Fast Parallel Computation of Longest Common Prefixes

Julian Shun
Carnegie Mellon University
Email: jshun@cs.cmu.edu

***Abstract*—Suffix arrays and the corresponding longest common prefix (LCP) array have wide applications in bioinformatics, information retrieval and data compression. In this work, we propose and theoretically analyze new parallel algorithms for computing the LCP array given the suffix array as input. Most of our algorithms have a work and depth (parallel time) complexity related to the LCP values of the input. We also present a slight variation of Kärkkäinen and Sanders' skew algorithm that requires linear work and poly-logarithmic depth in the worst case. We present a comprehensive experimental study of our parallel algorithms along with existing parallel and sequential LCP algorithms. On a variety of real-world and artificial strings, we show that on a 40-core shared-memory machine our fastest algorithm is up to 2.3 times faster than the fastest existing parallel algorithm, and up to 21.8 times faster than the fastest sequential LCP algorithm.**

I. Introduction

Suffix arrays [27] (also known as PAT arrays [12]) along with the corresponding longest common prefix array have applications in many fields, including bioinformatics, information retrieval and data compression. The suffix array (SA) is a data structure that stores the suffixes of a string in lexicographical order. Many applications of suffix arrays also require the longest common prefix (LCP) array, which stores the length of the longest common prefix between every adjacent pairs of suffixes in the suffix array. For example, the LCP values are used for efficient pattern matching with a suffix array [27], and used along with the suffix array to build a suffix tree [38] or simulate a suffix tree traversal [1]. The suffix array and its corresponding LCP array are often preferred over suffix trees for text indexing due to their lower space requirements [13]. With the rapid growth in data sizes, having fast parallel algorithms for suffix arrays and LCP arrays are particularly important. While there exists algorithms that compute both the suffix array and LCP array together, sometimes the suffix array is already available, and it is beneficial to have a fast algorithm for computing just the LCP array. Furthermore, separating the computation of SA and LCP allows one to use a fast SA algorithm that does not compute the LCP values, followed by a fast LCP algorithm. With such a separation, improvements in either SA algorithms or LCP algorithms improves the running time of the SA+LCP computation.

The suffix array and the first algorithm for constructing it were described by Manber and Myers [27]. Their sequential algorithm requires $O(n \log n)$ work[1], and also produces the LCP array. The first linear-work suffix array algorithms were described independently by Kärkkäinen and Sanders [17], Ko and Aluru [24] and Kim et al. [23]. Among these, the skew algorithm [17] (also named DC3 in [20]) of Kärkkäinen and Sanders can also compute the LCP array. In addition, many superlinear-work suffix array algorithms exist (see e.g. [33]), and some are faster in practice than the linear-work algorithms for certain inputs.

As for sequential standalone LCP algorithms (which compute the LCP array given the SA as input), the first linear-work algorithm was described by Kasai et al. [22]. Kärkkäinen et al. [19] later describe a linear-work algorithm for computing the permuted longest common prefix (PLCP) array. The LCP array can easily be computed from the PLCP array, and Kärkkäinen et al. show that their approach is more efficient in practice than that of Kasai et al. They also discuss another technique in the same paper based on irreducible LCP values, which requires $O(n \log n)$ work. The details of these algorithms are described in Section II. Gog and Ohlebusch [10] present a more space-efficient sequential LCP algorithm that requires the Burrows-Wheeler Transform [5] as input and requires $O(n^2)$ work in the worst case. Fischer [8] presents a sequential linear-work algorithm which computes both the SA and LCP, and is based on a modification of the sequential linear-work suffix array algorithm of Nong et al. [32]. The algorithm is not a standalone LCP algorithm in that both the SA and LCP are constructed. They show that the algorithm is competitive with using the fastest SA algorithm combined with the fastest standalone LCP algorithm for many real-world inputs. There have also been many papers describing how to reduce the working space requirements of LCP computation [3, 28, 34, 19, 41, 10, 2, 11] and adapting them to external memory [26, 4, 18].

As for parallel algorithms, there are two existing methods for computing the LCP values. The first method is to use the skew algorithm of Kärkkäinen and Sanders [17], which runs in linear work and $O(\log^2 n)$ depth (number of parallel time steps) with high probability[2]. Note that the skew algorithm is not a standalone LCP algorithm as it requires computing both the SA and LCP together. Deo and Keely [6] present a standalone parallel LCP algorithm for GPUs that is based on a parallelization of the sequential algorithm by Kasai et al. [22].

We note that by first constructing the suffix tree, we can obtain the LCP values by inspecting the depth of each internal node in the tree. Linear work and poly-logarithmic depth parallel suffix tree algorithms exist [15, 37, 7, 38], however, this approach is less satisfactory since constructing the suffix tree is less efficient in practice than constructing the SA and LCP array together. In fact, the fastest parallel suffix tree algorithm in practice requires first constructing the SA and LCP array [38].

With a fast parallel LCP algorithm, the performance of parallel applications that require the SA and LCP could be

[1]Throughout the paper, we use $\log x$ to denote the base 2 logarithm of x.

[2]We use "with high probability" (w.h.p.) to mean with probability at least $1 - 1/n^c$ for a constant $c > 0$.

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.37

Algorithm	Work	Depth
klaap-LCP (seq.)	$O(n)$	$O(n)$
kmp-LCP (seq.)	$O(n)$	$O(n)$
naive-LCP	$O(nl_{\text{avg}})$	$O(l_{\max})$
skew-SA+LCP	$O(n)$ w.h.p.	$O(\log^2 n)$ w.h.p.
skew-LCP	$O(n)$	$O(\log^2 n)$
par-iLCP	$O(n \log n)$	$O(\log n + l_{\max})$
par-LCP	$O(n + Kl_{\max})$	$O(n/K + l_{\max})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + l_{\max})$
par-PLCP	$O(n + Kl_{\max})$	$O(n/K + l_{\max})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + l_{\max})$
dk-LCP	$O(n + Kl_{\max})$	$O(n/K + \log n + l_{\max})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + \log n + l_{\max})$
dk-PLCP	$O(n + Kl_{\max})$	$O(n/K + \log n + l_{\max})$
	$O(n + Kl_{\text{avg}})$ expected	$O(n/K + \log n + l_{\max})$

TABLE I: Work and depth bounds for LCP algorithms. n = input size, $l_{\max}$ = maximum lcp value, l_{avg} = average lcp value, and K is an algorithm parameter, which trades off between work and depth. Our new algorithms are shown in bold font.

improved. For example, the fastest parallel algorithms for suffix tree construction [38] and Lempel-Ziv factorization [39] require computing the SA and LCP array, which is the dominant cost of the algorithms (at least 80% of the total running time).

Our contributions. In this work, we present several parallel standalone LCP algorithms. The first two are based on a parallelization of the sequential algorithms of [22] and [19] (par-LCP and par-PLCP, respectively), and require $O(n+Kl_{\max})$ work and $O(n/K+l_{\max})$ depth for a parameter $K \leq n$, where $l_{\max}$ is the maximum LCP value of the suffixes of the string. The parameter K represents a trade-off between work and parallelism. We discuss variants of these algorithms that require $O(n + Kl_{\text{avg}})$ expected work and $O(n/K + l_{\max})$ depth. The third algorithm is a slight modification of the skew algorithm [17], and requires linear work and $O(\log^2 n)$ depth in the worst case. We also apply Deo and Keely's parallelization idea (dk-LCP) to the sequential algorithm of Kärkkäinen et al. [19] (we refer to this variant as dk-PLCP). Finally, we present a straightforward parallelization of the irreducible LCP algorithm of Kärkkäinen et al. [19] (par-iLCP), which requires $O(n \log n)$ work and $O(\log n+l_{\max})$ depth. We note that the only two parallel algorithms that are require $O(n)$ work and poly-logarithmic depth independent of the LCP values of the string (i.e. are work-efficient) are the original skew algorithm (skew-SA+LCP) and skew-LCP, our variant for standalone LCP computation. For reference, we provide a table of the work and depth bounds for LCP algorithms in Table I, with our new algorithms/variants shown in bold font.

We present the first comprehensive evaluation of shared-memory implementations of parallel LCP algorithms, comparing our algorithms along with our CPU implementation of the parallel algorithm of Deo and Keely [6] and the original parallel skew algorithm. We also compare with the fastest sequential algorithms for computing the LCP array. Our parallel implementations use Cilk Plus [25], which is a dynamic multithreading language where simple constructs (e.g. parallel for-loops) are used to indicate which parts of the program are safe to run in parallel, and a run-time scheduler assigns work to threads and performs load-balancing. Our experiments on a 40-core shared-memory machine using a variety of real-world and artificial inputs show that par-PLCP usually performs the fastest among the parallel implementations, and outperforms Deo and Keely's algorithm by a factor of 1.5 to 2.3 in parallel. Compared to the fastest sequential LCP algorithm, par-PLCP is 14.4–21.8 times faster on 40 cores on the real-world inputs. We also show that while our linear-work and poly-logarithmic depth skew-LCP algorithm is 6–11x slower than par-PLCP, it outperforms the only existing algorithm with the same theoretical guarantees (skew-SA+LCP) by 1.4–2x.

i	S[i]	SA[i]	LCP[i]	PLCP[i]	suf_i
0	b	6	0	0	\$
1	a	5	0	3	a\$
2	n	3	1	2	*ana*\$
3	a	1	3	1	*anana*\$
4	n	0	0	0	*banana*\$
5	a	4	0	0	*na*\$
6	\$	2	2	0	*nana*\$

Fig. 1: SA, LCP, and PLCP arrays for S = *banana*\$.

II. Preliminaries

In this paper, we state complexity bounds of algorithms in the work-depth model, where the *work* W is equal to the number of operations required and the *depth* D is equal to the number of time steps required. Then if P processors are available, using Brent's scheduling theorem [16] we can bound the running time by $O(W/P + D)$. For sequential algorithms, the work and the depth terms are equivalent. We say that a parallel algorithm is *work-efficient* if its work is asymptotically equal to the work of the fastest sequential algorithm for the same problem.

We will make use of the basic parallel primitives, prefix sum and filter [16]. *Prefix sum* takes a sequence A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus a = a$ for all a, and returns the sequence $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus A[0] \oplus A[1] \oplus \ldots \oplus A[n-1])$. *Filter* takes a sequence A of length n, a predicate function f and returns a sequence A' of length n' containing the elements in $a \in A$ such that $f(a)$ returns true, in the same order that they appear in A. Filter can easily be implemented using prefix sum, and both require $O(n)$ work and $O(\log n)$ depth [16].

