathsf{lcp}(i_{\log b}, j_{\log b})$, then i_f and j_f are the position indices to the right side of i and j such that $T[i, i_f - 1] = T[j, j_f - 1]$ and $T[i_f] \neq T[j_f]$.

The time complexity of LCP-BQT is dominated by the loop of $\log b$ rounds, where each round takes $\mathcal{O}(n)$ time for the first step to iteratively compute the fingerprints of $\mathsf{pre}(T, l)$, and $\mathcal{O}(b)$ time for the next two steps to compute and compare the fingerprints of neighboring strings in the LCP-array. The space in need is limited to $\mathcal{O}(b)$ words ($\lceil \log n \rceil$ bits per word) by employing a hash table for storing and retrieving the fingerprints. Hence, the LCPs of any b pairs of

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

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

it proceeds to perform a loop of $\log K$ rounds in lines 4-9. A key operation in round k is to sort the entries in CP_k and PP_k for iteratively computing the fingerprints of $\mathsf{pre}(T, l)$. After that, these values are retrieved from the hash table to compute and compare the fingerprints of $T[CP_k[2i]+1, CP_k[2i+1]]$ and $T[PP_k[2i]+1, PP_k[2i+1]]$ for producing CP_{k+1} and PP_{k+1}. More specifically, to 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 fingerprints are identical; otherwise, decrease $CP_k[2i+1]$ and $PP_k[2i+1]$ by Δ_{k+1}. The algorithm then 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 K-order 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 to the lemma stated below.

Lemma 1 *Given T and SA_T, the K-order $LCPA_T$ can be correctly computed in $\mathcal{O}(n \log K)$ time using $\mathcal{O}(n)$ words RAM space with a high probability.*

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

3 *K*-Order LCP Computation in External Memory

A hash table is employed in Algorithm 1 to store the fingerprints for quick lookups. This is feasible in the random access machine model, but will not be practical when n becomes too large and the table can not be wholly accommodated in the internal memory any more. To overcome the problem of insufficient internal memory, we reformulate lcpa-ram to design our disk-friendly external memory algorithm lcpa-disk, where the fingerprints in use are sequentially retrieved from the external memory with high I/O efficiency.

3.1 Notation

Given the internal memory 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 the internal memory. We extend CP_k and PP_k previously in lcpa-ram to define their siblings ECP_k and EPP_k in lcpa-disk. Both of them consist of $2n$ entries and each entry is a tuple of $\langle idx, pos, fp \rangle$ as defined below.

- idx: The index of an entry in ECP_k/EPP_k, where $ECP_k[i].idx = EPP_k[i].idx = i$.
- pos: The starting or ending position index for a substring in T.
- fp: The fingerprint of a prefix $\mathsf{pre}(0, pos)$.

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

Different from Algorithm 1, lcpa-disk performs two runs of external memory sort to arrange the fixed-size entries by their integer keys of $\log n$ bits during each round. As illustrated in Algorithm 2, the entries in ECP_k and EPP_k are sorted by *pos* to form $IECP_k$ and $IEPP_k$ in line 5, and then sorted back by *idx* to their original order in ECP_k and EPP_k by line 7. The first sort is for iteratively computing the fingerprint of $\mathsf{pre}(0, pos)$. During this process, we record the computation result in the fp field of each entry instead of a hash table. After the entries of $IECP_k$ and $IEPP_k$ have been sorted back by *idx*, these fp are used to produce ECP_{k+1} and EPP_{k+1} following the same way adopted in lcpa-ram. Given $M = \Omega(nW)^{0.5}$, where W is the size of an I/O buffer large enough to amortize the overhead of each access to the external memory, each run 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. Hence we get the following result.

Lemma 2 *Given T and SA_T, the K-order $LCPA_T$ can be correctly computed in $O(n \log K)$ time using $O(n)$ words 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 fingerprint of $\mathsf{pre}(T, i)$ and assign $\mathsf{fp}(0, i)$ to $IECP_k[j].fp$ or $IEPP_k[j].fp$ if $IECP_k[j].pos = i$ or $IEPP_k[j].pos = i$, respectively.
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 $(\mathsf{fp}(ECP_k[2i].pos{+}1, ECP_k[2i{+}1].pos), \mathsf{fp}(EPP_k[2i].pos{+}1, EPP_k[2i{+}1].pos))$ 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 time complexity of lcpa-disk is dominated by the while-loop of $\log K$ rounds. One solution for reducing the running time is to decrease the loop's rounds.

