Fast, Parallel, and Cache-Friendly Suffix Array Construction

Jamshed Khan
University of Maryland, College Park, MD, USA

Tobias Rubel
University of Maryland, College Park, MD, USA

Laxman Dhulipala
University of Maryland, College Park, MD, USA

Erin Molloy
University of Maryland, College Park, MD, USA

Rob Patro
University of Maryland, College Park, MD, USA

Abstract

String indexes such as the suffix array (SA) and the closely related longest common prefix (LCP) array are fundamental objects in bioinformatics and have a wide variety of applications. Despite their importance in practice, few scalable parallel algorithms for constructing these are known, and the existing algorithms can be highly non-trivial to implement and parallelize. In this paper we present CaPS-SA, a simple and scalable parallel algorithm for constructing these string indexes inspired by samplesort. Due to its design, CaPS-SA has excellent memory-locality and thus incurs fewer cache misses and achieves strong performance on modern multicore systems with deep cache hierarchies. We show that despite its simple design, CaPS-SA outperforms existing state-of-the-art parallel SA and LCP-array construction algorithms on modern hardware. Finally, motivated by applications in modern aligners where the query strings have bounded lengths, we introduce the notion of a bounded-context SA and show that CaPS-SA can easily be extended to exploit this structure to obtain further speedups.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Suffix Array, Longest Common Prefix, Data Structures, Indexing, Parallel Algorithms

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.16

Supplementary Material *Software (Source Code)*: `https://github.com/jamshed/CaPS-SA`
archived at `swh:1:dir:a503464134894c9067090bc3f65c04308842c5bb`

Funding *Rob Patro*: supported by the NIH under grant award numbers R01HG009937 and by NSF awards CCF-1750472 and CNS-176368.

1 Introduction

Methods for aligning sequencing reads to reference genomes underlie some of the most well-developed and widely-used tools in bioinformatics [2]. Modern read-to-reference aligners typically employ an *index* over the reference text. A classic index for strings is the suffix array (SA) [35], which is an array of indices of the lexicographically sorted suffixes of a string. In alignment, the SA index is used by the popular STAR aligner [12] as well as in other tools [48, 50]. The SA has also been used in short-read error correction [20] and sequence clustering [19]. A related object frequently used in conjunction with the SA is the Longest Common Prefix (LCP) array, which contains the lengths of the longest shared prefixes between pairs of successive indices in the SA. For instance, the SA can be used

in concert with the LCP-array (and other auxiliary tables derived from these) in a data structure called an enhanced suffix array [1] to mimic the functionality of a suffix tree [49], but often more efficiently and using less space. An account of the pervasiveness of the SA and the LCP-array in computational genomics is best left to a dedicated review (see e.g. [46]).

Because of the utility of the SA (and the LCP-array) in string indexing, significant work has been dedicated to developing practical algorithms for its construction. It is well-established that SA and LCP-array construction can be performed sequentially in time linear to the size of strings. However, as modern genomics pipelines produce ever more data – including more complete reference genomes and pangenomes – there has been a concerted effort to improve the practical efficiency and reduce the runtime of SA and LCP-array construction. A host of efficient serial algorithms have been developed [30, 26, 29, 37, 14, 39, 33]. Likewise, in an effort to take advantage of the increased parallelism of modern computer hardware, a number of parallel algorithms have also been proposed; e.g. parallel DivSufSort [32], parallel DC3 [31], and parallel divide-and-conquer based SA-construction [24]. External memory algorithms [22, 25, 23] have also been a focus of recent research because of the memory bottlenecks that arise when building the SA and the LCP-array on genomic datasets. Besides, algorithms for GPU-settings [34] and distributed-memory [15, 16] have been developed. We refer the interested reader to [43, 5, 6] for a comprehensive review. For our purposes, we note that these increasingly advanced methods introduce new algorithmic techniques to enable parallelism or improve the worst-case time complexity (so that it is sublinear). The trade-off, often, is that these more complex algorithms may potentially be more difficult to implement, optimize for modern hardware and cache layouts, and to maintain.

In this work, we address these issues by introducing CaPS-SA, a highly parallel method for constructing the SA and the LCP-array. A core principle behind CaPS-SA is simplicity. Our approach draws on several existing algorithms and techniques, and focuses on their efficient combination for the problem of highly parallel SA construction. The algorithm builds upon the *parallel samplesort* algorithm [17], and is easy to implement and optimize for modern hardware.

A potential downside of our approach is that it is *output-sensitive* and as a result its worst-case time complexity on adversarial inputs is quadratic. However, in practice we find that the shared-memory implementation of CaPS-SA outperforms state-of-the-art methods (specifically Parallel-divsufsort [32] and Parallel-DC3 [31, 3]) in terms of runtime and scalability (although not in memory for Parallel-divsufsort). For example, CaPS-SA can build the SA and the LCP-array for the telomere-to-telomere human genome assembly (CHM13 v2) [40] in 141 seconds using 32 threads on a typical shared-memory machine, whereas the leading method Parallel-divsufsort requires 199 seconds. Our experimental study demonstrates that this superior performance of CaPS-SA can largely be attributed to two causes. First, CaPS-SA achieves better memory-locality (fewer cache misses) than the other methods (likely thanks to its straightforward approach), and second, the worst-case analysis of CaPS-SA does not represent a typical or real world use case. Overall, our work demonstrates that as parallel resources increase combining domain-specific optimizations (i.e. LCP-informed merging) with highly-efficient general sorting strategies (i.e. samplesort [17]) can outperform more sophisticated but complex algorithms. CaPS-SA is implemented in C++17 and is available under an open source license at `https://github.com/jamshed/CaPS-SA`.

The remainder of this manuscript is organized as follows. We discuss the preliminary concepts required for a formal treatment of the algorithm as well as the most relevant prior work and the methods against which we compare CaPS-SA in Sec. 2. Then we discuss CaPS-SA in Sec. 3, and provide an analysis of its asymptotic behavior. Sec. 4 describes the

experimental study for the proposed algorithm, and reports the results. We conclude with discussion on the potential of the method and prospective future directions for building on top of it.

2 Preliminaries

A *string* (or *text*) $T = a_0 a_1 \dots a_{n-1}$ is a finite ordered sequence of n symbols drawn from a finite ordered alphabet Σ. $|T|$ denotes the length n of T. The half-open interval $[i, j)$ is a shorthand for the closed interval $[i \dots j-1]$. T_i denotes the i'th symbol in T. The *substring* $T_{[i,j)}$ of T is the sequence of characters of T on the half-open interval $[i, j)$. We call a substring $T_{[i,j)}$ with $i = 0$ a *prefix* of T. Likewise, A substring $T_{[i,j)}$ with $j = |T|$ is a *suffix* of T, denoted by $T_{[i:]}$.

The ordering of Σ induces a lexicographical ordering of all possible strings over Σ. The *Suffix Array* (SA) of a string T is an array of the starting indices of all suffixes of T ordered by the suffixes' lexicographical order. The *Longest Common Prefix* $\mathrm{LCP}(T_1, T_2)$ of two strings T_1 and T_2 is the largest-sized prefix P of both T_1 and T_2, such that if $|P| = k$ then for all $0 < i < k$, $T_{1i} = T_{2i}$, and $T_{1k} \neq T_{2k}$. Given the suffix array SA of a string T, its LCP-array is the array L such that $L_i = \mathrm{LCP}(T_{[SA_i:]}, T_{[SA_{i-1}:]})$. [1] For instance, given the string $T = \mathrm{AACTGCGGAT\$}$ the SA and LCP array are given by following data structure:

Index	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	T	G	C	G	G	A	T	**$**
SA	10	0	1	8	5	2	7	4	6	9	3
LCP array	0	0	1	1	0	1	0	1	1	0	1

The *work* of an algorithm is the total number of operations it performs to compute the result. The *depth* (or span) of an algorithm is the longest sequence of dependent computations in its execution. In pseudo-code, we will use $(\|)$ as an infix operator to specify the parallel execution of statements – so $f(x) \parallel g(y)$ denotes the parallel execution of $f(x)$ and $g(y)$. We use $\mathcal{O}\big(f(n)\big)$ *with high probability (whp)* in n to mean $\mathcal{O}\big(cf(n)\big)$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

Prior Work. The SA can be constructed naively in $\mathcal{O}(n^2 \log n)$ work for an n-length text. Efficient algorithms can operate in $\mathcal{O}(n)$ work and $\mathcal{O}(n)$ space, which is the theoretical optimum, as it is the time and space required to record the SA and LCP array themselves. A comprehensive discussion of work on SA construction is well beyond the scope of this manuscript. As such, we here focus on several sequential and parallel algorithms which are of particular interest due to their speed and wide use.