We denote a string by S, its length by n, the i'th character (using zero-based indexing) of a string S by S[i], and the sub-string starting at the i'th character and ending at the j'th character of S by S[$i, \ldots, j$]. We denote the alphabet of S by Σ and the alphabet size by $|\Sigma|$. We assume that a string ends with a special character \$ lexicographically smaller than all characters in Σ.

We define suf_i of a string S to be the suffix of S starting at position i (i.e. S[$i, \ldots, n-1$]). The *suffix array* [27] of S is an array SA of length n such that SA is a permutation of the integers $[0, \ldots, n-1]$ and $\text{suf}_{\text{SA}[0]} < \text{suf}_{\text{SA}[1]} < \ldots < \text{suf}_{\text{SA}[n-1]}$, where "<" means lexicographically smaller. The *inverse array* Rank of SA stores the rank of each suffix in SA. In particular, Rank[j] = i if and only if SA[i] = j. The *longest common prefix* array is an array LCP of length n such that LCP[0] = 0 and for $i > 0$, LCP[i] contains the length of the longest common prefix (lcp) between $\text{suf}_{\text{SA}[i-1]}$ and $\text{suf}_{\text{SA}[i]}$. The *permuted longest common prefix* array [19] is an array PLCP of length n that stores the lcp's in the order that they appear in S instead of their order in SA. In other words, PLCP[$SA[i]$] = LCP[i]. As an example, Fig. 1 shows the SA, LCP, PLCP arrays for the string S = *banana*\$.

A *standalone LCP algorithm* takes as input the string S, its suffix array SA and its length n, and outputs the LCP array.

naive-LCP. We first note that the LCP array can be computed naively by comparing every pair of adjacent suffixes one character at a time from the beginning of the suffixes. This approach can easily be parallelized as the comparison of each suffix pair is independent of any other suffix pair. The work is proportional to the sum of all lcp values, which can be bounded

```
1: procedure NAIVE-LCP(S, SA, n)
2:     LCP[0] = 0
3:     parfor i = 1 to n − 1 do
4:         h = 0
5:         j = SA[i]
6:         k = SA[i − 1]
7:         while S[j + h] == S[k + h] do
8:             h = h + 1
9:         LCP[i] = h
```

Fig. 2: naive-LCP: naive parallel LCP algorithm.

by $O(nl_{\text{avg}})$, where l_{avg} is the average lcp value. The work is quadratic in the worst case. The depth is proportional to the maximum lcp value, $l_{\max}$. The pseudo-code for this brute-force algorithm, which we refer to as *naive-LCP*, is shown in Fig. 2.

klaap-LCP. The first linear-work LCP algorithm was described by Kasai et al. [22], which we refer to as *klaap-LCP*. The pseudo-code for the klaap-LCP algorithm is shown in Fig. 3, and is adapted from [22]. The klaap-LCP algorithm uses the observation $\mathsf{LCP}[\mathsf{Rank}[i]] \geq \mathsf{LCP}[\mathsf{Rank}[i-1]] - 1$ to achieve linear work. The algorithm first computes the Rank array (Lines 2–3). It then uses the Rank array to iterate over the suffixes in the order that they appear in the original string, keeping a counter h of the lcp value of the current suffix. To compute the lcp value of the next suffix in original string order, character comparisons are performed between the suffix and its previous suffix in SA order, starting with the $(h-1)$'st character of the suffixes. Kasai et al. show that this algorithm requires at most $2n$ character comparisons, giving an $O(n)$ work algorithm.

```
1: procedure KLAAP-LCP(S, SA, n)
2:     for i = 0 to n − 1 do                    ▷ Compute Rank array
3:         Rank[SA[i]] = i
4:     LCP[0] = 0
5:     h = 0
6:     for i = 0 to n − 1 do
7:         if Rank[i] ≠ 0 then
8:             k = SA[Rank[i] − 1]
9:             while S[i + h] == S[k + h] do
10:                h = h + 1
11:            LCP[Rank[i]] = h
12:            if h > 0 then
13:                h = h − 1
```

Fig. 3: klaap-LCP: sequential LCP algorithm of Kasai et al.

kmp-LCP. Kärkkäinen et al. [19] describe a modification of the klaap-LCP algorithm, which writes out the lcp values in a permuted order. We refer to this algorithm as *kmp-LCP* and the pseudo-code for the algorithm is shown in Fig. 4, and is adapted from [19]. In particular, it writes the lcp value of the i'th suffix in S in position i in the PLCP array (Line 15). Obtaining the LCP array is done in a post-processing phase (Lines 16–17), by applying the relation $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$ for all $i \in [0, \ldots, n-1]$. Another difference from klaap-LCP is that in the pre-processing phase (Lines 3–4) kmp-LCP computes the index of the preceding suffix in SA for each suffix (stored in the Φ array), whereas klaap-LCP does this in the main loop using the Rank array (Line 8 of Algorithm 3). This saves a random read to SA, since the read to $\mathsf{SA}[i-1]$ on Line 4 of kmp-LCP is already in cache whereas Line 8 of klaap-LCP involves a random read to SA.

As in klaap-LCP, the number of character comparisons in kmp-LCP is at most $2n$, but kmp-LCP was shown to perform faster in practice (by about 50%) than klaap-LCP due to requiring fewer random reads and writes. The authors of [19] discuss space-saving variants which computes only n/q entries of PLCP but requires $O(q)$ work for a random access. They also discuss certain applications where the PLCP array may be used instead of the LCP array [36]. In this paper, we assume that the entire LCP array must be computed.

```
1: procedure KMP-LCP(S, SA, n)
2:     Φ[SA[0]] = −1
3:     for i = 1 to n − 1 do                    ▷ Compute Φ array
4:         Φ[SA[i]] = SA[i − 1]
5:     h = 0
6:     for i = 0 to n − 1 do
7:         if Φ[i] == −1 then
8:             h = 0
9:         else
10:            k = Φ[i]
11:            while S[i + h] == S[k + h] do
12:                h = h + 1
13:            if h > 0 then
14:                h = h − 1
15:        PLCP[i] = h
16:    for i = 0 to n − 1 do                    ▷ Convert PLCP to LCP
17:        LCP[i] = PLCP[SA[i]]
```

Fig. 4: kmp-LCP: sequential LCP algorithm of Kärkkäinen et al.

dk-LCP. Deo and Keely describe a parallel version of klaap-LCP for GPUs [6]. The pseudo-code for our implementation of their algorithm is shown in Fig. 5, and we refer to it as *dk-LCP*. Lines 2–3 are the same as in klaap-LCP, except done in parallel. The algorithm finds all of the indices i_j such that $\mathsf{LCP}[\mathsf{Rank}[i_j]] = 0$, which can be done by comparing the first character of each suffix with the first character of its previous suffix in SA, and applying a parallel filter (Line 5)[3]. In particular, we mark all the indices i such that $\mathsf{S}[i] \neq \mathsf{S}[SA[\mathsf{Rank}[i]] - 1]$ and apply a filter keeping just the marked indices. These indices form intervals $[i_j, \ldots, i_{j+1} - 1]$, and since the intervals could be large (especially for strings from a small alphabet), each interval that is larger than some threshold is split into sub-intervals, and in parallel the sequential klaap-LCP algorithm is applied to each interval (parallel for-loop on Line 6) and sub-interval (parallel for-loop on Line 8). Our implementation uses a threshold of $\lfloor n/K \rfloor$ for some input parameter $K \leq n$ (for simplicity, the pseudo-code assumes K evenly divides n, but can be adapted for the general case). Since the first suffix of a sub-interval may not have an lcp value of 0, there is extra work done relative to klaap-LCP in computing its lcp value (unlike in klaap-LCP, it does not know the lcp value of its previous suffix). Hence the total work can no longer be bounded by $O(n)$. We provide an analysis of the algorithm in Section III. Deo and Keely's original GPU algorithm also includes a load-balancing component, but since our implementation uses Cilk Plus [25], we leave this to the work-stealing run-time scheduler.

skew-SA+LCP. The *skew algorithm* [17] is a linear-work parallel suffix array construction algorithm, and can be used to also compute the LCP array during the suffix array construction. The skew algorithm works in 4 steps:

1) Recursively construct the suffix array SA_{12} and longest common prefix array LCP_{12} of the suffixes starting at positions i in S where $i \bmod 3 \neq 0$.
2) Use SA_{12} to construct the suffix array SA_0 of the positions i in S where $i \bmod 3 = 0$.
3) Merge SA_{12} and SA_0 together to form SA.
4) Use SA and LCP_{12} to compute the full LCP array.

[3]The number of these indices is at most $|\Sigma|$. Without loss of generality, in the pseudo-code, we assume that all characters in Σ appear in the string.

```
1:  procedure DK-LCP(S, SA, n)
2:      parfor i = 0 to n − 1 do                    ▷ Compute Rank array
3:          Rank[SA[i]] = i
4:      LCP[0] = 0, i_0 = 0
5:      Compute indices i_1 < i_2 < ... < i_{|Σ|−1} such that for all
            1 ≤ j < |Σ|, S[i] ≠ S[SA[Rank[i]] − 1]      ▷ lcp is 0
6:      parfor j = 0 to |Σ| − 1 do                  ▷ Parallelize over intervals
7:          B = ⌈ (i_{j+1} − i_j)K / n ⌉             ▷ Number of sub-intervals
8:          parfor b = 0 to B − 1 do                ▷ Parallelize over sub-intervals
9:              h = 0
10:             start = i_j + bn/K
11:             end = min{i_j + (b+1)n/K, i_{j+1}}
12:             for i = start to end − 1 do         ▷ Sequential klaap-LCP
13:                 if Rank[i] ≠ 0 then
14:                     k = SA[Rank[i] − 1]
15:                     while S[i + h] == S[k + h] do
16:                         h = h + 1
17:                     LCP[Rank[i]] = h
18:                     if h > 0 then
19:                         h = h − 1
```

Fig. 5: dk-LCP: parallel LCP algorithm of Deo and Keely.