Following the idea, we modify the procedure in lines 8-9 of lcpa-disk as below to generate ECP_{k+2}/EPP_{k+2} directly from ECP_k/EPP_k.

1. Compute and compare the fingerprints 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 not equal, go to step 3.
2. Compute and compare the fingerprints 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}. Go to step 4.
3. Compute and compare the fingerprints 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 $\Delta_{k+1} + \Delta_{k+2}$.
4. Let $ECP_{k+2} = ECP_k$ and $EPP_{k+2} = EPP_k$.
5. Let $k = k + 2$.

This method can be generalized to simultaneously execute any number of successive rounds in the loop. However, it also causes a side effect on the disk use of the algorithm. That is, the workspace in use is nearly doubled, because each entry of ECP_k/EPP_k is extended to hold two more fingerprints for executing steps 2 or 3 when merging every two successive rounds into one. We show in Section 5 that our refined algorithm, namely lcpa-disk-m, can achieve a substantial speed improvement against lcpa-disk at the expense of an increase in disk space usage.

4 *K*-Order LCP Computation in a Distributed System

In what follows, we demonstrate how to parallelize our external memory algorithms lcpa-disk and lcpa-disk-m in a distributed system consisting of a cluster of d computing nodes $\{N_0, N_1, ...N_{d-1}\}$ shown in Fig.1. The internal and external memory capacities of each node are M and E, respectively, where E is far larger than M. All the nodes are interconnected by a high-speed switch operating in the full duplex mode.

4.1 Notation

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 one block to each node. Similarly, the entries in ECP_k/EPP_k and $IECP_k/IEPP_k$ are also evenly divided and distributed to all the computing nodes. The following symbols are used to present our distributed algorithm, where $i, j \in [0, d)$ and $k \in [0, \log K)$.

- SA_{T_i}: The i-th block of SA_T residing on N_i, where $SA_{T_i}[j] = SA_T[ie + j]$.
- $ECP_{i,k}/EPP_{i,k}$: The i-th block of ECP_k/EPP_k residing on N_i, where $ECP_{i,k}[j]$ $ECP_k[ie + j]$ and $EPP_{i,k}[j] = EPP_k[ie + j]$.
- $IECP_{i,k}/IEPP_{i,k}$: The i-th block of $IECP_k/IEPP_k$ residing on N_i, where $IECP_{i,k}[j] = IECP_k[ie + j]$ and $IEPP_{i,k}[j] = IEPP_k[ie + j]$.
- $SB1_{i,j}$ and $SB2_{i,j}$: Each N_i has two sending buffers $SB1_{i,j}$ and $SB2_{i,j}$ for every other node N_j $(j \neq i)$, where $SB1_{i,j}$ and $SB2_{i,j}$ cache the entries of $ECP_{i,k}/IECP_{i,k}$ and $EPP_{i,k}/IEPP_{i,k}$ to be sent to N_j, respectively.
- $RB1_{i,j}$ and $RB2_{i,j}$: Each N_i has two receiving buffers $RB1_{i,j}$ and $RB2_{i,j}$ for every other node N_j $(j \neq i)$, where $RB1_{i,j}$ and $RB2_{i,j}$ cache the entries of $ECP_{j,k}/IECP_{j,k}$ and $EPP_{j,k}/IEPP_{j,k}$ received from N_j, respectively.

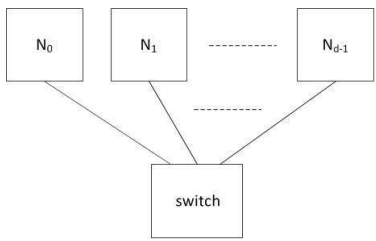

Fig. 1. The distributed system of d computing nodes.

4.2 Details

The algorithms lcpa-disk and lcpa-disk-m compute $ECP_{\log K}$ and $EPP_{\log K}$ via two runs of radix-sort in external memory. To emulate this radix-sort procedure in our distributed system, we set up a group of sending/receiding buffers in each computing nodes for data transmission during the sorts.