The state-of-the-art sequential program for SA construction is `divsufsort` [37, 14]. Subsequent work has elucidated the algorithm to be an efficient implementation of some two-stage algorithms [21, 14].

`Divsufsort` has been parallelized by Labeit et. al. in 2017 [32], and is considered a state-of-the-art tool in parallel SA construction. It has recently been used in several computational genomics tools, including the `CAMMiQ` method for microbial abundance quantification [51] and the `macle` tool for computing match complexity [42].

Another well-known strategy for SA construction is the Difference Cover modulo 3 (DC3) method [27], which has also been effectively parallelized [31].

[1] With the special case of $L_0 = 0$.

Relevant String Sorting Methods. String sorting is a well-studied algorithmic problem. The main difficulty in string sorting is that comparing two strings T_1 and T_2 requires $\mathcal{O}(\min(|T_1|, |T_2|))$ comparisons, which renders many traditional sorting algorithms for atomic objects costly. Of particular relevance to our work is the problem of *merging* two sorted lists of strings. Farach-Colton used an efficient merge-routine for building suffix trees in linear time [13] (though the space overhead of suffix trees renders them impractical for most modern applications). Ng and Kakehi analyzed an efficient merging strategy of sorted lists of strings with associated LCP-information [38]. They show that given random strings with uniform distribution of symbols, an LCP-informed merge-sort algorithm has an expected running time of $\mathcal{O}(n \log n)$ to sort n strings. The same merge procedure was used by Bingmann and Sanders in several samplesorting algorithms for sorting collections of strings [8].

Bingmann and Sanders propose two merge-based string sorting algorithms of particular interest to us here: Parallel Super Scalar String Sample Sort (pS^5) and Parallel Multiway LCP-Mergesort. pS^5 makes use of the merge routine in a samplesort framework, much like CAPS-SA. The algorithms differ in their inputs (a set of strings vs a single string) as well as their approach to partitioning the data. pS^5 uses machine-word-sized pivot keys to create a binary search tree which can fit into the cache of each core, then divides up the input set of strings evenly across the cores and bins them accordingly. As described in Section 3.1, CAPS-SA divides up the input into evenly sized partitions, then samples pivots using a two-step process and places them into each partition. Parallel Multiway LCP-Mergesort generalizes the merge-algorithm to k-way merges [7].

3 Methods

The proposed algorithm, CAPS-SA, is based on the *samplesort* [17] algorithm. Samplesort is a popular generalization of quicksort that achieves excellent performance on both shared-memory and distributed-memory architectures [44, 4]. Instead of partitioning the input array into two parts around a single pivot as in quicksort, it chooses a number of pivots $z_1, z_2, \ldots, z_{p-1}$ along with two sentinel pivots $z_0 = -\infty$ and $z_p = +\infty$, and partitions the data into p partitions such that an input element x_i is assigned to partition j iff $z_{j-1} < x_i \leq z_j$. It then sorts each partition using another (usually sequential) sorting algorithm (e.g., quicksort).

For constructing a suffix array, simply applying samplesort is costly since string comparisons in general require super-constant time. In more detail, first each suffix needs to be assigned to its partition by binary searching over the pivots. Secondly, sorting the suffixes in each partition may cost substantially more than linearithmic time due to string comparisons.

CAPS-SA addresses these issues using the following key idea of *jointly leveraging merge sort and LCP-arrays*. Whenever two suffixes are compared, the comparison is always done inside the operation of merging two sorted arrays of suffixes. Each sorted array is augmented with its LCP-array, and the merge operations avoid repeated comparisons of common prefixes among suffixes by exploiting these LCP-arrays. This approach has previously been used in general string sorting algorithms [38, 8, 7], but to our knowledge has not been leveraged for SA construction. The partitioning strategy for the suffixes is modified to make better use of the merge operation and achieve good parallelism. In particular, instead of randomly sampling pivots at the beginning of the algorithm, CAPS-SA partitions the suffixes uniformly into p subarrays, sorts the subarrays locally, and only then selects the pivots using oversampling. Once pivots are placed within each partition, the p partitions are further subdivided into $p - 1$ subarrays each, for a total of $p(p - 1)$ sub-subarrays. Since

each sub-subarray is flanked by two pivots, the partition that it should go to is known. Each partition is thus a collection of sorted sub-subarrays, which can be merged efficiently. The initial sorting of the uniform-sized subarrays is done using merge-sort to exploit the merge operation. Thus CaPS-SA ends up exploiting an efficient merging procedure with associated LCP-information to reduce expensive comparisons of suffixes, while not having to merge large sub-arrays due to its pivoting strategy. We discuss the algorithm in more detail in the following sections.

3.1 Parallel SA and LCP-array construction: CaPS-SA

Next, we provide a high-level overview of the CaPS-SA(T, p) algorithm. The input to the algorithm is a string T and a partition count (or, *subproblem count*) p, and as output it produces the SA and the LCP-array of T. Conceptually, the algorithm executes in four high-level steps which we illustrate in Figure 1.

```
CaPS-SA(T, p).
 1  SA, SA′, L, L′ = arrays of size |T|

    // populate the suffix array with some permutation of [0, n)
 2  for i ∈ [0, |T|) in parallel
 3      SA_i = SA′_i = i

    // sort p subarrays of uniform size
 4  m = |T|/p
 5  for i ∈ [0, p) in parallel
 6      b = im, e = (i + 1)m // range of the i'th subarray
 7      Merge-Sort(SA′_[b:e), SA_[b:e), L_[b:e), L′_[b:e), T)

    // sample (p − 1) pivots from SA
 8  V = Sample-Pivots(SA, L, T, p)

    // locate each v_j ∈ V in each sorted subarray; for two successive pivots v_{j−1} and v_j,
    // place all suffixes k in each subarray s.t. T_[v_{j−1}:] < T_[k:] ≤ T_[v_j:] in partition j in SA′
 9  (S, R) = Collate-Partitions(SA, SA′, L, L′, V, T, p)
    // S_j: index of partition j in SA′, R_{j,i}: position of pivot v_i in partition j

    // merge the p sorted subarrays in each partition
10  Par-Copy(SA′, SA), Par-Copy(L′, L) // Par-Copy(X, Y) copies X to Y in parallel
11  for j ∈ [0, p) in parallel
12      b = S_j, e = S_{j+1}
13      Merge-Partition(SA′_[b,e), SA_[b,e), p, R_j, L′_[b:e), L_[b:e), T)

    // compute LCP-values at the partition boundaries
14  Compute-Boundary-LCPs(SA, p, S, L, T)

15  return (SA, L)
```