To perform step (1) it assigns lexicographic integer labels $s'_i \in [1, \ldots, 2n/3]$ to the triples $\mathsf{S}[i, i+1, i+2]$ for $i \bmod 3 \neq 0$ using a stable integer sort followed by a prefix sum. If the names are all unique then the array of labels is the suffix array SA_{12}, and LCP_{12} contains all 0's; otherwise it recurses on the string $\mathsf{S}' = s_1 s_2$ where s_1 is formed by concatenating all of the labels s'_i for $i \bmod 3 = 1$ in order of i and s_2 is formed by concatenating all of the labels s'_j for $j \bmod 3 = 2$ in order of j. The authors of [17] show that the stable integer sorting here can be done in linear work and $O(\log n)$ depth w.h.p. for a constant initial alphabet by combining techniques in [35, 14].

To perform step (2), the suffixes at positions i where $i \bmod 3 = 0$ can be sorted by sorting the pairs $(\mathsf{S}[i], \mathsf{suf}_{i+1})$ using an integer sort, as the suffixes suf_{i+1} are at mod 1 positions and hence already in sorted order in SA_{12} from step (1). The integer sort requires $O(n)$ work and $O(\log n)$ depth w.h.p.

The merge in step (3) can be performed by using pairs $(\mathsf{S}[i], \mathsf{suf}_{i+1})$ if comparing a mod 0 suffix with a mod 1 suffix, and triples $(\mathsf{S}[i], \mathsf{S}[i+1], \mathsf{suf}_{i+2})$ if comparing a mod 0 suffix with a mod 2 suffix. This ensures that the suffixes appearing in the pairs or triples already appear in sorted order in SA_{12}. Computing the relative order of two suffixes in SA_{12} can be done in constant work by pre-computing an inverse array mapping each suffix to its position in SA_{12}. The inverse array can be computed in linear work and $O(1)$ depth. The merge can be done using a parallel merging algorithm in $O(n)$ work and $O(\log n)$ depth [16].

Finally, to perform step (4) the algorithm uses the fact that an lcp value in LCP corresponds to 3 times the corresponding value in LCP_{12}, and the fact that the lcp value between the two suffixes at positions i and j of the LCP_{12} array is equal to $\min_{i \leq k < j} \mathsf{LCP}_{12}[k]$. For two suffixes $\mathsf{suf}_{\mathsf{SA}[i-1]}$ and $\mathsf{suf}_{\mathsf{SA}[i]}$, the algorithm first compares c characters ($0 \leq c \leq 2$) from the beginning of the suffixes until both $(\mathsf{SA}[i-1]+c) \bmod 3 \neq 0$ and $(\mathsf{SA}[i]+c) \bmod 3 \neq 0$. If fewer than c characters match, then $\mathsf{LCP}[i] = c'$, where c' is the length of the prefix that matches. Otherwise, let l be equal to the lcp between $\mathsf{suf}_{\mathsf{SA}[i-1]+c}$ and $\mathsf{suf}_{\mathsf{SA}[i]+c}$. These suffixes are represented in LCP_{12} because they are at mod 1 and/or mod 2 positions, and the positions in LCP_{12} can be looked up using the inverse array from step (3). However, the suffixes may not be adjacent in LCP_{12}, so a range minima query between the two positions in LCP_{12} is done if necessary to give the lcp value between the suffixes. Then $\mathsf{LCP}[i]$ is equal to $c + 3l + l'$, where l' is the lcp value between $\mathsf{suf}_{\mathsf{SA}[i-1]+c+3l}$ and $\mathsf{suf}_{\mathsf{SA}[i]+c+3l}$. l' is at most 2 and is computed by comparing the characters of the suffixes one-by-one. To answer range minima queries in $O(1)$ work/depth, the algorithm builds a range minima query table over LCP_{12}, which requires $O(n)$ work and $O(\log n)$ depth [16].

The overall work of the algorithm is $O(n)$ since each level of recursion requires linear work and reduces the problem size to $2n/3$. The depth is $O(\log^2 n)$ w.h.p. as there are $O(\log n)$ levels of recursion, each requiring $O(\log n)$ depth w.h.p. We later show how to modify the skew algorithm to compute the LCP array given the suffix array as input.

irredicible-LCP. Kärkkäinen et al. [19] describe a technique for computing the PLCP array based on irreducible lcp values, which we refer to as *irreducible-LCP*. $\mathsf{PLCP}[i]$ is *reducible* if $\mathsf{S}[i-1] = \mathsf{S}[\mathsf{SA}[i-1]-1]$ and *irreducible* otherwise. For reducible values, it can be shown [28, 19] that $\mathsf{PLCP}[i] = \mathsf{PLCP}[i-1]-1$. The algorithm works by computing the PLCP values corresponding to the irreducible lcp's using the brute-force method of comparing suf_i and $\mathsf{suf}_{\mathsf{SA}[i-1]}$ from the beginning, and using the results to compute each remaining PLCP value in constant work. The authors show that the sum of all irreducible lcp values is at most $2n \log n$. Hence, the overall work is $O(n \log n)$ (note that this is not work-efficient). The authors also show that in practice the algorithm is slower than kmp-LCP. We later present a straightforward parallelization of this algorithm.

III. Algorithms and Analysis

In this section, we present several parallel algorithms for computing the longest common prefix array given a string and its corresponding suffix array. We also analyze the work and depth bounds of the algorithms.

par-LCP and par-PLCP. Our first approach is similar to that of Deo and Keely [6], but instead of requiring a pre-processing step to find the intervals that are processed in parallel, we split the input into equal-sized intervals. This approach can be used to parallelize both klaap-LCP and kmp-LCP. The algorithms use a parameter $K \leq n$, which trades off between parallelism and work, and split the input into intervals of size at most $\lfloor n/K \rfloor$ (there are at most $K+1$ intervals). We refer to our parallelization of klaap-LCP as *par-LCP* (pseudo-code shown in Fig. 6) and our parallelization of kmp-LCP as *par-PLCP* (pseudo-code shown in Fig. 7). For simplicity, the pseudo-code assumes K evenly divides n, but can be adapted for the general case. We process the intervals in parallel, where each interval runs klaap-LCP or kmp-LCP sequentially, with a counter h starting at 0. The parameter K could, for example, be set to $O(P)$ where P is the number of processors available to the computation, and is what we use in our experiments.

In par-LCP (Fig. 6), the Rank array is computed in parallel on Lines 2–3. Then Line 5 is a parallel for-loop splitting the indices into equal-sized chunks, where each chunk is processed sequentially in Lines 6–14 using klaap-LCP. For par-PLCP (Fig. 7), the loops computing Φ (Lines 3–4) and computing LCP (Lines 17–18) can be trivially parallelized. Again, on Line 5, the indices are split in a parallel for-loop, and each chunk is processed sequentially in Lines 6–16 using kmp-LCP.

In contrast to dk-LCP (and dk-PLCP, which is described next), par-LCP and par-PLCP do not have a pre-processing

```
1:  procedure PAR-LCP(S, SA, n)
2:      parfor i = 0 to n − 1 do                    ▷ Compute Rank array
3:          Rank[SA[i]] = i
4:      LCP[0] = 0
5:      parfor j = 0 to K − 1 do                    ▷ Parallelize over intervals
6:          h = 0
7:          for i = jn/K to (j+1)n/K − 1 do         ▷ Sequential klaap-LCP
8:              if Rank[i] ≠ 0 then
9:                  k = SA[Rank[i] − 1]
10:                 while S[i + h] == S[k + h] do
11:                     h = h + 1
12:                 LCP[Rank[i]] = h
13:                 if h > 0 then
14:                     h = h − 1
```

Fig. 6: par-LCP: our parallelization of klaap-LCP.

phase to find all the indices for which the lcp value is 0, therefore leading to splits that perform more extra work on average for the first element of each chunk. However, we later show experimentally the extra work is more than offset by not having to perform the pre-processing phase.

dk-PLCP. We observe that the approach of Deo and Keely can also be used to parallelize kmp-LCP and refer to this variant as *dk-PLCP*. Due to space constraints we do not show the pseudo-code for this algorithm, but it is very similar to that of dk-LCP in Fig. 5.

Analysis. We now analyze the theoretical performance of the four parallel algorithms (par-LCP, par-PLCP, dk-LCP and dk-PLCP) based on splitting the computation into intervals. In the analysis, we assume that K evenly divides n, but the bounds still hold in the general case. The performance is based on the maximum or average lcp value of the suffixes of the string, which we denote as $l_{\max}$ and l_{avg}, respectively.

Theorem 3.1: For a parameter $K \leq n$, par-LCP and par-PLCP require $O(n + Kl_{\max})$ work and $O(n/K + l_{\max})$ depth.

Proof: For each interval, the maximum value of the counter h is $l_{\max}$ and there are n/K decrements, so the number of character comparisons (equal to the number of times h is incremented) is at most $n/K + l_{\max}$. This analysis is similar to that of [22]. Over all K intervals, the number of character comparisons is at most $n + Kl_{\max}$. The work of the main loop (Lines 5–14 of par-LCP and Lines 5–16 of par-PLCP) is thus $O(n + Kl_{\max})$.

An alternative argument for the work bound of the main loop is that except for the first element of each interval, the work for the rest of the elements is exactly the same as in the sequential algorithm and hence bounded by $O(n)$. The first element of an interval can do at most $l_{\max}$ comparisons, and over all K intervals, this contributes $O(Kl_{\max})$ to the work. Hence the total work is bounded by $O(n + Kl_{\max})$.

The intervals can be processed in parallel, but each interval is done sequentially doing at most $n/K + l_{\max}$ comparisons, so the depth of the main loop is $O(n/K + l_{\max})$. The parallel loops on Lines 2–3 of par-LCP and Lines 3–4 and 17–18 of par-PLCP require $O(n)$ work and $O(1)$ depth. Therefore, the work of the algorithms is $O(n + Kl_{\max})$ and depth is $O(n/K + l_{\max})$. ■

Note that if $K = \omega(n/l_{\max})$ then our algorithms do more than $O(n)$ work in the worst case. However, in our experiments we set K to be the number of threads, which is less than $n/l_{\max}$ for most inputs. Also, for real-world strings the $O(Kl_{\max})$ term is usually very loose as it is unlikely that the first elements of many intervals have an lcp value close to $l_{\max}$.