Algorithm 3 shows our distributed algorithm lcpa-ds, where each round of the while-loop performs two runs of radix-sorts to compute the fp field of each entry in ECP_k and EPP_k by using the sending and receiving buffers in lines 7-15. Specifically, for each entry x in $ECP_{i,k}$, node N_i dispatches x to the sending buffer $SB1_{i,j}$ if $j = x.pos/e$ and delivers it to $RB1_{j,i}$ on N_j in lines 7-8. Upon the arrival of each entry from the other nodes, N_j caches the entry in its receiving buffers $\{RB1_{j,0}, RB1_{j,1}, ..., RB1_{j,d-1}\}$ and then radix-sorts the received entries by pos to form $IECP_{j,k}$ in lines 9-10. Then it scans T_j from left to right for iteratively computing the fingerprints of the involved prefix and record them in the fp fields of the corresponding entries in $IECP_{j,k}$. Thereafter, in lines 12-13, each entry y satisfying $i = y.pos/e$ is moved from $IECP_{j,k}$ to the sending buffer $SB1_{j,i}$ and then delivered to $RB1_{i,j}$ on N_i. Finally, N_i radix-sorts the received entries in $\{RB1_{i,0}, RB1_{i,1}, ..., RB1_{i,d-1}\}$ by their idx to regenerate $ECP_{i,k}$ in lines 14-15. Note that the entries in $EPP_{i,k}$ can be processed using

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 For $j \in [0, d)$, create receiving buffers $RB1_{i,j}$ and $RB2_{i,j}$.
6 **while** $k < \log K$ **do**
7 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 $[je, je+e)$.
8 For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to $RB1_{j,i}$ and $RB2_{j,i}$ on N_j.
9 For $j \in [0, d)$, cache entries from $SB1_{j,i}$ and $SB2_{j,i}$ on N_j in $RB1_{i,j}$ and $RB2_{i,j}$.
10 Radix-sort entries in $RB1_i$ and $RB2_i$ by pos to produce $IECP_{i,k}$ and $IEPP_{i,k}$.
11 For $p \in [0, e)$ and $q \in [0, 2e)$, scan T_i rightward to iteratively compute the fingerprint of $\mathsf{pre}(T, ie+p)$ and assign $\mathsf{fp}(0, ie+p)$ to $IECP_{i,k}[q].fp$ or $IEPP_{i,k}[q].fp$ if $IECP_{i,k}[q].pos = ie+p$ or $IEPP_{i,k}[q].pos = ie+p$.
12 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 $[je, je+e)$.
13 For $j \in [0, d)$, send entries in $SB1_{i,j}$ and $SB2_{i,j}$ to $RB1_{j,i}$ and $RB2_{j,i}$ on N_j.
14 For $j \in [0, d)$, cache entries from $SB1_{j,i}$ and $SB2_{j,i}$ on N_j in $RB1_{i,j}$ and $RB2_{i,j}$.
15 Radix-sort entries in $RB1_i$ and $RB2_i$ by idx to reproduce $ECP_{i,k}$ and $EPP_{i,k}$.
16 For $p \in [0, e)$, scan $ECP_{i,k}$ and $EPP_{i,k}$ rightward to compute and compare the fingerprints 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 $EPP_{i,k+1}$.
17 Let $k = k+1$.
18 **end**
19 For $p \in [0, e)$, scan T_i, $ECP_{i,\log K}$ and $EPP_{i,\log K}$ rightward to literally compute $\Upsilon_{i,p} = \mathsf{lcp}(ECP_{i,\log K}[2p].pos, EPP_{i,\log K}[2p].pos)$.
20 For $p \in [0, e)$, let $\mathsf{lcp}(SA_{T_i}[p], SA_{T,i}[p-1]) = ECP_{i,\log K}[2p].pos + \Upsilon_{i,p} - SA_{T_i}[p]$.

$\{SB2_{i,0}, SB2_{i,1}, ..., SB2_{i,d-1}\}$ and $\{RB2_{i,0}, RB2_{i,1}, ..., RB2_{i,d-1}\}$ in the same way as described above. Then, the algorithm generates $ECP_{i,k+1}$ and $EPP_{i,k+1}$ by scanning $ECP_{i,k}$ and $EPP_{i,k}$ once in line 16.

After the while-loop, the steps in lines 19-20 figure out the LCP-array of $T[ie, ie+e)$ in linear time from $ECP_{i,\log K}$, $EPP_{i,\log K}$ and SA_{T_i} residing on N_i. Finally, we can simply collect and concatenate the LCP-array on each node to form $LCPA_T$. Hence, we get this result:

Lemma 3 *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 exploits the use of sending/receiving buffers to perform the radix-sorts among the computing nodes in a distributed manner, where the overhead of time and space for data transmission sums up to $\mathcal{O}(e \log K)$ during the whole loop, for each node sends and receives $\mathcal{O}(e)$ entries per round.