First, it populates an unsorted SA. Then this initial SA is broken into p subarrays of uniform size $|T|/p$, and each subarray is sorted with Merge-Sort, in parallel. Next, $p-1$ global pivots are sampled from the sorted subarrays together. Then in each sorted subarray, in parallel, each pivot is located with a binary search. The locations of the $p-1$ pivots thus found in each sorted subarray break the subarray into p sorted sub-subarrays. Besides, the position of each pivot in the final SA is now defined by its location in each of the p subarrays. The local ordering of the suffixes in each sorted subarray and the global position of the pivots thus define p partitions for the final SA, each of which is a collection of p sub-subarrays: one from each of the p sorted subarrays. Then for each partition, in parallel, its p sorted sub-subarrays are merged recursively into a fully sorted partition. Together, these sorted partitions, in order, produce the final SA and the final LCP-array. The LCP-values for pairs that cross partition boundaries are computed at the end.

```
MERGE(X, Y, L_X, L_Y, Z, L_Z, T).
    // merges the sorted arrays of suffixes (of T) X
    // and Y with LCP arrays L_X and L_Y resp.,
    // into Z; also constructs Z's LCP array in L_Z
 1  i = j = k = 0
 2  m = 0 // LCP of the last compared pair
 3  l_x = 0 // LCP(X_i, X_{i-1})
 4  while i < |X| and j < |Y|
 5      l_x = L_{X i}
 6      if l_x > m
 7          Z_k = X_i, L_{Z k} = l_x, m = m
 8      elseif l_x < m
 9          Z_k = Y_j, L_{Z k} = m, m = l_x
10      else
11          n = m + LCP(T_{[X_i + m :]}, T_{[Y_j + m :]})
12          Z_k = { X_i   if T_{X_i + n} < T_{Y_j + n}
                  { Y_j   otherwise
13          L_{Z k} = { l_x   if Z_k == X_i
                      { m   otherwise
14          m = n

15      if Z_k == X_i
16          i = i + 1
17      else
18          j = j + 1
19          SWAP(X, Y), SWAP(L_X, L_Y), SWAP(i, j)

20      k = k + 1

21  COPY(X_{[i:]}, Z_{[k:]}), COPY(L_{X [i:]}, L_{Z [k:]})

22  if j < |Y|
23      Z_k = Y_j, L_{Z k} = m
24      COPY(Y_{[j+1:]}, Z_{[k+1:]}),
        COPY(L_{Y [j+1:]}, L_{Z [k+1:]})
```

```
MERGE-SORT(X, Y, L, L', T).
    // sorts the array X of suffixes (of T) into Y;
    // constructs the LCP array of sorted X in L
    // using L' as working space;
    // a necessary precondition is X == Y
1   n = |X|
2   if n == 1
3       L_0 = 0
4       return

5   m = n/2
6   MERGE-SORT(Y_{[0:m)}, X_{[0:m)}, L'_{[0:m)}, L_{[0:m)}, T) ||
    MERGE-SORT(Y_{[m:n)}, X_{[m:n)}, L'_{[m:n)}, L_{[m:n)}, T)
7   MERGE(X_{[0:m)}, X_{[m:n)}, L'_{[0:m)}, L'_{[m:n)}, Y, L, T)
```

```
MERGE-PARTITION(X, Y, n, R, L_X, L_Y, T).
    // merges the collection X of n sorted subarrays
    // of suffixes into Y; R contains the ranges of the
    // subarrays in X; L_X is the collection of the
    // LCP arrays of the subarrays, and the LCP array
    // of Y is constructed in L_Y;
    // necessary preconditions are X == Y and L_X == L_Y
1   if n == 1
2       return

3   m = n/2
4   l = S_m - R_0 // #suffixes in the first m subarrays
5   MERGE-PARTITION(Y_{[0:l)}, X_{[0:l)}, m, R_{[0:m)},
                    L_{Y [0:l)}, L_{X [0:l)}, T) ||
    MERGE-PARTITION(Y_{[l:R_n)}, X_{[l:R_n)}, n - m, R_{[m:n)},
                    L_{Y [l:R_n)}, L_{X [l:R_n)})
6   MERGE(X_{[0:l)}, X_{[l:R_n)}, L_{X [0:l)}, L_{X [l:R_n)}, Y, L_Y, T)
```

The algorithm is presented as following, and its major steps are detailed in the following subsections. Then we analyze the asymptotic characteristics of the algorithm.

The MERGE operation. For efficient suffix comparisons, CAPS-SA utilizes the MERGE operation. A pair of suffixes is compared only when merging two sorted lists of suffixes, with the only exception being the case when the algorithm performs a binary search using a pivot suffix. When merging sorted suffixes, merging without any extra information about the suffixes in its input lists can be costly due to super-constant time string comparisons. To avoid comparing repeated prefixes of suffixes, the MERGE procedure in CAPS-SA utilizes the LCP-arrays of the input suffix lists, generated recursively in the MERGE-SORT procedure.

The MERGE$(X, Y, L_X, L_Y, Z, L_Z, T)$ procedure takes two sorted arrays X and Y of suffixes, their respective LCP-arrays L_X and L_Y, and populates the array Z as the merged output for X and Y. Also, the LCP-array of Z is produced in L_Z. The procedure works exactly like the classic merge routine, with the following modifications.

At a given moment, let X_i and Y_j be the two suffixes being compared, and Z_k be the output of the comparison. Without loss of generality, say that $X_i < Y_j$ is found, i.e. $Z_k = X_i$. Let m denote the LCP-length of the the last compared pair in each step of the merge. Then after the current step finishes comparing X_i, Y_j, we have that $m = \text{LCP}(X_i, Y_j)$. $X_i < Y_j$ implies that $T_{X_i+m} < T_{Y_j+m}$. The next suffixes to compare are X_{i+1} and Y_j. Let $l_x = L_{Xi+1} = \text{LCP}(X_{i+1}, X_i)$. $X_{i+1} > X_i$ implies that $T_{X_{i+1}+l_x} > T_{X_i+l_x}$. There are three possible outcomes when comparing l_x and m (illustrated in Figure 2):

1. $l_x > m$: It implies that $T_{X_{i+1}+m} = T_{X_i+m}$. Combining with $T_{X_i+m} < T_{Y_j+m}$, we get $T_{X_{i+1}+m} < T_{Y_j+m}$. It follows that

$$Z_{k+1} = X_{i+1}, L_{Zk+1} = \text{LCP}(Z_{k+1}, Z_k) = \text{LCP}(X_{i+1}, X_i) = l_x, m = \text{LCP}(X_{i+1}, Y_j) = m$$

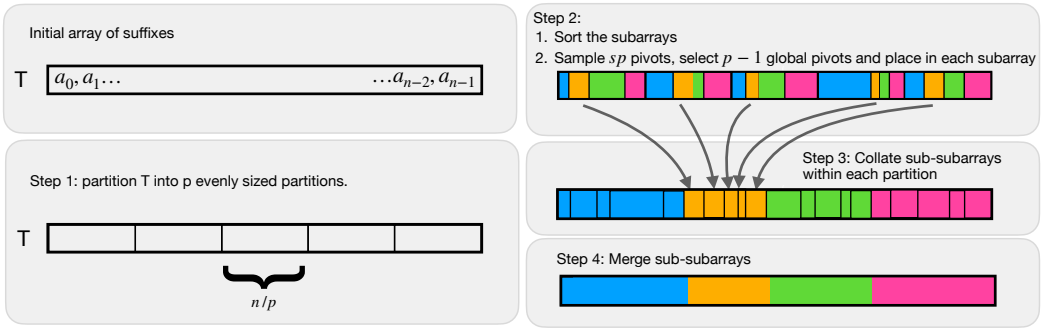

Figure 1 Overview of CAPS-SA. In the first step of the algorithm the input text T is partition evenly across n partitions. Then each partition is sorted, pivots are sampled using the sampling routine, and located within each partition to create sub-partitions. Subsequently each sub-partition is collated. Finally the MERGE routine is used to complete the suffix and the LCP-array construction.

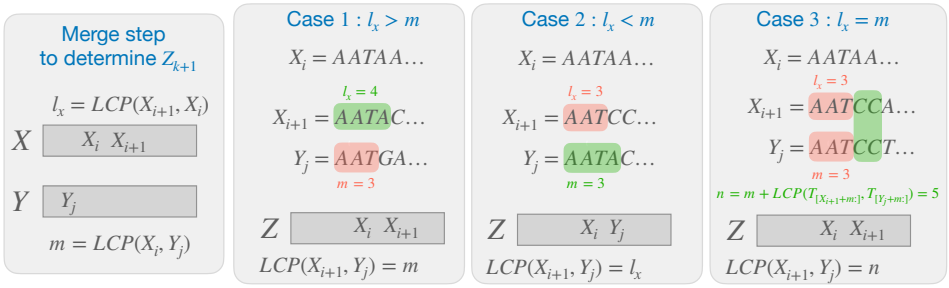

Figure 2 Figure illustrating the cases that can occur on the $(k+1)$'th step of the MERGE routine, which determines Z_{k+1}. Cases 1 and 2 require $\mathcal{O}(1)$ work and simply compare the LCP-lengths of the previous step and $LCP(X_{i+1}, X_i)$, which are already available. Step 3 requires work proportional to $\mathcal{O}(LCP(X_{i+1}, Y_j) - m)$, since X_{i+1}, Y_j already share a prefix of size m.

2. $l_x < m$: It implies that $T_{X_i+l_x} = T_{Y_j+l_x}$. Combining with $T_{X_{i+1}+l_x} > T_{X_i+l_x}$, we get $T_{X_{i+1}+l_x} > T_{Y_j+l_x}$. It follows that

$$Z_{k+1} = Y_j, L_{Z\,k+1} = LCP(Z_{k+1}, Z_k) = LCP(Y_j, X_i) = m, m = LCP(X_{i+1}, Y_j) = l_x$$

3. $l_x == m$: We compute $n = m + LCP(T_{[X_{i+1}+m:]}, T_{[Y_j+m:]})$, and set the following:

$$Z_{k+1} = \begin{cases} X_{i+1} & \text{if } T_{X_{i+1}+n} < T_{Y_j+n} \\ Y_j & \text{otherwise} \end{cases}$$

$$L_{Z\,k+1} = LCP(Z_{k+1}, Z_k) = \begin{cases} LCP(X_{i+1}, X_i) = l_x & \text{if } Z_{k+1} == X_{i+1} \\ LCP(Y_j, X_i) = m & \text{otherwise} \end{cases}$$

$$m = LCP(X_{i+1}, Y_j) = n$$

The MERGE procedure continues this way through X and Y. Finally, when either of X and Y has been depleted, the rest of the entries at the other one are copied to the end of Z and L_Z.