By using randomization, we can improve the work bound to

```
1:  procedure PAR-PLCP(S, SA, n)
2:      Φ[SA[0]] = −1
3:      parfor i = 1 to n − 1 do                    ▷ Compute Φ array
4:          Φ[SA[i]] = SA[i − 1]
5:      parfor j = 0 to K − 1 do                    ▷ Parallelize over intervals
6:          h = 0
7:          for i = jn/K to (j+1)n/K − 1 do         ▷ Sequential kmp-LCP
8:              if Φ[i] == −1 then
9:                  h = 0
10:             else
11:                 k = Φ[i]
12:                 while S[i + h] == S[k + h] do
13:                     h = h + 1
14:                 if h > 0 then
15:                     h = h − 1
16:             PLCP[i] = h
17:     parfor i = 0 to n − 1 do                    ▷ Convert PLCP to LCP
18:         LCP[i] = PLCP[SA[i]]
```

Fig. 7: par-PLCP: our parallelization of kmp-LCP.

$O(n + Kl_{\text{avg}})$ in expectation, as discussed in Lemma 3.2 below. This improvement is significant when $l_{\text{avg}} \ll l_{\max}$.

Lemma 3.2: Modified versions of par-LCP and par-PLCP require $O(n + Kl_{\text{avg}})$ expected work and $O(n/K + l_{\max})$ depth.

Proof: Instead of fixing the interval start indices at jn/K for $0 \leq j < K$, we pick an integer uniformly at random between 0 and $n/K - 1$ and shift all start indices to the right by this amount. We then add back a start index at $i = 0$ (if it was shifted) to guarantee that all elements are processed.

We consider the extra work performed for the first elements of the intervals, except for at $i = 0$. Summing over all possible random shifts, each first element where $i > 0$ will be a first element of an interval exactly once, and the total extra work for these elements can be upper bounded by nl_{avg} (the sum over all lcp values). Each random shift is picked with $1/(n/K) = K/n$ probability, so the expected work for these elements for a single execution is $(K/n)nl_{\text{avg}} = Kl_{\text{avg}}$. The extra work for the first element at $i = 0$ can be bounded by $l_{\max}$. The remainder of the work done in the main loop is the same as in the sequential algorithm, and so contributes $O(n)$ to the total work. Therefore, the total expected work is $O(n + Kl_{\text{avg}})$.

Again, the depth is bounded by the maximum size of an interval plus $l_{\max}$, giving a bound of $O(n/K + l_{\max})$. ■

We also provide an analysis of dk-LCP and dk-PLCP, which is similar to that of par-LCP and par-PLCP.

Lemma 3.3: dk-LCP and dk-PLCP require $O(n + Kl_{\max})$ work and $O(n/K + \log n + l_{\max})$ depth.

Proof: For dk-LCP, computing the indices where the lcp value is 0 (Line 5) is done with a parallel filter, which requires $O(n)$ work and $O(\log n)$ depth. Lines 2–3 can be done in $O(n)$ work and $O(1)$ depth. Each interval larger than size n/K is divided into sub-intervals of size n/K (except for the last sub-interval which may contain fewer than n/K elements), so the overall depth becomes $O(n/K + \log n + l_{\max})$. Similar to the analysis of par-LCP and par-PLCP, the work for each sub-interval is $O(n/K + l_{\max})$. The intervals that were not sub-divided do no more work than the sequential algorithm as the first lcp value is 0, and hence contribute $O(n)$ work. The maximum number of sub-intervals is $O(K)$ so this gives an overall work of $O(n + Kl_{\max})$. The analysis for dk-PLCP is similar. ■

Analogous to Lemma 3.2, for dk-LCP and dk-PLCP, we can shift the sub-intervals in each interval by a random amount and obtain the bounds in the following lemma. The proof is

omitted as it is similar to the proof of Lemma 3.2.

Lemma 3.4: Modified versions of dk-LCP and dk-PLCP require $O(n + Kl_{\text{avg}})$ expected work and $O(n/K + \log n + l_{\max})$ depth.

Random Strings. Here we analyze the behavior of the algorithms on random strings. We consider properties of random strings from the alphabet Σ where each character of the string is chosen uniformly at random from Σ, and $|\Sigma| \geq 2$. The expected length of the longest repeated sub-string of a random string has been shown to be $O(\log_{|\Sigma|} n)$ [21, 27]. This is also the expected maximum lcp value, since the longest common prefix of any two suffixes is a repeated sub-string in the string.

Lemma 3.5: For a random string from an alphabet of size $|\Sigma| \geq 2$, par-LCP, par-PLCP, dk-LCP and dk-PLCP require $O(n)$ work and $O(\log n)$ depth in expectation for $K = O(n/\log n)$.

Proof: The expected maximum lcp value of a suffix is $O(\log_{|\Sigma|} n)$ which is $O(\log n)$ for $|\Sigma| \geq 2$. We apply Theorem 3.1 and Lemma 3.3 with $l_{\max} = O(\log n)$ and $K = O(n/\log n)$. ■

If $|\Sigma|$ is known beforehand then we can set $K = O(n/\log_{|\Sigma|} n)$, and achieve linear work and a depth bound of $O(\log_{|\Sigma|} n)$ for par-LCP and par-PLCP.

skew-LCP—Standalone LCP Computation with the Skew Algorithm. We discuss a slight modification of the skew algorithm [17] that can be used as a standalone LCP algorithm (referred to as *skew-LCP*) given the suffix array SA as input. We state the changes that need to be made to the skew algorithm as described in Section II.

For step (1), we construct SA_{12} by marking the indices i such that $\mathsf{SA}[i] \bmod 3 \neq 0$, and apply a parallel filter keeping just the elements at these indices. Computing the new lexicographic names is still done by comparing triples and using a parallel prefix sum to compute the new name of each triple. However, since the suffixes in SA_{12} are already sorted (SA is sorted), we can assign new lexicographic names in the range $[1, \ldots, 2n/3]$ based on the suffix's index in SA_{12}, instead of using an integer sort. Creating the string S' to recurse on is done as before—by moving all of the mod 1 suffixes to the beginning and mod 2 suffixes to the end of the string using a parallel for-loop. Steps (2) and (3) are no longer required since we do not need to generate SA. Step (4) to generate the LCP array remains the same as before.

Theorem 3.6: skew-LCP requires $O(n)$ work and $O(\log^2 n)$ depth.

Proof: For each level of recursion, the prefix sum and filter take linear work and $O(\log n)$ depth, and to answer range minima queries in $O(1)$ work and depth in step (4), a range minima query look-up table can be built in linear work and $O(\log n)$ depth [16]. As each recursive call reduces the problem to two-thirds of the original size, the work recurrence is $W(n) = W(2n/3) + O(n)$ and the depth recurrence is $D(n) = D(2n/3) + O(\log n)$. Solving the recurrences gives the theorem. ■

We note that the bounds of the original skew algorithm [17] computing both SA and the LCP array are $O(n)$ work and $O(\log^2 n)$ depth w.h.p. for a constant alphabet. The complexity bound required the use of integer sorting algorithms [35, 14] which limited the alphabet size. Since skew-LCP does not involve integer sorting, the bounds hold for general alphabets.

Just like the original skew algorithm, skew-LCP can be adapted to other models of computation using the techniques in [17]. On the Bulk Synchronous Parallel (BSP) model [42], skew-LCP requires $O(n/P + L\log^2 P + gn/P)$ time for a communication parameter g, synchronization cost L and number of processors P. This bound was true only for $P = O(n^{1-\epsilon})$ in the original skew algorithm due to the need for integer sorting. The bounds for skew-LCP in the external-memory and cache-oblivious models are the same as for the original skew algorithm—that is $O((n/B)\log_{M/B}(n/B))$ I/O's (external-memory) or cache misses (cache-oblivious) for a block size of B and a fast memory size of M.

```
1:  procedure PAR-ILCP(S, SA, n)
2:      Φ[SA[0]] = −1
3:      parfor i = 1 to n − 1 do                          ▷ Compute Φ array
4:          Φ[SA[i]] = SA[i − 1]
5:      Compute all indices i_1 < i_2 < ... < i_{m−1}, such that
            S[i_j − 1] ≠ S[Φ[i_j] − 1]
6:      i_0 = 0, i_m = n
7:      parfor j = 0 to m − 1 do
8:          h = 0
9:          if Φ[i_j] ≠ −1 then
10:             k = Φ[i_j]
11:             while S[i_j + h] == S[k + h] do
12:                 h = h + 1
13:         PLCP[i_j] = h                                 ▷ Irreducible lcp value
14:         parfor l = i_j + 1 to i_{j+1} − 1 do
15:             PLCP[l] = h − (l − i_j)                   ▷ Reducible lcp values
16:     parfor i = 0 to n − 1 do                          ▷ Convert PLCP to LCP
17:         LCP[i] = PLCP[SA[i]]
```

Fig. 8: parallel-iLCP: parallel irreducible LCP algorithm.

par-iLCP—A Parallel Irreducible LCP algorithm. We describe a straightforward parallelization of the irreducible-LCP algorithm described in Section II, which we call *par-iLCP*. The pseudo-code is shown in Fig. 8. The parallel for-loops on Lines 3–4 and 16–17 are the same as in par-PLCP, since the algorithm first computes the PLCP array before converting it to the LCP array. On Line 5, we compute all of the indices i_j, where $\mathsf{PLCP}[i_j]$ corresponds to an irreducible lcp value (an *irreducible index*). This is done with a parallel for-loop and a parallel filter with the predicate $\mathsf{S}[i_j - 1] \neq \mathsf{S}[\Phi[i_j] - 1]$ (equivalent to $\mathsf{S}[i_j - 1] \neq \mathsf{S}[\mathsf{SA}[i_j - 1] - 1]$), and requires $O(n)$ work and $O(\log n)$ depth.

Then for each irreducible index in parallel (Line 7), we first compute its PLCP value by comparing characters one-by-one (Lines 8–13). All of the indices after the irreducible index i_j and before the next irreducible index i_{j+1} correspond to reducible lcp values, so we then apply the formula $\mathsf{PLCP}[l] = \mathsf{PLCP}[i_j] - (l - i_j)$ from [28, 19] for all $i_j < l < i_{j+1}$ in parallel (Lines 14–15). The work of the main loop (Lines 7–15) is the same as in the sequential irreducible-LCP algorithm, namely $O(n \log n)$. The work for the rest of the algorithm is $O(n)$. The depth is $O(l_{\max} + \log n)$ as computing the lcp values for the irreducible indices requires $O(l_{\max})$ depth and the parallel filter requires $O(\log n)$ depth. This gives the following theorem:

Theorem 3.7: par-iLCP requires $O(n \log n)$ work and $O(\log n + l_{\max})$ depth.