It is noteworthy that the internal memory on each node is partitioned into two parts, where the first part maintains the sending/receiving buffers and the other part establishes the I/O buffers for reading/writing and sorting the entries of $ECP_{i,k}$ and $EPP_{i,k}$. High performance can be achieved if these buffers are large enough to compensate the delay of data transmission and amortize the overhead of each I/O operation.

5 Experimental Results

A series of simulation experiments are conducted on a real data set collected from the web site http://download.wikimedia.org/enwiki/ for performance evaluation of our C++ programs for the external and distributed algorithms presented in Section 3 and 4, in terms of time and space consumption. We employ lcpa-disk-m as a baseline for performance and scalability assessment of lcpa-disk. The hardware platform for lcpa-disk and lcpa-disk-m is a computer equipped with 2 Intel Xeron E3-1220 CPUs, a 4GB DDR3 main memory and a 2TB 7200RPM disk, while the one for lcpa-ds is a distributed system consisting of 4 identical computers interconnected with a gigabit switch. We use gcc 4.8.4 and its MPI alternative (mpicc) to compile the programs for lcpa-disk/lcpa-disk-m and lcpa-ds, respectively, under Ubuntu 14.04 operating system.

STXXL [26] is a C++ STL library designed for efficient computations in external memory, freely available at http://stxxl.sourceforge.net/. Instead of to develop a radix sorter specific for our purpose, we use STXXL to perform the external memory sorts in our programs. Specifically, a priority queue provided by STXXL is employed for sorting the entries in ECP_k/EPP_k by *pos* to form $IECP_k/IEPP_k$ and another for sorting them back by *idx*. Benefitting from the powerful priority queues, the programs for lcpa-disk, lcpa-disk-m and lcpa-ds are less than 400, 600 and 700 lines, respectively.

Each result in the following tables and figures is a mean of two runs of the programs, where the running time and peak disk use are collected by shell command "time" and "stxxl::block_manager" provided by STXXL, respectively.

5.1 Performance of the External Algorithms

The following parameters and metrics are used for the experiments:

- S: corpora size.
- ST: total running time.

– MT: average running time spent in processing a character per loop's round.
– PD: peak disk use.
– LR: number of loop's rounds.
– W: every W successive loop's rounds in lcpa-disk are merged into one in lcpa-disk-m.
– H: internal memory allocated to each priority queue when sorting the entries in external memory.

To show the effect of W, we demonstrate in Table 1 the experimental results of lcpa-disk and lcpa-disk-m. As can be seen, lcpa-disk-m has a smaller ST than lcpa-disk when $W = 2$, while the latter outperforms the former in terms of MT and PD. Specifically, given $K = 8192$, the speed of lcpa-disk-m for $W = 2$ is nearly 1/3 faster than that of lcpa-disk when S varies from 200MB to 2GB. However, the total running time of lcpa-disk-m for $W = 3$ grows rapidly as S increases and exceeds that of lcpa-disk when S approaches 2GB. This can be explained as follows. As described in Section 3, lcpa-disk-m can merge every W successive loop's rounds for decreasing LR by computing all the fingerprints possibly involved in these W successive loop rounds. For example, given $W = 2$ and $W = 3$, lcpa-disk-m maintains 3 and 7 fingerprints in each entry of ECP_k and EPP_k, respectively, and updates them during each round of the loop. This leads to a sharp growth in the computation and I/O overhead per round against lcpa-disk and becomes a performance bottleneck.

Fig. 2 illustrates the variation trend of MT with increasing S. The non-linear relationship between the two parameters (i.e. MT and S) violates the assumption that MT is linear proportional to S. This behavior is partially due to the use of STXXL, where the priority queue provided by the library takes logarithmic time to sort elements in external memory.

Table 1. The performance results for the external memory algorithms.

S	lcpa-disk				lcpa-disk-m ($W = 2$)				lcpa-disk-m ($W = 3$)			
(MB)	LR	MT	ST	PD	LR	MT	ST	PD	LR	MT	ST	PD
200	13	1.03	2803	30.94	7	1.33	1950	47.36	5	2.43	2541	77.52
400	13	1.11	6034	31.50	7	1.40	4108	47.36	5	2.20	4605	78.58
600	13	1.18	9650	31.58	7	1.48	6928	47.36	5	3.14	9860	130.86
800	13	1.34	14557	31.52	7	1.80	10547	47.38	5	3.06	12827	103.50
2048	13	1.72	48021	46.04	7	2.47	37029	60.48	5	4.59	49280	94.40

Notes: LR, MT in microseconds per (character · round), ST in seconds and PD in bytes per character, where $K = 8192$, $W \in \{2, 3\}$ and S from 200MB to 2GB.