Local sorting. CAPS-SA starts out with some permutation of $[0, |T|)$, and sorts its p disjoint subarrays, each of size $|T|/p$, in parallel using MERGE-SORT. The MERGE-SORT(X, Y, L, L', T) procedure takes as input an array X of suffixes, and sorts it into Y. Besides, the LCP-array

of sorted suffixes is produced in L, using L' as working space. As typical Merge-Sort implementation requires linear extra space in each invocation, CaPS-SA uses the arrays X and Y in a back-and-forth manner to reuse the extra space in the invocations. For such, Y needs to be equal to X before an invocation. The merge step in the sort uses the Merge-procedure described earlier.

Pivot selection. CaPS-SA deviates from samplesort in its pivot selection strategy. In a typical samplesort, pivots are to be sampled from the initial array and then partitioning would be based on their intervals. Instead, in parallel, the Sample-Pivots(SA, T, p) procedure (see Suppl.) in CaPS-SA samples s suffixes from each of the p subarrays, where s is the *sampling factor*. Then these $s \times p$ sample suffixes are sorted using a sequential Merge-Sort. Subsequently, $p - 1$ evenly-spaced pivots are selected from the sorted output to form the pivot set V.

These pivots define the ranges of the samplesort partitions, and are used to split each of the subarrays in the next collation step. We show in Theorem 1 that with a sufficient sampling factor s, the size of each partition is within a constant factor of $|T|/p$ with high probability, which ensures a balanced load for processing each partition in the last step of the algorithm.

Collating partitions. Having finalized the pivot set V, the algorithm locates each pivot suffix $v \in V$ in each sorted subarray. Each subarray is searched for the $p-1$ pivots in parallel.

Consider a pivot $v \in V$ and some sorted subarray A. The position of v in A is the last index where v can be inserted without breaking the sorted order of A. This index is computed using a binary search for the suffix v in A. During a binary search suffixes are compared without any associated LCP array, contrary to the Merge procedure. As a practical speedup, we skip some repeated character comparisons between v and the suffixes in A using the *simple accelerant* idea [18].

After placing each pivot v into A, the index i of v in A implies that all the suffixes in $A_{[0:i)}$ are $\leq v$. Hence the sum C_v of these indices of v across all the p sorted subarrays provides the total suffix count in the SA that are not lexicographically larger than v – the index of v in the final SA is $C_v - 1$. Along with the sentinel pivot positions $C_0 = 0$ and $C_p = |T|$, these $p - 1$ pivots divide the final SA into p partitions. Consider two successive pivots v_{j-1} and v_j. In each sorted subarray A, all the suffixes k such that $T_{[v_{j-1}:]} < T_{[k:]} \leq T_{[v_j:]}$ will be present in the index-range $[C_{j-1}, C_j)$ of the final SA. That is, all the suffixes between the locations for v_{j-1} and v_j belong to the $(j-1)$'th partition.

Thus the pivot locations in a sorted subarray A break A into p sub-subarrays, where the j'th sub-subarray is known to be present in the j'th partition of the final SA. After the binary searches, CaPS-SA moves these sub-subarrays in parallel to collate all sub-subarrays for the same partition. The LCP-arrays of these sub-subarrays are also collated together. The Collate-Partitions(SA, SA', L, L', V, T, p) procedure (see Suppl.) describes it in more detail.

Merging partitions. Having grouped together the corresponding sub-subarrays for every partition, CaPS-SA merges together the sorted sub-subarrays in each partition, in parallel. A partition consists of p sorted collections of suffixes, with all of the collections stored contiguously. The Merge-Partition(X, Y, n, R, L_X, L_Y, T) procedure takes this collection X of n sorted sub-subarrays, and produces the merged output in the same contiguous region of memory Y recursively. L_X is the collection of the LCP-arrays of the sorted groups in X, and the merged LCP-array is produced in L_Y. The sorted groups in X (and L_X) are delineated by R.

The Merge-Partition procedure is same as the Merge-Sort procedure, except for that it is more general – the sorted units where Merge-Sort bottoms out are single suffixes, whereas Merge-Partition bottoms out earlier at sorted groups of suffixes. As noted earlier, Merge-Partition also uses the space in X and Y back-and-forth to reuse the extra spaces required.

3.2 Asymptotics

In this section, we analyze the computational complexity of the CaPS-SA(T, p) algorithm executed on a text T with length $n = |T|$, given a subproblem-count p.

3.2.1 Work Analysis

We start by analyzing the overall work of the algorithm and providing self-contained proofs on the total work due to symbol comparisons made by our algorithm.

Local sorting. This step executes the classic Merge-Sort on each subarray. For a subarray A with m suffixes, this amounts to a total work of $T(m) = 2T(m/2) + \mathcal{O}(m) + C(A)$, where $C(A)$ denotes the number of symbol comparisons made in the execution in the third case of the Merge procedure. We analyze the total amortized cost of these $C(A)$ values across all the recursion-trees of all the subarrays in Theorem 3. Omitting $C(A)$ from $T(m)$, each local sort has $n/p \log n/p$ work.

Pivot selection. With a sampling factor s, there are $s \times p$ pivots sampled in total across all the subarrays. CaPS-SA sorts these pivots with Merge-Sort and picks the $p - 1$ equidistant pivots from these as the global pivots. The Merge-Sort amounts to a total work of $\mathcal{O}(sp \log sp + \sum L_{p_i})$, where L_p is the output LCP-array of the sort. This holds from Theorem 3.

Collating partitions. The collation step first locates each of the $p - 1$ pivots in each of the sorted subarrays using binary search. The length of a pivot suffix is $\mathcal{O}(n)$, and the sorted subarrays are of size n/p. The work of each binary search is $\mathcal{O}(n + \log n/p)$ in practice ($n = |T|$) with the simple-accelerant [18] strategy. For adversarial inputs however, the work can still be $\mathcal{O}(n \log n/p)$ in the worst-case. Then the suffix indices are moved into their appropriate final partitions. This step simply reorders the elements across the sorted subarrays, and thus requires $\mathcal{O}(n)$ total work.

Merging partitions. The Merge-Partition procedure works similar to the Merge-Sort procedure, except for that the recursion bottoms out at a sorted group of suffixes, instead of at a single suffix. Unlike the Merge-Sort instances however, each of which operate on n/p-sized subarrays, the Merge-Partition instances may work on various sizes of partitions. Theorem 1 provides a bound on the partition sizes.

▶ **Theorem 1.** *With a sampling factor s, every partition has size at most cn/p for some constant c with high probability.*

Proof. The algorithm samples s pivots from each subarray, for a total of sp samples. It then picks $p - 1$ evenly spaced pivots (every s'th sample) from the sorted samples to use as global pivots.

Consider the final location of these sp samples in the final suffix array. Every s'th of them marks the boundary of a partition. Thus, a partition has size $d \geq cn/p$ only if fewer than s of the samples fall into these d suffixes. Otherwise, at least one sample would be picked as a final pivot and would thus break this partition.

Let SA be the final suffix array, and X_i be a random variable indicating whether SA_i is one of the sp samples. Then $\Pr[X_i = 1] = sp/n$. Thus the random variable denoting the number of samples picked from a region of size cn/p is $X = \sum_{i=1}^{cn/p} X_i$. By linearity of expectation, we get $E[X] = \sum_{i=1}^{cn/p} E[X_i] = cn/p \Pr[X_i = 1] = cn/p \cdot sp/n = cs$. Applying the Chernoff bound we have:

$$\begin{aligned}\Pr[X < s] \leq \Pr[X \leq s] &= \Pr\big[X \leq \frac{1}{c}E[X]\big] = \Pr\big[X \leq \big(1 - (1 - \frac{1}{c})\big)E[X]\big] \\ &\leq \exp\big(-1/2(1 - 1/c)^2 E[X]\big) = \exp(-1/2(1 - 1/c)^2 cs) \qquad (1)\end{aligned}$$

With $s = 32 \ln n$ and letting $c' = c(1 - 1/c)^2$,

$$\Pr[X < s] \leq \exp\big(-1/2(1 - 1/c)^2 c \cdot 32 \ln n\big) = \exp\big(\ln(n^{-16c(1-1/c)^2})\big) = 1/n^{16c'}$$

Since the event of a partition having size $\geq cn/p$ implies the event $X < s$, we get:
$\Pr[\text{a given partition has size} \geq cn/p] \leq \Pr[X < s] \leq 1/n^{16c'}$.
$\Rightarrow \Pr[\text{at least one partition has size } \geq cn/p] \leq \sum_{i=1}^{p} 1/n^{16c'} = p/n^{16c'}$ (by union-bound).
$\Rightarrow \Pr[\text{no partition has size} \geq cn/p] \geq 1 - p/n^{16c'}$.
Since p is at most $\mathcal{O}(n)$, no partition has size $\geq cn/p$ *whp*. ◀

Thus each partition has size at most cn/p for some constant c *whp*. Merging a partition A with m sorted subgroups has work $T(m) = 2T(m/2) + \mathcal{O}(m) + C(A)$, where $C(A)$ is the number of symbol comparisons made in MERGE. Omitting $C(A)$ from $T(m)$, this solves to $\mathcal{O}(cn/p \log p) = \mathcal{O}(n/p \log p)$ *whp*.