IV. Experiments

In this section, we present a detailed experimental evaluation of LCP algorithms in a shared-memory setting. We first discuss our implementations and the experimental setup. We then discuss the performance of the standalone LCP implementations. Finally, we evaluate the implementations when used in conjunction with suffix array code. Additional experiments can be found in the Appendix.

	chr22	etext99	HG18.fasta	howto	jdk13c	proteins	rctail96	rfc	sprot34	Venter0	w3c2	wikisamp8	wikisamp9	random	identical	sqrtn
size (MB)	34.6	105	3083	39.4	69.7	1184	115	116	110	427	104	100	1000	100	100	100
$\|\Sigma\|$	5	146	27	197	113	27	93	120	66	5	256	204	207	10	1	2
$l_{\max}$	$2 \cdot 10^5$	$3 \cdot 10^5$	$2 \cdot 10^7$	70720	37334	$6 \cdot 10^5$	26597	3445	7373	1139	10^6	1265	2032	15	10^8	10^8
l_{avg}	1979	1109	$4 \cdot 10^5$	268	679	1422	282	93	89.1	44	42300	53.2	68	7.31	$5 \cdot 10^7$	$5 \cdot 10^7$
klaap-LCP (seq.)	2.34	6.67	315	2.16	2.94	76.7	5.69	6.09	5.71	31.8	4.27	4.8	56.1	7.39	0.522	1.86
kmp-LCP (seq.)	1.67	5.53	233	1.67	2.56	58.9	4.78	5.14	4.83	26.3	3.74	4.08	43.8	6.07	0.726	1.57
naive-LCP (T_1)	51.9	93.5	–	10	43.6	1420	37.2	17.6	19.2	61.3	3250	12.9	191	5.37	–	–
naive-LCP (T_{40})	2.11	2.82	–	0.326	1.32	45.5	0.965	0.403	0.373	**1.41**	119	0.256	4.01	**0.169**	–	–
naive-LCP (T_1/T_{40})	24.6	33.2	–	30.7	33	31.2	38.5	43.7	51.5	43.5	27.3	50.4	47.6	31.8	–	–
skew-LCP (T_1)	15.2	58.4	2610	18.5	45.6	887	91.7	82.2	67.3	257	69	63.4	784	34.1	18.9	34.9
skew-LCP (T_{40})	0.584	1.99	64	0.705	1.48	26.6	2.45	2.28	2.21	8.34	2.31	2.06	21.9	1.26	0.814	2.07
skew-LCP (T_1/T_{40})	26	29.3	40.8	26.2	30.8	33.3	37.4	36.1	30.4	30.8	29.9	30.8	35.8	27.1	23.2	16.9
par-iLCP (T_1)	2.97	9.27	407	2.5	3.11	87.2	6.68	7.8	6.79	49.3	4.64	5.57	62.7	11.3	0.976	1.81
par-iLCP (T_{40})	0.115	0.41	15.8	0.12	0.196	4.85	0.354	0.384	0.355	2.03	0.3	0.31	3.31	0.51	0.243	0.261
par-iLCP (T_1/T_{40})	25.8	22.6	25.8	20.8	15.9	18	18.9	20.3	19.1	24.3	15.5	18	18.9	22.2	4	6.9
par-LCP (T_1)	2.29	6.5	311	2.12	2.93	76.2	5.61	5.95	5.63	30.2	4.22	4.76	55.9	7.31	0.568	1.91
par-LCP (T_{40})	0.144	0.44	14.2	0.138	0.215	4.88	0.388	0.389	0.359	1.93	0.31	0.312	3.32	0.481	**0.119**	**0.179**
par-LCP (T_1/T_{40})	15.9	14.8	21.9	15.4	13.6	15.6	14.5	15.3	15.7	15.6	13.6	15.3	16.8	15.2	4.8	10.7
par-PLCP (T_1)	1.68	5.51	233	1.66	2.56	58.8	4.78	5.16	4.84	25.1	3.85	4.07	44.1	6.98	0.767	1.58
par-PLCP (T_{40})	**0.083**	**0.31**	**10.7**	**0.095**	**0.173**	**3.89**	**0.293**	**0.31**	**0.287**	1.42	**0.268**	**0.251**	**2.73**	0.343	0.143	0.186
par-PLCP (T_1/T_{40})	20.2	17.8	21.8	17.5	14.8	15.1	16.3	16.6	16.9	17.7	14.4	16.2	16.2	20.3	5.4	8.5
dk-LCP (T_1)	3.25	9.32	384	2.98	4.01	106	7.76	8.37	7.83	55.1	5.83	6.63	76.4	10.5	1.06	3.1
dk-LCP (T_{40})	0.195	0.606	20.1	0.185	0.265	6.51	0.523	0.535	0.495	2.7	0.389	0.406	4.56	0.663	0.212	0.301
dk-LCP (T_1/T_{40})	16.7	15.4	19.1	16.1	15.1	16.3	14.8	15.6	15.8	20.4	15	16.3	16.8	15.8	5	10.3
dk-PLCP (T_1)	2.06	6.78	328	1.99	2.99	71.7	5.68	6.23	5.81	31	4.35	4.79	52.1	7.55	1.14	1.98
dk-PLCP (T_{40})	0.107	0.386	13.3	0.117	0.196	4.47	0.34	0.358	0.335	1.77	0.302	0.306	3.16	0.446	0.227	0.236
dk-PLCP (T_1/T_{40})	19.3	17.6	24.7	17	15.3	16	16.7	17.4	17.3	17.5	14.4	15.7	16.5	16.9	5	8.4

TABLE II: Running times (seconds) of the LCP algorithms on different inputs on a 40-core machine with hyper-threading. Our new algorithms are shown in bold font. T_1 is the time using a single thread, T_{40} is the time using 40 cores (80 hyper-threads), and T_1/T_{40} is the parallel speedup . The numbers in bold indicate the fastest parallel LCP running time for an input among all implementations. The entries labeled "–" indicate that the experiment did not finish running in a reasonable amount of time.

We implement all of the algorithms listed in Table I, and as a reminder, among the parallel LCP algorithms compared, par-LCP, par-PLCP, dk-PLCP and skew-LCP, and par-iLCP are new, and naive-LCP, dk-LCP and skew-SA+LCP are existing algorithms. We list the main findings of our experimental study:

1) On a 40-core Intel Nehalem machine with two-way hyper-threading, par-PLCP achieves the best parallel running times for most real-world inputs. It is 1.5–2.3x faster than the existing parallel LCP algorithm of Deo and Keely [6].
2) While skew-LCP has better worst-case theoretical guarantees than par-PLCP, it is 6–11x slower in practice.
3) For real-world inputs, the performance of par-LCP, par-PLCP, dk-LCP and dk-PLCP is quite robust to the choice of the parameter K as long as K is not too extreme.
4) par-PLCP achieves good parallel speedup relative to kmp-LCP, the fastest sequential LCP algorithm.
5) All of the parallel algorithms achieve good self-relative speedup on most inputs.
6) In parallel, computing the SA and LCP arrays separately is faster than computing them together with the skew algorithm.
7) Comparing the two parallel LCP algorithms which require $O(n)$ work and poly-logarithmic depth, in parallel skew-LCP is 1.4–2x times faster than the original skew algorithm.

Implementations. We implement the parallel algorithms in Cilk Plus [25] using the `cilk_for` construct to express parallel for-loops.

In our implementation of par-LCP and par-PLCP, we set K equal to the number of available threads P (except for the experiment in Fig. 10). Therefore the interval size is at most $\lfloor n/P \rfloor$ and number of intervals is either P or $P+1$. In practice, we found this to give the best balance between the extra work spent in computing the lcp values for the first element of each chunk and the amount of parallelism. We also implemented the modified versions of par-LCP and par-PLCP using random shifting as discussed in Lemma 3.2, but did not find an improvement over the original versions. This is because in the original versions, the work for computing the first element of each interval is usually much lower than $l_{\max}$ in practice.

We implement the dk-LCP and dk-PLCP algorithms using the parallel filter code from the Problem Based Benchmark Suite (PBBS) [40]. We set $K = 2P$ (except for the experiment in Fig. 10) and split each interval with size greater than $\lfloor n/K \rfloor$ into sub-intervals of size $\lfloor n/K \rfloor$, except for the last sub-interval, which may be smaller. For single-threaded execution we set $K = 1$. We found this setting to give the best performance across all inputs. Note that the value of K here is higher than in par-LCP and par-PLCP. This is because the sizes of the intervals in dk-LCP and dk-PLCP vary more, and creating more tasks (intervals) gives more flexibility to the run-time scheduler to achieve better load-balancing.

Our implementation of par-iLCP uses the parallel filter code from the PBBS, and the for-loop over the indices between two irreducible values is only parallelized when the size is greater than 1000 (to avoid the overhead of a parallel for-loop for smaller sizes). We also implement the naive parallel LCP algorithm (naive-LCP) from Fig. 2.

Since the loops in the above algorithms are so simple, the performance is mainly determined by the number of cache misses. We did not see any way to further optimize the code.

We implement skew-LCP, the standalone LCP algorithm described in Section III, by making the necessary modifications to the parallel implementation of the skew algorithm from the

PBBS [40]. Our implementations of the sequential klaap-LCP and kmp-LCP algorithms follow the pseudo-code shown in Fig. 3 and 4, respectively.

We note that Gog and Ohlebusch [10] describe a sequential LCP algorithm that requires the Burrows-Wheeler transform array as input. Its implementation [9] uses compressed integers and are semi-external, leading to lower space usage but higher running time, and hence it is difficult to perform a direct comparison with our internal memory implementations that do not use compressed integers.

Experimental Setup. We run our experiments on a 40-core (with hyper-threading) machine with 4×2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache), and 256GB of main memory. We run all parallel experiments with two-way hyper-threading enabled, for a total of 80 threads. We compile the code with `g++` version 4.8.0 (which supports Cilk Plus) with the `-O2` flag. The times that we report are based on a median of three trials.

