Future Generation Computer Systems 149 (2023) 650–663

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Tunnel: Parallel-inducing sort for large string analytics

Zhihui Du [a,*], Sen Zhang [b], David A. Bader [a]

[a] New Jersey Institute of Technology, Newark, 07102, NJ, USA
[b] State University of New York, College at Oneonta, Oneonta, 13820, NY, USA

ARTICLE INFO

Article history:
Received 13 February 2023
Received in revised form 1 August 2023
Accepted 5 August 2023
Available online 9 August 2023

Keywords:
Suffix array
String algorithm
Parallel sorting
String analysis

ABSTRACT

The suffix array is a crucial data structure for efficient string analysis. Over the course of twenty-six years, sequential suffix array construction algorithms have achieved $\mathcal{O}(n)$ time complexity and in-place sorting. In this paper, we present the *Tunnel* algorithm, the first large-scale parallel suffix array construction algorithm with a time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$ based on the parallel random access machine (PRAM) model. The *Tunnel* algorithm is built on three key ideas: dividing the problem of size $\mathcal{O}(n)$ into p sub-problems of reduced size $\mathcal{O}\left(\frac{n}{p}\right)$ by replacing long suffixes with shorter prefixes of size at most a constant D; introducing a *Tunnel* mechanism to efficiently induce the order of a set of suffixes with long common prefixes; developing a strategy to transform a partially ordered suffix set into a total order relation by iteratively applying the *Tunnel* inducing method. We provide a detailed description of the algorithm, along with a thorough analysis of its time and space complexity, to demonstrate its correctness and efficiency. The proposed *Tunnel* algorithm exhibits scalable performance, making it suitable for large string analytics on large-scale parallel systems.

© 2023 Elsevier B.V. All rights reserved.

1. Introduction

Suffix arrays were initially introduced by Manber and Myers [1] as a space-efficient alternative to suffix trees [2]. They have since found applications in various domains such as string processing, data compression, text indexing, information retrieval, and computational biology. As the volume of string data continues to grow, the development of high-performance suffix array construction algorithms (SACAs) has become a critical and challenging problem.

Thirteen years after the introduction of suffix arrays, three research groups—Ko and Aluru [3], Kärkkäinen and Sanders [4], and Kim et al. [5]–independently achieved the first linear time algorithm for suffix sorting over integer alphabets. This significant breakthrough reduced the time complexity of suffix array construction algorithms from $\mathcal{O}(n\log(n))$ to $\mathcal{O}(n)$. These sequential algorithms are asymptotically optimal in terms of time complexity.

Moreover, several lightweight algorithms [6–9] with small working space requirements were subsequently developed. Notably, Nong et al. [10] achieved $\mathcal{O}(1)$ space complexity for constant alphabets, and Li et al. [11] achieved $\mathcal{O}(1)$ in-place sorting for integer alphabets with read-only inputs. It took approximately thirteen years to reduce the working space from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. These advancements in space efficiency have further enhanced the practicality and applicability of SACAs.

Numerous parallel SACAs have been developed to leverage the power of parallel computing. For instance, Futamura et al. [12] made early attempts to implement a parallel SACA using the sequential prefix-doubling method. Shun's problem-based benchmark suite (PBBS) [13] utilized the task-parallel Cilk Plus programming model to implement a parallel multicore skew algorithm. Osipov [14] and Deo and Keely [15] implemented parallel versions of the Difference Cover 3 algorithm [16] or skewed algorithm on GPUs. Homann et al. [17] introduced the mkESA tool for multithreaded CPUs, which parallelized the sequential induce copy method. Lao et al. [18,19] implemented a parallel recursive algorithm for multicore computers.

Although these parallel methods have demonstrated improved practical performance compared to their sequential counterparts, none of them can efficiently handle very large strings using a large number of processors (p) in $\mathcal{O}\left(\frac{n}{p}\right)$ time. To achieve scalable performance, there is a need for a parallel SACA with $\mathcal{O}\left(\frac{n}{p}\right)$ time complexity. This paper aims to address this gap and presents significant contributions in the following areas.