Total symbol comparisons. A subproblem in the algorithm is some subarray of the SA that can be processed independently of the other subarrays in a given step. Let X be some subproblem in either of the two steps: local-sorting and partition-merging. For the local-sorting case, sorting X with MERGE-SORT consists of $\log n/p$ recursion-levels. For the partition-merging case, the MERGE-PARTITION procedure for X executes in $\log p$ recursion-levels. We label the bottom-most level as level 0, and count the levels upwards in the recursion-tree.

Let $x \in X$ be a suffix. At any given level i, x is present in exactly one MERGE-SORT (or MERGE-PARTITION) instance executing on X. Let x'_i be the suffix that immediately precedes x in the output of that MERGE-SORT (or MERGE-PARTITION) instance, and let $L_i(x) = \text{LCP}(x, x'_i)$. If x is the first suffix in the output, then x'_i is the empty suffix. We prove the following.

▶ **Theorem 2.** *In* MERGE-SORT *and* MERGE-PARTITION, $L_i(x) \leq L_{i+1}(x)$ *for a suffix* x *at each recursion level* $i \in [0, d-1)$*, where* d *is the depth of the recursion-tree.*

Proof. Let x be present in the MERGE-SORT (or MERGE-PARTITION) instance M at level $i+1$, and say M spawns the two instances M_l and M_r. M_l and M_r are at level i, and x is present in exactly one of them. Let it be M_l.

Now, x'_{i+1} is either x'_i i.e. the same suffix preceding x in M_l, or some other suffix y from M_r. If $x'_{i+1} = x_i$, then $L_{i+1}(x) = L_i(x)$, and the claim holds.

In the other case, the output array of M has the following form: $[\ldots, x'_i, \ldots, y, x, \ldots]$. Suppose that the claim is false, i.e. $L_i(x) > L_{i+1}(x)$. Which is, $\mathrm{LCP}(x, x'_i) > \mathrm{LCP}(x, y)$. $\mathrm{LCP}(x, y) < \mathrm{LCP}(x, x'_i)$ implies that x'_i and y share the same prefix of length $l = \mathrm{LCP}(x, y)$, and mismatch first at index l. Let $c_x, c_{x'_i}$, and c_y be the l'th symbol in x, x'_i, and y resp. As $y > x'_i$ in the output, $c_y > c_{x'_i}$. Besides, since x and y first mismatch at the l'th symbol and $x > y$, $c_x > c_y$. Thus $c_x > c_{x'_i}$. $\mathrm{LCP}(x, x'_i) > l$ implies that the l'th symbols in x and x'_i are the same, i.e. $c_x = c_{x'_i}$. Thus we get $c_x > c_{x'_i}$ and $c_x = c_{x'_i}$, resulting in a contradiction. Hence, $\mathrm{LCP}(x, y) \leq \mathrm{LCP}(x, x'_i)$. ◀

Theorem 3 provides a bound on the number of total comparisons made across all the Merge-Sort and Merge-Partition instances in the algorithm execution.

▶ **Theorem 3.** *The total number of symbol comparisons made across all the* Merge-Sort *and* Merge-Partition *instances in* CaPS-SA *for an n-length text is $\mathcal{O}(n \log n + \sum_{i=1}^{n-1} L_i)$ w.h.p, where L is the output* LCP*-array.*

Proof. Symbol comparisons occur only as part of the Merge procedure in both Merge-Sort and Merge-Partition. Given two sorted lists of suffixes X and Y along with their LCP-arrays, the $\mathrm{Merge}(X, Y, L_X, L_Y, Z, L_Z, T)$ procedure iterates through the X_i's and Y_j's and fills in the Z_k's in the sorted order, along with their LCP-values in L_{Zk}. Without loss of generality, suppose that $X_i < Y_j$ is found at some iteration. Then the next iteration compares X_{i+1} and Y. Let $l_x = \mathrm{LCP}(X_{i+1}, X_i)$ and $m = \mathrm{LCP}(X_i, Y_j)$, LCP of the last compared pair. Symbols from the suffixes X_{i+1} and Y_j will only be compared iff $l_x = m$ holds. In this case, we compute $n = \mathrm{LCP}(X_{i+1}, Y_j)$ with exactly $n - m + 1$ symbol comparisons. The $+1$ term is due to the first mismatching symbol pair. m is set to n for the next iteration. We argue that before the new $m = n$ value is assigned as the LCP-value in the output LCP-array L_Z in some future iteration, it remains unchanged.

In the next iteration, if case (1), i.e. $l_x > m$ holds, then m remains unchanged. If case (2), i.e. $l_x < m$ holds, then m is assigned at output L_{Zk+1}. In the event of case (3), either l_x or m is assigned to L_{Zk+1}, and these are equal in this case. If X has been depleted during the merge while Y still has remaining elements, the current m is assigned as the LCP-value for the first of the remaining elements from Y.

Thus, whenever symbol comparisons are done in case (3) of merge, it results in a new value $m' \geq m$ for the variable m. $m = m'$ persists until m' has been assigned as the LCP-value for some merged output. Thus the number of matching symbol comparisons made in case (3) accumulates in the LCP-values at the output.

All the LCP-values start out with 0 at Merge-Sort. Theorem 2 states that the LCP-value associated to a given suffix can never decrease while winding up the recursion-trees of Merge-Sort and Merge-Partition. Thus the sum $\sum_{i=1}^{n-1} L_i$ of the final LCP-values in the SA is the total number of matching symbol comparisons made across all the Merge-Sort and Merge-Partition executions.

The extra mismatching comparison in case (3) of Merge costs $\mathcal{O}(1)$. In the worst case, this case occurs in each iteration of Merge. Omitting the matching symbol comparisons, a Merge-Sort or a Merge-Partition instance working on m elements incurs $T(m) = 2T(m/2) + \mathcal{O}(m)$ mismatches in the worst case. This solves to $p \times \mathcal{O}(n/p \log n/p)$ and $p \times \mathcal{O}(n/p \log p)$ *whp* for the p Merge-Sorts and Merge-Partitions, resp. Thus $\mathcal{O}(n \log n)$ mismatching symbol comparisons are made *whp*. ◀

Total work. Locally sorting the p subarrays cost $p \times \mathcal{O}(n/p \log n/p) = \mathcal{O}(n \log n/p) =$ work without the symbol comparisons. Omitting the symbol comparisons in sorting the sampled pivots, the pivot selection step has $\mathcal{O}(sp \log sp)$ work. In the collation step, there are $p(p-1)$

binary searches, costing $\mathcal{O}(p^2 n \log n/p)$ work in the worst-case, and $\mathcal{O}(p^2(n + \log n/p))$ in practice. Merging the p partitions separately cost $p \times \mathcal{O}(n/p \log p) = \mathcal{O}(n \log p)$ whp without the symbol comparisons.

The total number of symbol comparisons in the local-sort and the partitions-merge steps is $\mathcal{O}(n \log n + \sum_{i=1}^{n-1} L_i)$ *whp* as per Theorem 3, where L is the output LCP-array. In sorting the sampled suffixes, the number of symbol comparisons done is also bounded by this [2].

We note that the algorithm requires on the order of $4w|T|$ bytes of working space, where $w \in \{4, 8\}$ is the numerical size used to store SA and LCP values.

3.2.2 Parallelization

Our implementation fully parallelizes the work across the different partitions. Within a partition, we perform recursive calls to MERGE-SORT in parallel, but perform the MERGE procedure serially. We show the following theorem about the depth of our algorithm:

▶ **Theorem 4.** *The overall depth of the algorithm is* $\mathcal{O}((n/p) \log n)$ whp.

Proof. The dominant factor in the merge algorithm is the depth of the MERGE routine, which simply performs a linear number of comparisons in the input size. The depth of a comparison is $\mathcal{O}(1)$ in cases 1 and 2 of Figure 2, and requires a string comparison in the final case.