As reported in [19], the disk space requirements for eSAIS and LCPscan are $65n$ and $21n$ bytes, respectively, while the peak disk use of our implementation for lcpa-disk-m ($W = 2$) rises up to $61n$ bytes for processing a 2GB corpora. This indicates that lcpa-disk-m has a space requirement comparable to that of eSAIS,

S (MB)	LR	lcpa-ds ($N = 4$)			lcpa-disk-m			ST Ratio (%)
		MT	ST	PD	MT	ST	PD	
200	7	0.71	1038	13.50	1.33	1950	47.36	0.54
400	7	0.72	2113	13.50	1.40	4108	47.36	0.52
600	7	0.74	3235	13.75	1.48	6928	47.36	0.50
800	7	0.78	4547	13.82	1.80	10547	47.38	0.43
2048	7	1.24	18617	16.21	2.47	37029	60.48	0.50

Notes: LR, MT in microseconds per (character · round), ST in seconds and PD in bytes per character, where $K = 8192$, $d = 4$, $W = 2$ and S from 200MB to 2GB.

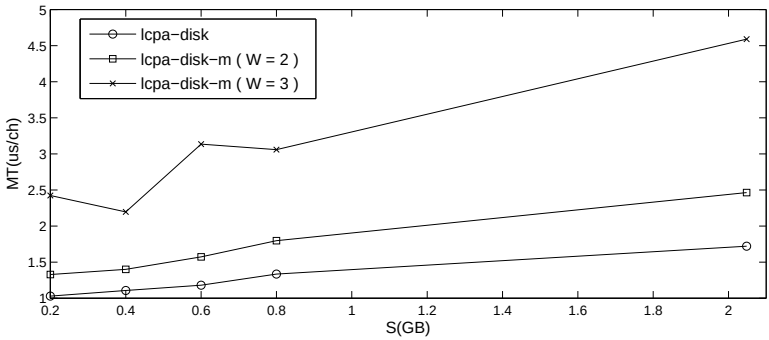

Fig. 2. MT for lcpa-disk and lcpa-disk-m with S varying from 200MB to 2GB, given $K = 8192$ and $W \in \{2, 3\}$.

but around 3 times as that of LCPscan. However, as will be seen in the following part, our method is much easier to be implemented and strongly scalable for the parallel and distributed model. In particular, the communication overhead on each node of the distributed model is balanced as $\mathcal{O}(\frac{n}{d})$. For $d \geq 3$, the space required by each node will be less than that of LCPscan.

5.2 Performance of the Distributed Algorithm

The algorithm lcpa-ds is strongly scalable, in terms of that it can be naturally executed in parallel except for the external memory sorts. We have described in Section 4 how to emulate the sorter by using a group of sending/receiving buffers. The communication overhead of each computing node in Fig. 1 is upper bounded by $\mathcal{O}(e)$. According to Amdahl's Law, the theoretical speed of lcpa-ds is $d - 1$ times faster than that of lcpa-disk, where d is the number of computing nodes in the distributed system. To validate this point, we adopt lcpa-disk-m as a baseline to evaluate the performance of lcpa-ds.

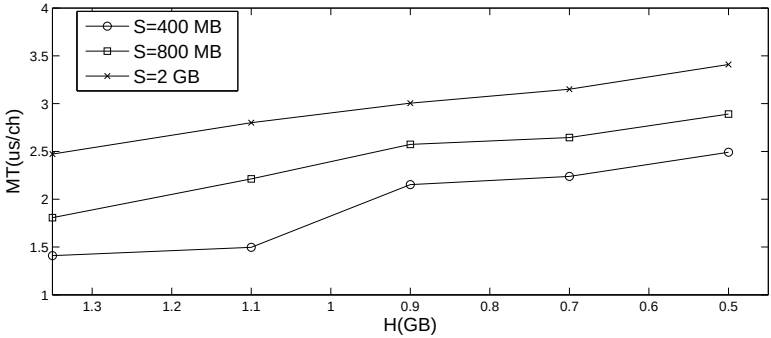

Fig. 3. MT for lcpa-disk-m with H varying from 1.35 to 0.5GB, given $K = 8192$, $W = 2$, $d = 4$ and $S \in \{400, 800, 2048\}$ MB.