For running the experiments we use a variety of strings available online[4], XML code from Wikipedia samples (*wikisamp8* and *wikisamp9*), human genomic data[5] (*HG18.fasta*), protein data[6] (*proteins*), short reads of a DNA sequence[7] (*Venter0*) and artificial inputs. Our artificial inputs are all of size 10^8 and include a random string with an alphabet size of 10 (*random*), an all identical string (*identical*), and a binary string where every 10^4'th position contains one character and all other positions contain the other character (*sqrtn*). We use one byte to represent each character for all inputs. Table II shows the file size, alphabet size ($|\Sigma|$), maximum lcp value ($l_{\max}$) and average lcp value (l_{avg}) for each input.

Comparison of LCP algorithms. Table II shows the single-threaded (T_1), 40-core (T_{40}) times, and parallel speedups (T_1/T_{40}) for all of the standalone LCP implementations. The fastest parallel time per input in Table II is shown in bold.

Firstly, we look at the performance of naive-LCP, the naive parallel algorithm. As expected, naive-LCP performs relatively well for the inputs with small average lcp values, but significantly worse for the inputs with large lcp values. For Venter0 and the random string, naive-LCP performs the best among all implementations due to the small lcp values. For several inputs we do not report numbers for naive-LCP as it did not finish in a reasonable amount of time due to the large lcp values.

Fig. 9 shows a bar chart comparing the running times for the parallel implementations using 80 hyper-threads on several inputs (for clarity of presentation, naive-LCP is not included as it is an order of magnitude slower on some inputs). From Table II and Fig. 9, we see that *par-PLCP performs the fastest on most of the inputs*. We do see some exceptions, however. For the identical and sqrtn strings, par-LCP performs the best. This is because most contiguous suffixes in the suffix array also appear contiguously in the original string, and thus most memory accesses are cache-friendly. par-PLCP was designed to reduce random accesses at the cost of an extra phase to convert the PLCP array into the LCP array so this makes it slower than par-LCP for these two strings. For Venter0, which has small lcp values, par-PLCP performs almost as fast as naive-LCP. For the random string, which has even smaller lcp values, par-PLCP is about two times slower than naive-LCP. However, even though par-PLCP is not the fastest on these inputs, it still performs reasonably well. For all of the other inputs, par-PLCP is the fastest in parallel, so without prior knowledge about an input, par-PLCP will likely give the best performance.

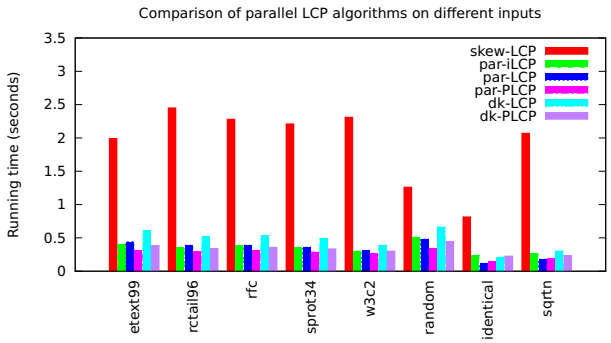

Fig. 9: Comparison of running times of parallel LCP algorithms using 40 cores (80 hyper-threads).

We see that *skew-LCP is 6–11 times slower than par-PLCP*, even though it requires $O(n)$ work and $O(\log^2 n)$ depth in the worst-case, which is better than the worst-case complexity of par-PLCP. This is because the constants in its work bound are higher than for par-PLCP, and the extra work in computing the first element of each interval in par-PLCP (the $O(Kl_{\max})$ term) is not high in practice. For par-iLCP, although it is not the fastest on any input, it is at most 3 times slower than the fastest implementation. Furthermore, it always outperforms skew-LCP. This is likely because for most inputs, the amount of work performed is less than its worst-case bound of $O(n \log n)$.

We note that par-PLCP is overall faster than par-LCP, and dk-PLCP is overall faster than dk-LCP. This is consistent with the study of sequential LCP implementations by Kärkkäinen et al. [19], showing that kmp-LCP is faster than klaap-LCP.

We also observe that in parallel par-LCP outperforms dk-LCP by 23–78%, and par-PLCP outperforms dk-PLCP by 13–59%. dk-LCP and dk-PLCP guarantee that the elements with an lcp value of 0 are at the beginning of intervals with the goal of performing less wasted work compared to the corresponding sequential algorithm. However, it requires a pre-processing phase to identify the indices of elements for which the lcp value is 0 using a parallel filter. Therefore the overall time becomes slower than that of par-LCP and par-PLCP, which simply work on equal-sized chunks. *Compared to dk-LCP, our implementation of the only existing parallel standalone LCP algorithm, our fastest LCP algorithm par-PLCP is 1.5–2.3x faster on 40 cores with hyper-threading*.

For further analysis, in the Appendix, we show breakdowns of the parallel running times for par-LCP, par-PLCP, dk-LCP and dk-PLCP for several inputs, and also discuss the space usage of the parallel LCP algorithms.

Varying K. In the complexity bounds of par-LCP, par-PLCP, dk-LCP and dk-PLCP, the parameter K represents a trade-off between work and parallelism. To see how it affects performance in practice, we measure the parallel running times as we vary K. Fig. 10 shows the running time of the four implementations using 40 cores (80 hyper-threads) as a function of K for etext99 and wikisamp8. For par-LCP and par-PLCP, the interval size is $\lfloor n/K \rfloor$, except for possibly the last interval. For dk-LCP and dk-PLCP, the number of intervals beginning

[4] http://people.unipmn.it/manzini/lightweight/corpus/
[5] http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz
[6] http://pizzachili.dcc.uchile.cl/texts/protein/
[7] ftp://ftp.ncbi.nih.gov/pub/TraceDB/Personal_Genomics/Venter/

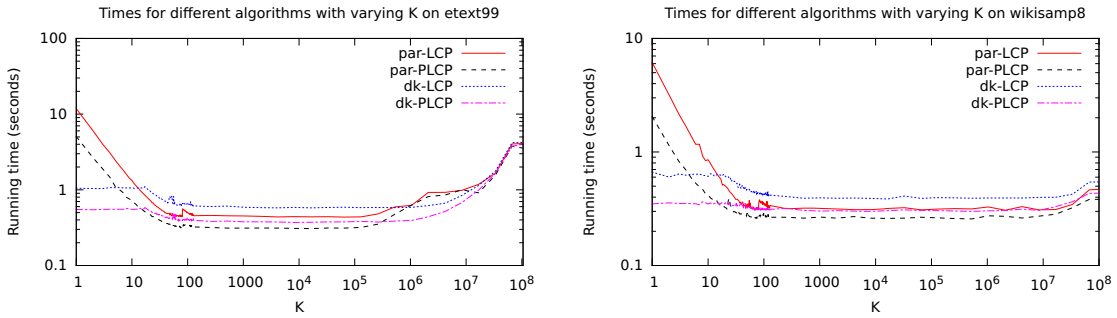

Fig. 10: Parallel running times versus K for different algorithms on etext99 (left) and wikisamp8 (right). The y-axis is in log-scale.

with an lcp value of 0 is fixed (at most $|\Sigma|$), but we divide each interval larger than size $\lfloor n/K \rfloor$ into sub-intervals of size $\lfloor n/K \rfloor$, except for possibly the last sub-interval.

For small values of K (less than 10) all of the implementations do not exhibit much parallelism, but dk-LCP and dk-PLCP benefit from performing less wasted work in the main loop. For larger values of K there is enough parallelism and par-LCP and par-PLCP are faster due to not requiring a parallel filter. We see that for these inputs, *the performance of the algorithms is quite robust across different values of K as long as it is not too small or too large.* We observed similar behavior for the other real-world inputs.

Comparing to sequential. As shown in Table II, on a single thread, par-LCP and par-PLCP do just as well as klaap-LCP and kmp-LCP, respectively. This is because in our implementations, when there is only a single thread, only one interval is used ($K = 1$) and the parallel implementations do the same amount of work as their sequential counterparts. The speedup curves of par-PLCP with respect to kmp-LCP for several inputs are plotted in Fig. 11. Compared to the sequential kmp-LCP code, par-PLCP achieves a speedup of 14.4–20.3x for the inputs in Fig. 11 (and 21.8x for HG18.fasta). For the identical and sqrtn strings, the speedups are only 5.4x and 8.5x, respectively, since the parallel version does much more work than the sequential version due to the large lcp values; the speedup comes from the parallelism in generating the Φ array and converting the PLCP array to the LCP array.

Note that since we vary K based on the number of threads available, the amount of work done at each data point is not the same. In particular, with more threads we have more intervals, leading to more work compared to a single-threaded execution. Adjusting K is done to minimize the work, while taking advantage of all of the available parallelism. This, however, limits the speedup compared to a sequential execution. If we were to keep K constant for different numbers of threads, the amount of work would be the same for all thread counts, and we would see better speedups up to about K threads as each thread would have an interval to process.

Self-relative speedup. *All of the parallel implementations achieve good self-relative speedup on the real-world inputs.* For the implementations whose work is independent of the number of threads, on 80 hyper-threads, naive-LCP, skew-LCP and par-iLCP achieve speedups of up to 51.5x, 40.8x and 25.8x respectively (see Table II). par-iLCP does not achieve good speedups on the identical and sqrtn strings as the available parallelism is low due to the large lcp values. As for the implementations whose work varies with thread count (par-LCP, par-PLCP, dk-LCP and dk-PLCP), the self-relative speedups

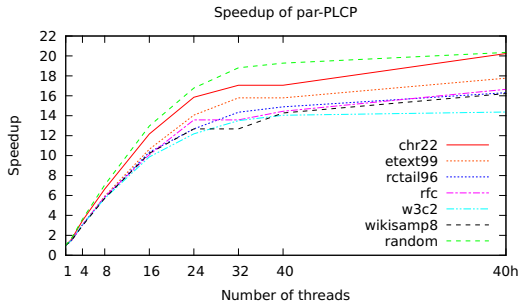

Fig. 11: Speedup of par-PLCP with respect to kmp-LCP. (40h) indicates 80 hyper-threads.

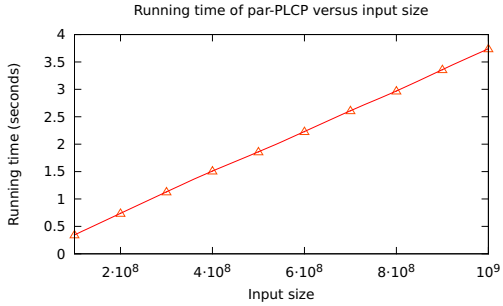

Fig. 13: Running time versus input size of random text for par-PLCP using 40 cores (80 hyper-threads).

are lower, ranging from 13.6x to 24.7x on the real-world inputs. Again, these implementations do not get good speedup on the identical and sqrtn strings due to the low parallelism. Since the implementations perform many random memory accesses, the speedups are also likely limited by the memory bandwidth of the machine and the latency associated with memory contention.