The string comparison can be parallelized work-efficiently (i.e., in the same work as a serial character-by-character comparison) by using a simple prefix-doubling strategy. In more detail, the comparison algorithm works in rounds comparing 2^i characters in the i'th round until a mismatch occurs. Clearly for strings of length $\mathcal{O}(n)$ only $\mathcal{O}(\log n)$ rounds are required, and thus the overall work is asymptotically the same as the serial algorithm, and the depth is $\mathcal{O}(\log n)$. Thus, for merging two sorted arrays in the algorithm, each of size k, we require a depth of $\mathcal{O}(k \log n)$.

Putting these facts together, for a single call to the MERGE-SORT routine, we have a recurrence of the form $D(k) = D(k/2) + \mathcal{O}(k \log n)$, which is root dominated and solves to $\mathcal{O}(k \log n)$. Since our algorithm is parallelized across different partitions, and by Theorem 1 each partition has size at most $\mathcal{O}(n/p)$, the overall depth of the algorithm is $\mathcal{O}((n/p) \log n)$ *whp*. ◀

We note that the depth is not poly-logarithmic, as in the classic parallel MERGE-SORT. However, the amount of parallelism generated by our algorithm is more than enough to keep the processors all busy in practice. Indeed, we note that many samplesort implementations use a similar strategy in practice and use a serial sort within each partition, and thus also do not have poly-logarithmic depth in practice. In our implementation, we exploit parallelism using the parallel primitives and the work-stealing scheduler from `ParlayLib` [9].

3.2.3 Optimizations

We applied a number of optimizations into the implementation of the algorithm that provide practical speedups. We make use of vectorization support using `AVX` instructions from modern processors to speed up the computation of the $\text{LCP}(X, Y)$ routine used in the MERGE-procedure and in the binary searches in locating pivots.

[2] $\mathcal{O}(sp \log sp + \sum_{i=1}^{sp} L_{p\,i})$ comparisons are performed, where L_p is the LCP-array of the sorted samples.

In the proposed Merge-Sort and the Merge-Partition procedures in the algorithm, we have nested parallelism for their recursive invocations. This is applied in the implementation up-to some fixed granularity, due to the associated overhead of scheduling small tasks.

In the binary searches for the sampled pivots in each sorted subarray, instead of searching for the appropriate position of an entire pivot suffix, we look for a fixed-length prefix of the pivot. This helps reduce the total work associated to locating the pivots, with an associated trade-off with the final partition sizes. With sufficiently large prefix lengths, the partition sizes do not get significantly affected in our observation.

4 Results

We performed a number of experiments to characterize the performance of the CaPS-SA algorithm and its implementation. We evaluated its performance compared to the available implementations of two leading methods for SA construction: Parallel-divsufsort [32] and Parallel-DC3 [3]. We assessed its ability to construct SA and LCP-arrays on a number of genomic datasets.

Next, we evaluated the parallel scaling of the algorithm. Then we explore the idea of *Bounded-context suffix arrays*, and the performance of CaPS-SA for various prefix-context lengths.

A varied collection of datasets has been used in the experiments. Table 1 delineates the pertinent characteristics of the datasets. We follow [46] by removing `N`-repeats, which occur when the sequence underlying a region of the assembly cannot be resolved. We verified the correctness of the implementation by cross-checking its output against from that of Parallel-divsufsort.

Computation system. The experiments have been performed on a server having two Intel Xeon E5-2699 2.20 GHz CPUs with 44 cores in total and 512 GB of 2.40 GHz DDR4 RAM. The system is run with Ubuntu 20.04.4 (GNU/Linux 5.4.0-132-generic x86_64). The SA and LCP-array construction times and the maximum memory usages of the tools were measured with the GNU `time` command. For the large datasets Axolotl genome dataset, we used machines with two AMD EPYC 7313 CPUs with 32 cores in total and 2 TB of DDR4 RAM, running on Red Hat Enterprise Linux 8.7 (GNU/Linux 4.18.0-425.19.2.el8_7.x86_64).

4.1 Dataset characteristics

Table 1 provides some pertinent characteristics of the datasets used. The GRCh38 dataset is the Human Build 38 patch release 13 version of the human genome reference from the Genome Reference Consortium[3], which is a chromosome-level assembly of the full genome. The T2T dataset is the latest T2T CHM13v2.0 Telomere-to-Telomere assembly of the CHM13 cell line with chromosome Y from NA24385, from the T2T consortium, which is a complete genome-level assembly of the genome [40]. Together, these two human datasets represent what we imagine may be a *typical* use-case for genome construction in the context of a tool like STAR [12]. Though largely similar, the CHM13 assembly has resolved telomeric and centromeric regions, and more complete coverage, specifically in highly-repetitive regions. Thus, we expect it represents a more challenging problem instance for suffix array construction.

[3] `https://www.ncbi.nlm.nih.gov/grc`

The CdBG (Compacted de Bruijn Graph) dataset is the collection of the maximal unitigs extracted from the de Bruijn graph (with k-mer size 27) of the human sequencing read set NIST HG004 (SRA3440461 – 95) [52] by Cuttlefish 2 [28]. This dataset represents a potential use-case where one may wish to build an index for the sequence stored in the CdBG data structure. The ability to index the CdBG has proven useful in many contexts [10], and the SA can provide one possible index for providing efficient lookup over the sequence contained in the CdBG.

The great white shark dataset is the genome reference of *Carcharodon carcharias* [36] and the axolotl dataset is the genome reference sequence of *Ambystoma mexicanum* [47]. These represent large problem instances, where one may wish to build the SA on large reference genomes.

Table 1 Dataset statistics: number of bases, mean LCP, and standard deviation (rounded to nearest whole number) of the final LCP-array.

Dataset	Size	Mean LCP	Std. Dev. of LCP
Human (GRCh38)	2,945,849,068	3,807	72,678
Human (T2T)	3,117,292,071	2,518	61,987
CdBG (Human reads)	3,993,272,308	18	6
Great white shark	4,267,160,925	109	1,192
Axolotl	28,203,219,824	49	160

4.2 SA and LCP-array construction

We evaluate the performance of CaPS-SA in constructing the SA and the LCP-array of a number of genomic datasets, compared to the SA construction performance of: 1. Parallel-divsufsort [32] and 2. Parallel-DC3 [3]. Table 2 contains the results of the benchmarking. As the state-of-the-art sequential benchmark, we note the performance of the `divsufsort` implementation from PBBS [3].

Table 2 Time- and memory-performance results for constructing SAs (and LCP-array in case of CaPS-SA) with 32 threads. `divsufsort` is shown as a serial benchmark. Time is reported in seconds, and the memory usages are reported in GBs in parentheses. Best performances among the parallel algorithms in each instance are highlighted. Parallel-DC3 and `divsufsort` could not be run on Axolotl because we could not modify the PBBS code-base to accommodate the large numerical size.

Dataset	CaPS-SA	Parallel-divsufsort	Parallel-DC3	`divsufsort`
Human (GRCh38)	**131** (47)	173 (**32**)	250 (110)	273 (17)
Human (T2T)	**141** (50)	199 (**34**)	261 (115)	319 (18)
CdBG (Human reads)	**138** (64)	291 (**44**)	323 (126)	409 (23)
Great white shark	**173** (71)	278 (**48**)	381 (135)	438 (24)
Axolotl	**900** (910)	1068 (**326**)	-	-

We note that CaPS-SA executes significantly faster than the other parallel algorithms in all the instances, whereas Parallel-divsufsort uses the least amount of memory. Interestingly for the smaller datasets CaPS-SA does not require much more memory than Parallel-divsufsort despite constructing both the SA and LCP. Memory usage could be improved by bit-packing the indexes, or through the extension to an external memory algorithm.

4.3 Parallel Scaling

In order to assess how sensitive runtime is to parallelism we evaluated CaPS-SA against Parallel-DC3 and Parallel-divsufsort as the number of threads increased. We report the results in Figure 3, which illustrated that CaPS-SA exploits parallelism better – becoming the fastest method as the thread count becomes high despite doing more work asymptotically.