As observed from Table. 2, given $K = 8192$, $W = 2$ and $d = 4$, lcpa-ds outperforms lcpa-disk-m in terms of time and space consumption, where MT/ST and PD for the former are 1/2 and 1/4 times of that for the latter, respectively. Particularly, MT for lcpa-ds increases from 0.71 to 1.24 when S increases from 200MB to 2GB. However, the performance gain does not fully meet our expectation for the ideal case. The main reason lies on the limited internal memory capacity available on each computing node. In our implementation of the program, each computing node spends its internal memory on not only the sending/receiving buffers, but also the I/O buffers and internal memory heaps. Each internal memory heap is employed by a priority queue for amortizing the overhead of disk accesses when swapping data between the internal and external memory. The swapping process is fast when H is large enough, because most of the data can be cached in the internal memory heap. However, a significant performance degradation in MT will occur when H becomes smaller, as shown in Fig. 3. One solution for relieving the problem is to reduce the block size e by adding more computing nodes. Fig. 4 shows that, when S increases from 200MB to 2GB, MT is much smaller and grows more slowly for $d = 4$ against that for $d = 2$, due to a decrease in the overhead for I/O operations.

6 Conclusion

We present in this paper a practical K-order LCP-array 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 internal memory, external

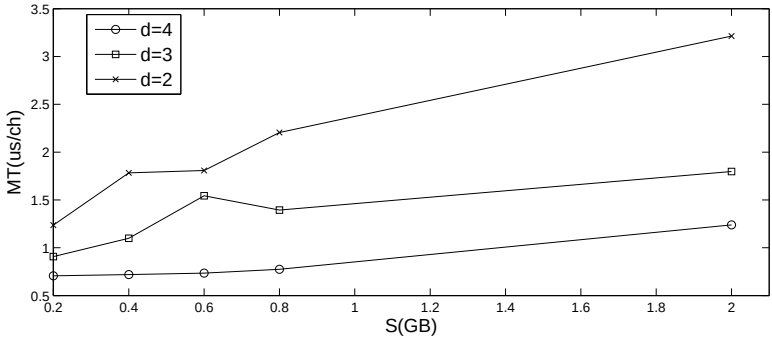

Fig. 4. MT for lcpa-ds with S varying from 200MB to 2GB, given $K = 8192$, $W = 2$ and $d \in \{2, 3, 4\}$.

memory 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 a lack of scalable LCP-array construction algorithms for such a distributed model. In this sense, our algorithms provide a candidate solution to meet the demand. For performance improvement of the algorithms, we are investigating techniques to exploit the use of GPUs for speeding up the computation of fingerprints.

Acknowledgment

The work of G. Nong was supported by the Guangzhou Science and Technology Program grant 201707010165 and the Project of DEGP grant 2014KTSCX007. The work of W. H. Chan was supported by GRF (18300215), Research Grant Council, Hong Kong SAR.

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. S. Burkhardt and J. Kärkkäinen. Fast Lightweight Suffix Array Construction and Checking. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 55–69, Morelia, Mexico, May 2003.
4. G. Manzini and P. Ferragina. Engineering a Lightweight Suffix Array Construction Algorithm. *Algorithmica*, 40:33–50, Sep 2004.
5. K. B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

6. P. Ko and S. Aluru. Space Efficient Linear Time COnstruction of Suffix Arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 200–210, Morelia, Mexico, May 2003.
7. D. K. Kim, J. Jo, and H. Park. A Fast Algorithm for Constructing Suffix Arrays for Fixed-Size Alphabets. In *Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms*, pages 25–28, Angra dos Reis, Brazil, May 2004.
8. J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 943–955, Eindhoven, Netherlands, June 2003.
9. 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.
10. S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*, 39(2):1–31, 2007.
11. J. Fischer. Inducing the LCP-Array. In *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. 2011.
12. 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, 2012.
13. T. Kasai, G. Lee, and H. Arimura. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, Jerusalem, Israel, July 2001.
14. G. Manzini. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 9th Workshop on Algorithm Theory*, pages 372–383, Humlebaek, Denmark, July 2004.
15. J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted Longest-Common-Prefix Array. In *Proceedings of the 20th Annual Symposuim on Combinatorial Pattern Matching*, pages 181–192, Lille, France, June 2009.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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*, pages 326–337, Ljubljana, Slovenia, September 2012.

22. P. Bille, I. L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse Suffix Tree Construction in Small Space. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming*, pages 148–159, Riga, Latvia, July 2013.
23. 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.
24. 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.
25. R. Karp and M. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.
26. R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL Data Sets. *Software: Practice and Experience*, 38(6):589–637, 2008.