Varying thread count. Fig. 12 shows the running time as a function of thread count for the different LCP implementations on etext99 and wikisamp8. Except for naive-LCP and skew-LCP, all of the parallel implementations outperform the best sequential implementation (kmp-LCP) with 4 or more threads.

Varying input size. To show scalability with increasing input size, we ran par-PLCP on random strings of varying sizes ($|\Sigma| = 10$). In Fig. 13 we plot the 40-core running time of par-PLCP as a function of input size. We observe that the running time scales linearly with the input size.

A. Performance of suffix array and LCP construction

In addition to studying the performance of the LCP algorithms on their own, we also study the overall performance of suffix array and LCP construction. We show that in the parallel setting, separating suffix array and LCP construction leads to

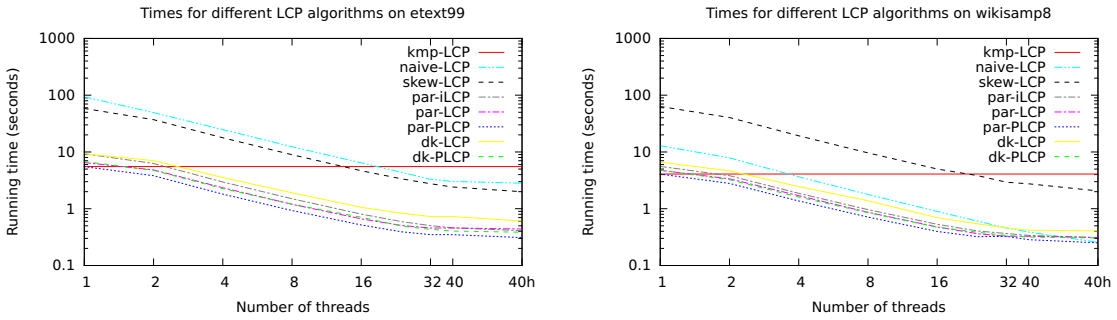

Fig. 12: Running times versus number of threads of LCP algorithms on etext99 (left) and wikisamp8 (right) in log-log scale. (40h) indicates 80 hyper-threads.

	chr22	etext99	HG18.fasta	howto	jdk13c	proteins	rctail96	rfc	sprot34	Venter0	w3c2	wikisamp8	wikisamp9	random	identical	sqrtn
divsufsort-SA (seq.)	4.21	17.3	†	4.65	8.48	268.5	16.6	15	15.7	83.7	13.2	14.6	190.9	20.7	0.62	1.69
range-SA (T_1)	6.67	38.5	1120	12.2	33.6	533	51.4	42.7	39.4	97	74	36.3	411	16	130	119
range-SA (T_{40})	**0.738**	**3.01**	**95**	**1.05**	2.45	**34.6**	3.82	**3.22**	**3.09**	**7.62**	5.01	3.19	**32.2**	**1.26**	9.01	6.81
skew-SA (T_1)	15.2	57.8	2020	19.5	34.6	736	59.4	60.9	57.7	214	55.1	50.4	555	34	14.6	19.8
skew-SA (T_{40})	0.931	3.26	97.4	1.16	**1.98**	39.3	**3.41**	3.48	3.31	12.7	**3.22**	**2.99**	32.8	1.99	**1.07**	**1.63**
divsufsort-SA + kmp-LCP (seq.)	5.88	22.8	†	6.32	11	327.4	21.4	20.1	20.5	110	16.9	18.7	234.7	26.7	1.35	3.26
skew-SA+LCP (T_{40})	1.15	4.01	122	1.44	2.71	50.8	4.56	4.48	4.28	16	4.34	3.98	43.1	2.48	1.45	2.84
parallel-SA + par-PLCP (T_{40})	**0.82**	**3.32**	**105.7**	**1.15**	**2.15**	**38.5**	**3.7**	**3.53**	**3.38**	**9.04**	**3.49**	**3.24**	**34.9**	**1.6**	**1.21**	**1.82**

TABLE III: *Top:* Running times (seconds) of SA algorithms on a single thread (T_1) and on 40 cores with hyper-threading (T_{40}). The numbers in bold indicate the fastest parallel SA running time for an input. *Bottom:* Running times (seconds) of the various SA+LCP combinations. The numbers in bold indicate the fastest parallel SA+LCP running time for an input. *Note:* The entries labeled † indicate that the implementation failed to run. (Refer to Table II for input statistics.)

performance improvements in practice over constructing both arrays together. We first discuss the suffix array algorithms that we use and then discuss the performance when combined with LCP algorithms.

Performance of suffix array algorithms. In Table III, we report the times for suffix array computation using the fastest available parallel algorithms, skew-SA and range-SA, which are part of the PBBS [40]. *skew-SA* is the parallel implementation of the skew algorithm that does not compute the LCP array. *range-SA* is a parallel algorithm based on the prefix-doubling idea of sorting prefixes of suffixes with the prefix sizes increasing in powers of two. This idea has been used in several sequential suffix array algorithms [33] and also in parallel suffix tree algorithms [16]. range-SA requires $O(n \log n)$ work in the worst-case and does not generate the LCP array. We also report the times for standalone suffix array construction in Table III using the fastest available sequential algorithm (*divsufsort-SA*) implemented by Mori [29]. Mori also provides a parallel implementation of divsufsort-SA using OpenMP [29], however we were unable to obtain any speedup compared to the corresponding sequential implementation.

The fastest parallel suffix array time per input is shown in bold in Table III, and we see that in parallel there is no clear winner between range-SA and skew-SA. Compared to the sequential divsufsort-SA, the best parallel implementation achieves a speedup of 4.1–11x on the real-world inputs. On the random string, it achieves a 16.4 fold speedup over divsufsort-SA, while for the identical and sqrtn strings, it performs about the same or worse, as the two parallel implementations are not well-suited for inputs with a lot of repeated structure. divsufsort-SA is faster than both range-SA and skew-SA on a single thread for all inputs except HG18.fasta, on which it failed to run, and the random string, on which it loses to range-SA.

Generating both the suffix array and LCP array. In Table III, we report the times for computing both the suffix array and the LCP array. For sequential times, we report the time for divsufsort-SA followed by kmp-LCP (*divsufsort-SA + kmp-LCP*). We also tried the implementation of Fischer's sequential algorithm [8, 30] which generates both the suffix array and LCP array, but found it to be slower than divsufsort-SA followed by kmp-LCP for all of our inputs. For parallel times, we report the time for the parallel skew algorithm from the PBBS [40] that generates both the suffix array and LCP array (*skew-SA+LCP*) and also the time for running the fastest parallel suffix array algorithm for the input followed by par-PLCP (*parallel-SA + par-PLCP*).

In parallel, the faster parallel SA algorithm followed by par-PLCP always outperforms skew-SA+LCP, with a speedup factor ranging from 1.1 to 1.8, confirming that *separating LCP construction from the suffix array construction leads to improved performance in the parallel setting*. Separating the construction of the two arrays allows us to use a faster parallel SA algorithm that does not compute the lcp values followed by a fast LCP algorithm. Furthermore, improvements in either parallel SA algorithms or parallel LCP algorithms leads to an overall performance improvement in the construction process.

The improvement in the parallel running time of SA and LCP array construction improves the overall running time of parallel applications that require SA and LCP, such as suffix tree construction [38] and Lempel-Ziv factorization [39]. The improvements are significant as the SA + LCP computation is the dominant part of the computation in these applications (at least 80% of the total running time).

Compared to the sequential method of applying divsufsort-SA followed by kmp-LCP, applying the faster parallel suffix array algorithm followed by par-PLCP achieves a speedup of 4.8–12.2x on 40 cores for the real-world inputs. For the random string, the speedup is 16.7x, which is higher than for the real-world inputs, due to the good speedup of the parallel suffix array algorithm. For the identical and sqrt strings, the speedup

is less than 2x mostly due to the poor speedup of the parallel suffix array algorithm.

Linear work and poly-logarithmic depth algorithms. We also compare skew-SA+LCP and skew-LCP, the two LCP algorithms with linear work and poly-logarithmic depth without dependence on the lcp values of the suffixes of the input. From Tables II and III, we see that *in parallel, skew-LCP outperforms skew-SA+LCP by 1.8–2x for real-world inputs and 1.4–2x for artificial inputs.*

V. Conclusion

We have presented and analyzed new parallel algorithms for computing the longest common prefix of a string given its suffix array. We have also presented a comprehensive experimental analysis of various LCP algorithms on a collection of real-world and artificial texts, showing that our fastest algorithm is faster than the previous parallel algorithms for LCP computation [6, 17]. Directions for future work include designing LCP algorithms with improved work/depth bounds, and experimenting on a machine with a much larger number of cores where non-uniform memory accesses would be a bigger concern. We are also interested in extending our LCP algorithms to the external-memory or distributed-memory settings. For example, our par-PLCP algorithm could be used in conjunction with a recent external-memory PLCP algorithm [18] to obtain a parallel disk-based algorithm. To adapt the algorithms to a distributed-memory setting, the main challenge would be to reduce the number of random accesses to the various arrays, as these would likely require communication between processors.

Acknowledgments

This work is supported by a Facebook Graduate Fellowship, the National Science Foundation under grant number CCF-1314590, and the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program. We thank Guy Blelloch for helpful discussions and numerous suggestions to help improve the paper, including the randomization idea discussed in Lemma 3.2. We also thank Simon Gog and Timo Beller for explaining the details of the LCP implementations of [9, 10]. In addition, we thank Harsha Simhadri for helpful discussions regarding the skew algorithm. Finally, we are grateful for the helpful comments from the anonymous reviewers.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, Mar. 2004.

[2] M. J. Bauer, A. J. Cox, G. Rosone, and M. Sciortino, "Lightweight LCP construction for next-generation sequencing datasets," in *Workshop on Algorithms in Bioinformatics (WABI)*, 2012, pp. 326–337.