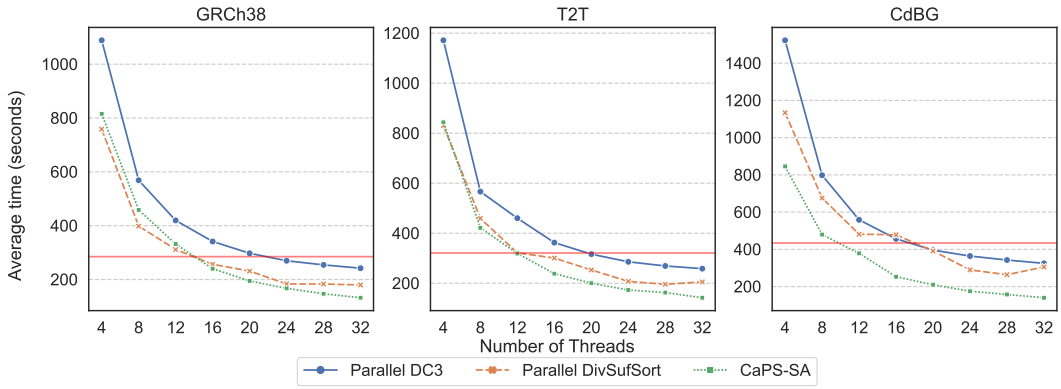

Figure 3 Runtimes of CaPS-SA, Parallel-divsufsort, and Parallel-DC3 as thread count increases for GRCh38, T2T, and CdBG. `divsufsort` runtime is in red.

On the GRCh38 and T2T datasets, CaPS-SA and Parallel-divsufsort become faster than `divsufsort` at around the same number of threads, after which CaPS-SA becomes faster. On CdBG, CaPS-SA sees gains from parallelism over the best serial program with approximately half the number of threads. This may be because CdBG has fewer long common-prefixes, and so CaPS-SA achieves closer to its best-case work.

4.4 Cache Performance

The samplesort-based design of CaPS-SA optimizes cache-performance. In order to evaluate the empirical cache behavior of CaPS-SA as compared to other algorithms for SA construction, we profiled the programs on the GRCh38 and the T2T reference genomes. Because cache-behavior can degenerate as parallelism increases, we evaluate it across 1, 16, and 32 threads. The results in Table 3 show that CaPS-SA outperforms other parallel SA indexing programs by an order of magnitude. All measurements were taken with the Linux `perf` command.

Table 3 Cache-miss rates (in %) for compared methods on GRCh38 and T2T datasets with respect to number of threads. Reported numbers are averaged over 5 runs to obviate operating system jitter. Lower is better, and the best result is highlighted.

	Human (GRCh38)			Human (T2T)		
Method	**1 thr**	**16 thr**	**32 thr**	**1 thr**	**16 thr**	**32 thr**
CaPS-SA	4.83	**4.92**	**4.99**	7.34	**7.61**	**7.81**
Parallel-divsufsort	21.98	25.03	26.31	37.38	37.98	38.68
Parallel-DC3	34.75	35.91	37.43	35.04	36.23	37.66
`divsufsort`	**0.93**	–	–	**0.99**	–	–

`divsufsort` is somewhat more cache-efficient than CaPS-SA, but interestingly its parallelization in Parallel-divsufsort has significantly worse caching performance than both `divsufsort` and CaPS-SA. It is possible that incorporating some of the implementation details of `divsufsort` could provide further improvements to the cache-performance of CaPS-SA.

4.5 Bounded-context SA Construction

By virtue of organizing all suffixes of the underlying text T, the suffix array provides the powerful ability to efficiently search for query patterns of *any* length in the text. While this capability arises naturally from the definition of the SA, such flexibility is rarely needed in the SA's most common applications in genomics. Specifically, when used to efficiently find *seed* sequences from a genomic read, the maximum length of the query is often very short. Many modern aligners use seed lengths in the range of 15–31, and even with the maximum mappable prefix concept used by STAR [12], the query length is bounded above by the error-free prefix length of the remainder of the read (rarely more than ~ 100 nucleotides).

As such, indices that can provide efficient lookup and locate queries for patterns less than some maximum length, say k, are often very useful in this context. For example, the k-BWT data structure [45, 41, 11] builds a transform of the text that organizes character occurrences by their *bounded context* (in this case, their right context of length k). This allows the index to be built efficiently, since rotations of the text need not have their relative orders resolved beyond their initial length k contexts, while simultaneously allowing efficient and correct query for any pattern length $\leq \mathsf{k}$.

Here, we experiment with an analogous version of the SA– the *bounded-context SA*. Specifically, the bounded-context SA of order k resolves the lexicographic order of all suffixes of the text up to (and including) their prefixes of length k. If a pair of suffixes share a prefix of length $\geq \mathsf{k}$, then they may appear in an arbitrary relative order within the bounded-context SA of order k. Without any meaningful modifications to the query algorithms, this variant of the SA allows locating all occurrences of queries of any length $\leq \mathsf{k}$ in the text. Such a variant of the SA is very straightforward to construct using CaPS-SA, as we simply declare equal any suffixes that are equal up to (or beyond) their length k prefixes. At the same time, this variant can be more efficient to construct using our algorithm, as the context length k places a strict upper bound on the number of comparisons we must perform when attempting to determine the relative order of a pair of suffixes. Specifically, it follows directly from Theorem 3 that CaPS-SA performs at most $\mathcal{O}(\mathsf{n} \log \mathsf{n} + \mathsf{nk})$ character comparisons, in the worst case, when constructing the context-bounded SA of order k. In Table 4, we report the time required to construct the context-bounded SA of orders 64 and 256 of the GRCh38 and CHM13 human genome assemblies compared to the time required to construct the standard (full-context) SA. As expected, the bounded-context SA can be constructed substantially faster than the full-context SA.

Table 4 Timings (in seconds) in constructing the bounded-context SA with various orders.

Dataset	**full-context**	64	256
Human (GRCh38)	132	104	100
Human (T2T)	144	105	103

5 Conclusion

In this manuscript, we introduced a new method, CaPS-SA, for parallel SA and LCP-array construction. CaPS-SA displays very good cache performance (i.e. very low cache miss rate), and scales well to many threads. As a result, CaPS-SA is able to outperform existing state-of-the-art parallel SA construction algorithms like Parallel-divsufsort and Parallel-DC3 on genomic datasets. At the same time, CaPS-SA is substantially simpler than existing state-of-the-art algorithms. This simplicity eases implementation, and leads to many opportunities for further future improvements. Likewise, CaPS-SA provides the LCP-array directly as a byproduct of SA construction, and does not require a separate algorithm to produce this useful auxiliary data structure. We hope that will prove to be useful in utilities where parallel SA construction is a core subproblem, and also hope that the relatively straightforward algorithm will benefit from further optimizations, enhancements, and alternative implementations within the community.

As CaPS-SA scales well with the level of available parallelism, and performs well for large references, we expect that it will provide a useful option for tools that seek to build the SA in parallel environments. In addition to the *time* taken to construct the SA or the LCP array, another consideration is the memory (specifically the RAM) required for construction. One approach to improve the *memory-scalability* of SA construction algorithms is to develop external-memory construction algorithms. For example, `pSAscan` [24] is a state-of-the-art external-memory algorithm for SA construction. Such approaches make use of external-memory (i.e. disk) and algorithms that access and construct the SA in a structured way are likely amenable to external-memory variants.

We note that, though we have not explored it in this manuscript, CaPS-SA is highly-amenable to external memory implementation. This is because the initial partitioning generates many small subproblems that can be solved independently – i.e. some subproblem can be paged into RAM while others remain on disk. Pivot sampling from the subproblems can be done through a similar paging process. Likewise, after pivot selection, many approximately equal-sized partitions will be created, and these sub-problems, which target specific output intervals of the final suffix array, can be solved independently and in parallel with the relevant data for only a working subset of partitions paged into RAM with the remaining partitions residing on disk. Further, given a sufficiently fine-grained partitioning, the algorithm can likely provide tight controls on the required working memory. As more RAM use is allowed, a larger number of partitions will be allowed in RAM at once, and our algorithm will be able to better make use of available parallelism. On the other hand, as the maximum allowed RAM usage is restricted, fewer partitions will be present in memory at once, potentially limiting parallelism, but adhering to the requested RAM constraints. In practice, we believe that, so long as a sufficiently fine-grained partitioning is used, external-memory variants of our algorithm will still be able to efficiently make use of many threads while still substantially reducing the required working memory. We leave the efficient implementation of an external-memory variant of CaPS-SA to future work.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

2 Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. Technology dictates algorithms: recent developments in read alignment. *Genome Biology*, 22(1):249, August 2021. doi:10.1186/s13059-021-02443-7.

3 Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The problem-based benchmark suite (PBBS), v2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 445–447, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508422.

4 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.9.

5 Timo Bingmann. Scalable string and suffix sorting: Algorithms, techniques, and tools. *arXiv preprint arXiv:1808.00963*, 2018.

6 Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders. *Scalable Text Index Construction*, pages 252–284. Springer Nature Switzerland, Cham, 2022. doi:10.1007/978-3-031-21534-6_14.

7 Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77:235–286, 2017.

8 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *Algorithms–ESA 2013: 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings 21*, pages 169–180. Springer, 2013.

9 Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib-a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

10 Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent a set of k-long DNA sequences. *ACM Comput. Surv.*, 54(1), March 2021. doi:10.1145/3445967.

11 J Shane Culpepper, Matthias Petri, and Simon J Puglisi. Revisiting bounded context block-sorting transformations. *Software: Practice and Experience*, 42(8):1037–1054, 2012.

12 Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

13 M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

14 Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In *Prague Stringology Conference 2017*, page 62, 2017.

15 Johannes Fischer and Florian Kurpicz. *Lightweight Distributed Suffix Array Construction*, pages 27–38. Society for Industrial and Applied Mathematics, 2019. doi:10.1137/1.9781611975499.3.

16 Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807609.

17 W Donald Frazer and Archie C McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

19 Scott Hazelhurst and Zsuzsanna Lipták. KABOOM! a new suffix array based algorithm for clustering expression data. *Bioinformatics*, 27(24):3348–3355, December 2011.

20 Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, February 2011.

21 Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No. PR00268)*, pages 81–88. IEEE, 1999.

22 Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11:137–149, 2017.

23 Juha Kärkkäinen and Dominik Kempa. Engineering external memory LCP array construction: Parallel, in-place and large alphabet. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

24 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching*, pages 329–342, Cham, 2015. Springer International Publishing.

25 Juha Kärkkäinen, Dominik Kempa, Simon J Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017.

26 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30*, pages 943–955. Springer, 2003.

27 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

28 Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with Cuttlefish 2. *Genome Biology*, 23(1):190, September 2022. doi:10.1186/s13059-022-02743-6.

29 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings 14*, pages 186–199. Springer, 2003.

30 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings*, pages 200–210. Springer, 2003.

31 Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

32 Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017.

33 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *String Processing and Information Retrieval: 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, pages 268–284. Springer, 2018.

34 Gang Liao, Longfei Ma, Guangming Zang, and Lin Tang. Parallel DC3 algorithm for suffix array construction on many-core accelerators. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1155–1158, 2015. doi:10.1109/CCGrid.2015.56.

35 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

36 Nicholas J. Marra, Michael J. Stanhope, Nathaniel K. Jue, Minghui Wang, Qi Sun, Paulina Pavinski Bitar, Vincent P. Richards, Aleksey Komissarov, Mike Rayko, Sergey Kliver, Bryce J. Stanhope, Chuck Winkler, Stephen J. O'Brien, Agostinho Antunes, Salvador Jorgensen, and Mahmood S. Shivji. White shark genome reveals ancient elasmobranch adaptations associated with wound healing and the maintenance of genome stability. *Proceedings of the National Academy of Sciences*, 116(10):4446–4455, 2019. doi:10.1073/pnas.1819778116.

37 Yuta Mori. divsufsort. https://github.com/y-256/libdivsufsort, 2015. Accessed on 1 May 2023.

38 Waihong Ng and Katsuhiko Kakehi. Merging string sequences by longest common prefixes. *IPSJ Digital Courier*, 4:69–78, 2008.

39 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE transactions on computers*, 60(10):1471–1484, 2010.

40 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

41 Matthias Petri, Gonzalo Navarro, J Shane Culpepper, and Simon J Puglisi. Backwards search in context bound text transformations. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 82–91. IEEE, 2011.

42 Anton Pirogov, Peter Pfaffelhuber, Angelika Börsch-Haubold, and Bernhard Haubold. High-complexity regions in mammalian genomes are enriched for developmental genes. *Bioinformatics*, 35(11):1813–1819, 2019.

43 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4–es, 2007.

44 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms–ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings 12*, pages 784–796. Springer, 2004.

45 M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings DCC '97. Data Compression Conference*, pages 469–, 1997. doi:10.1109/DCC.1997.582137.

46 Anish Man Singh Shrestha, Martin C Frith, and Paul Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in bioinformatics*, 15(2):138–154, 2014.

47 Jeramiah J Smith, Nataliya Timoshevskaya, Vladimir A Timoshevskiy, Melissa C Keinath, Drew Hardy, and S Randal Voss. A chromosome-scale assembly of the axolotl genome. *Genome Res.*, 29(2):317–324, February 2019.

48 Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, March 2013.

49 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

50 Yuzhen Ye, Jeong-Hyeon Choi, and Haixu Tang. RAPSearch: a fast protein similarity search tool for short reads. *BMC Bioinformatics*, 12(1):159, May 2011.

51 Kaiyuan Zhu, Alejandro A Schäffer, Welles Robinson, Junyan Xu, Eytan Ruppin, A Funda Ergun, Yuzhen Ye, and S Cenk Sahinalp. Strain level microbial detection and quantification with applications to single cell metagenomics. *Nature Communications*, 13(1):6430, 2022.

52 Justin M. Zook, David Catoe, Jennifer McDaniel, Lindsay Vang, Noah Spies, Arend Sidow, Ziming Weng, Yuling Liu, Christopher E. Mason, Noah Alexander, Elizabeth Henaff, Alexa B.R. McIntyre, Dhruva Chandramohan, Feng Chen, Erich Jaeger, Ali Moshrefi, Khoa Pham, William Stedman, Tiffany Liang, Michael Saghbini, Zeljko Dzakula, Alex Hastie, Han Cao, Gintaras Deikus, Eric Schadt, Robert Sebra, Ali Bashir, Rebecca M. Truty, Christopher C. Chang, Natali Gulbahce, Keyan Zhao, Srinka Ghosh, Fiona Hyland, Yutao Fu, Mark Chaisson, Chunlin Xiao, Jonathan Trow, Stephen T. Sherry, Alexander W. Zaranek, Madeleine Ball, Jason Bobe, Preston Estep, George M. Church, Patrick Marks, Sofia Kyriazopoulou-Panagiotopoulou, Grace X.Y. Zheng, Michael Schnall-Levin, Heather S. Ordonez, Patrice A. Mudivarti, Kristina Giorda, Ying Sheng, Karoline Bjarnesdatter Rypdal, and Marc Salit. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(1):160025, June 2016. doi:10.1038/sdata.2016.25.

A Methods

Pseudo-codes for the procedures absent in the main text, Sample-Pivots, Collate-Partitions, and Compute-Boundary-LCPs are provided in the following.

Sample-Pivots(SA, T, p).

```
1  s = 32 ln |T| // sampling factor
2  V, V', w1, w2 = arrays of size s × p

   // sample pivots from each sorted subarray
3  for i ∈ [0, p) in parallel
4      b = im, e = (i + 1)m // range of the i'th subarray
5      sample s pivots from SA[b:e) into V[b:e) w/o replacement

6  V' = V
7  Merge-Sort(V, V', w1, w2, T)
8  sample (p − 1) pivots from V' into V
9  return V
```

COLLATE-PARTITIONS(SA, SA', L, L', V, T, p).

```
1  P = 2-D array of size p × (p + 1) // P_{i,j}: location of j'th pivot in i'th subarray

2  m = |T|/p // size of the subarrays
3  for i ∈ [0, p) in parallel // for each sorted subarray
4      b = im, e = (i + 1)m // range of the i'th subarr
5      P_{i,0} = 0, P_{i,p} = m // two sentinel pivots
6      for j ∈ [0, p − 1) // for each pivot
7          P_{i,j+1} = BINARY-SEARCH(SA_{[b:e)}, V_j)

8  S = array of size p
9  for j ∈ [0, p) in parallel
10     S_j = Σ_{i=0}^{p−1} (P_{i,j+1} − P_{i,j}) // sum size of the j'th sub-subarrays
11 S = PREFIX-SUM-SCAN(S) // S_j: index of the j'th partition in the final SA

12 R = 2-D array of size p × (p + 1) // R_{j,i}: index of the i'th sub-subarray in partition j
13 for j ∈ [0, p) in parallel // for each partition
14     let Y : SA'_{[S_j:S_{j+1})} // location for partition j
15     R_{j,0} = 0
16     for i ∈ [0, p) // for each sorted subarray
17         let A : SA_{[im,(i+1)m)} // i'th subarray
18         let X : A_{[P_{i,j}:P_{i,j+1})} // j'th sub-subarray in A
19         COPY(X, Y_{[R_{j,0}:)})
20         R_{j,i+1} = R_{j,i} + |X|
21 return (S, R)
```

COMPUTE-BOUNDARY-LCPS(SA, p, S, L, T).

```
1  for j ∈ [1, p) in parallel
2      s = SA_{S_j}, t = SA_{S_j−1}
3      L_{S_j} = LCP(T_{[s:]}, T_{[t:]})
```