[3] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger, "Computing the longest common prefix array based on the Burrows-Wheeler transform," *J. of Discrete Algorithms*, vol. 18, pp. 22–31, 2013.

[4] T. Bingmann, J. Fischer, and V. Osipov, "Inducing suffix and LCP arrays in external memory," in *Algorithm Engineering and Experiments (ALENEX)*, 2013, pp. 88–102.

[5] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," HP Labs, Tech. Rep., 1994.

[6] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the GPU," in *Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 197–206.

[7] M. Farach and S. Muthukrishnan, "Optimal logarithmic time randomized suffix tree construction," in *International Colloquium on Automata, Languages and Programming (ICALP)*, 1996, pp. 550–561.

[8] J. Fischer, "Inducing the LCP-array," in *International Conference on Algorithms and Data Structures (WADS)*, 2011, pp. 374–385.

[9] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Symposium on Experimental Algorithms (SEA 2014)*, 2014, pp. 326–337.

[10] S. Gog and E. Ohlebusch, "Fast and lightweight LCP-array construction algorithms," in *Algorithm Engineering and Experiments (ALENEX)*, 2011, pp. 25–34.

[11] ——, "Compressed suffix trees: Efficient computation and storage of LCP-values," *J. Exp. Algorithmics*, vol. 18, May 2013.

[12] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, "Information retrieval," 1992, ch. New Indices for Text: PAT Trees and PAT Arrays, pp. 66–82.

[13] D. Gusfield, *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[14] T. Hagerup and R. Raman, "Waste makes haste: tight bounds for loose parallel sorting," in *Foundations of Computer Science (FOCS)*, 1992, pp. 628–637.

[15] R. Hariharan, "Optimal parallel suffix tree construction," in *Symposium on Theory of Computing (STOC)*, 1994, pp. 290–299.

[16] J. Jaja, *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[17] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *International Colloquium on Automata, Languages and Programming (ICALP)*, 2003, pp. 943–955.

[18] J. Kärkkäinen and D. Kempa, "LCP array construction in external memory," in *Symposium on Experimental Algorithms (SEA)*, 2014, pp. 412–423.

[19] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Combinatorial Pattern Matching (CPM)*, 2009, pp. 181–192.

[20] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *J. ACM*, vol. 53, no. 6, pp. 918–936, Nov. 2006.

[21] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn, "New approaches for computer analysis of nucleic acid sequences," *Natl. Acad. Sci. USA*, vol. 80, pp. 5660–5664, 1993.

[22] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Combinatorial Pattern Matching (CPM)*, 2001, pp. 181–192.

[23] D. Kim, J. Sim, H. Park, and K. Park, "Linear-time construction of suffix arrays," in *Combinatorial Pattern Matching (CPM)*, 2003, pp. 186–199.

[24] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," in *J. of Discrete Algorithms*, vol. 3, 2005, pp. 143–156.

[25] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, 2010.

[26] F. A. Louza, G. P. Telles, and C. D. D. A. Ciferri, "External memory generalized suffix and LCP arrays construction," in *Combinatorial Pattern Matching (CPM)*, 2013, pp. 201–210.

[27] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[28] G. Manzini, "Two space saving tricks for linear time LCP array computation," in *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, 2004, pp. 372–383.

[29] Y. Mori, "libdivsufsort: A lightweight suffix-sorting library," 2010, http://code.google.com/p/libdivsufsort.

[30] ——, "sais: An implementation of the induced sorting algorithm," 2010, http://sites.google.com/site/yuta256/sais.

[31] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.

[32] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Data Compression Conference (DCC)*, 2009, pp. 193–202.

[33] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, Jul. 2007.

[34] S. J. Puglisi and A. Turpin, "Space-time tradeoffs for longest-common-prefix array computation," in *International Symposium on Algorithms and Computation (ISAAC)*, 2008, pp. 124–135.

[35] S. Rajasekaran and J. H. Reif, "Optimal and sublogarithmic time randomized parallel sorting algorithms," *SIAM J. Comput.*, vol. 18, no. 3, pp. 594–607, 1989.

[36] K. Sadakane, "Succinct representations of lcp information and improvements in the compressed suffix arrays," in *Symposium on Discrete Algorithms (SODA)*, 2002, pp. 225–232.

[37] S. Sahinalp and U. Vishkin, "Symmetry breaking for suffix tree construction," in *Symposium on Theory of Computing (STOC)*, 1994, pp. 300–309.

[38] J. Shun and G. E. Blelloch, "A simple parallel cartesian tree algorithm and

its application to parallel suffix tree construction," in *ACM Transactions on Parallel Computing (TOPC)*, 2014.
[39] J. Shun and F. Zhao, "Practical parallel Lempel-Ziv factorization," in *Data Compression Conference (DCC)*, 2013, pp. 123–132.
[40] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the Problem Based Benchmark Suite," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012, pp. 68–70.
[41] J. Sirén, "Sampled longest common prefix array," in *Combinatorial Pattern Matching (CPM)*, 2010, pp. 227–237.
[42] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, Aug. 1990.

Appendix

Timing breakdowns. To understand where the running times of par-LCP, par-PLCP, dk-LCP and dk-PLCP are coming from, we show breakdowns of the parallel running times for the four implementations on several inputs in Fig. 14. For the real-world inputs, par-LCP spends about 25–40% of its time in computing the Rank array (Lines 2–3 of Fig. 6), which performs n random writes. However for the identical string, only 14% of the time is spent in this loop since the suffix array stores the indices in reverse order and hence the writes are cache-friendly. For the real-world inputs, par-PLCP spends about 40–50% of its time in computing the Φ array (Lines 3–4 of Fig. 7), which also performs n random writes. Again, for the identical string, the writes are cache-friendly so this phase takes only 12% of the time. The main loop of par-PLCP (Lines 5–16 of Fig. 7) is cheap for the real-world inputs and the random string (10–33% of the total time). For the identical string, it takes two-thirds of the time since the other two phases are very cheap due to good locality of the suffix ordering, and the main loop does much more work due to the large lcp values.

For dk-LCP and dk-PLCP, computing the indices corresponding to an lcp value of 0 takes about 15–30% of the total time, and this step is not needed by par-LCP and par-PLCP. Without including the time for computing the indices, the breakdowns of dk-LCP and dk-PLCP are similar to those of par-LCP and par-PLCP, respectively. We found that the improvement in the main loop from identifying the indices corresponding to an lcp value of 0 was insignificant. Therefore, overall dk-LCP and dk-PLCP are slower than par-LCP and par-PLCP, respectively.

Space Usage. The space usage of par-LCP and par-PLCP are the same as the corresponding sequential implementations. In particular, assuming that $n < 2^{32}$ (the integer arrays use 4 bytes per element), par-LCP requires $13n$ bytes—$4n$ bytes for each of LCP, SA and Rank and n bytes for the string S. par-PLCP also requires $13n$ bytes—$4n$ bytes for each of LCP, SA and Φ and n bytes for S. The array Φ can be reused to store PLCP. dk-LCP and dk-PLCP require an array to store the indices with a corresponding lcp value of 0, in addition to the $13n$ bytes of the corresponding sequential implementation. The parallel filter code of the PBBS [40] uses two integer arrays (one array stores the indices computed) and a boolean array of size n. The integer array not storing the indices can be reused as the LCP array, so the peak space usage is $18n$ bytes (the other arrays need to be computed before and used during the filter). par-iLCP also uses $18n$ bytes, as it uses a parallel filter to compute and store the indices of the irreducible lcp values, in addition to arrays used in par-PLCP. naive-LCP requires the least space, as it only needs S, SA and LCP for a total of $9n$ bytes. skew-LCP uses significantly more space than the other implementations as it uses several additional arrays for auxiliary information, some of which are kept during recursion, and also a range minima data structure. In Fig. 15, we report the actual space usage of the LCP algorithms on several inputs, measured using the `massif` program from Valgrind [31]. The results agree with our analysis of space usage above.

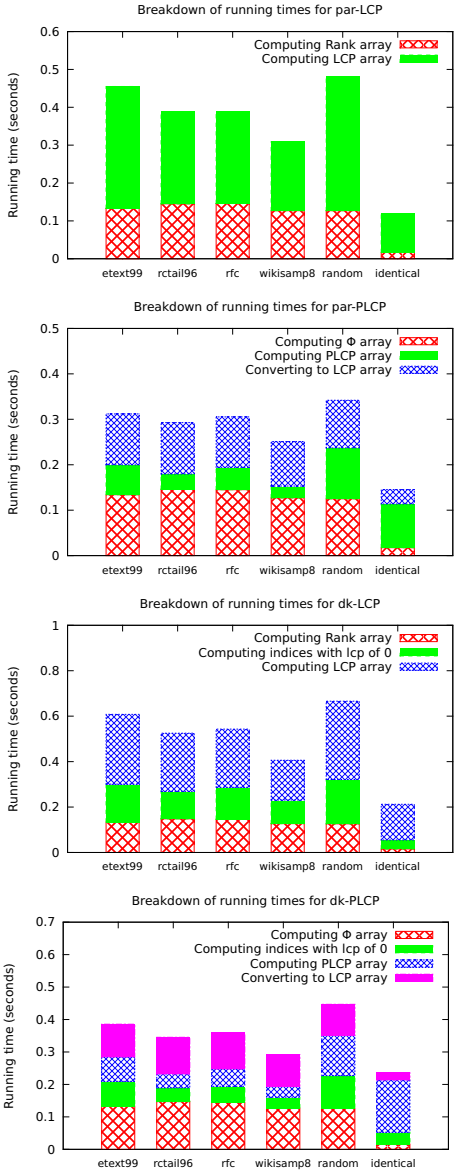

Fig. 14: Breakdown of running times on 40 cores with hyper-threading.

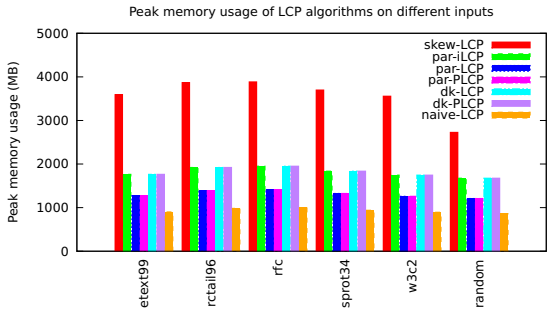

Fig. 15: Peak memory usage (megabytes) of LCP algorithms.