- We propose a high-level parallel suffix sorting framework. This framework aims to divide a large string's suffix sorting problem with time $T(n,p)$ into many evenly-sized reduced

* Corresponding author.
E-mail addresses: zhihui.du@njit.edu (Z. Du), zhangs@oneonta.edu (S. Zhang), bader@njit.edu (D.A. Bader).

https://doi.org/10.1016/j.future.2023.08.009
0167-739X/© 2023 Elsevier B.V. All rights reserved.

sub-problems with time $T\left(\frac{n}{p}, 1\right)$, and the large problem can be solved by handling many reduced sub-problems on p processors in parallel. In other words, it satisfies $T(n, p) = T\left(\frac{n}{p}, 1\right)$.
- We develop a parallel-inducing sort mechanism *Tunnel*. The *Tunnel* mechanism can support parallel sorting of a group of suffix subsets based on another group of suffix subsets whose order can be determined by their much shorter prefixes.
- We design a total order strategy. A partial order relation can be evolved into a total order relation by iteratively employing our parallel-inducing sort method on different suffix subsets.
- We present the first parallel suffix array construction algorithm, *Tunnel*, with a time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$. *Tunnel* is optimal in terms of asymptotic time complexity.

These contributions collectively address the need for a scalable and efficient parallel suffix array construction algorithm, providing a framework, mechanisms, and strategies that enable the sorting of large strings in parallel with a time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$.

2. Problem description

We first give some basic definitions and notations to present the problem clearly.

Definition 1. Suffix Array: Given a string $S = S[0..n-1]$ with n characters, the string's suffix array (SA) is an array of integers providing the indices of suffixes of S in lexicographical order. This means that $\forall i < j$, we have Suf($SA[i]$) $<$ Suf($SA[j]$), where Suf(k) means the suffix starting at position k.

In this paper, $SA[i]$ is also called the *rank* of $Suf(i)$. For simplicity, we also use index i to stand for suffix i if the context is clear.

Definition 2. Integer Alphabet with Read-Only Input: The alphabet Σ is a set of characters ($\Sigma \subseteq \mathbb{Z}$) that can be used to build a string. Given a string $S = S[0..n-1]$ with n characters, $\forall S[i], 0 \leq i < n$, we have $S[i] \in \Sigma$. The integer alphabet with read-only input means that the given input string S cannot be changed during building its suffix array. Since different characters can be encoded as different integers, we assume $\forall S[i]$, we have $S[i] \in \{x | 1 \leq x \leq |\Sigma|\}$.

In this paper, our problem is based on an integer alphabet with read-only input instead of a constant alphabet, which has only constant characters, or an integer alphabet whose input strings can be updated during the sorting procedure. Either the constant or the integer alphabet is a special case of our problem.

The proposed problem is as follows: Given a very large string S built from an integer alphabet Σ with length n and a parallel random access machine (PRAM) with p processors, can we design a parallel algorithm to construct the suffix array of S in $\mathcal{O}\left(\frac{n}{p}\right)$ time without modifying the input string during the sorting procedure?

In other words, the goal is to develop an efficient large-scale parallel algorithm that constructs the suffix array of S on p processors, ensuring that the time complexity is proportional to $\frac{n}{p}$ while preserving the integrity of the original string S. The algorithm should exploit the parallelism offered by the PRAM model to achieve scalable performance on large-scale inputs.

3. Essential idea and algorithm framework

Unlike existing parallel SACAs, our approach does not aim to explore parallelism within the framework of sequential SACAs. Instead, we propose a parallel framework that divides the entire problem into multiple reduced sub-problems. We then develop an approach to achieve three sub-objectives, solving the final problem step by step by simultaneously handling different sub-problems in parallel.

The essential idea and novelty of our method encompass three major aspects:

- Develop a sampling method to select suffixes from the original string of length n and generate p substrings, each with a length of approximately $\mathcal{O}\left(\frac{n}{p}\right)$, where p represents the total number of processors. This ensures that the size of each sub-problem remains within the range of $\mathcal{O}\left(\frac{n}{p}\right)$.
- Devise a parallel-inducing sort method that generates partial suffix arrays with a partial order. The parallel-inducing sort method guarantees a linear execution time in relation to the size of the largest sub-problem.
- Design a workflow strategy that leverages the partial order relation obtained at each step to generate new sub-problems, thereby progressively achieving a total order relation and efficiently solving the final problem.

In the remainder of this paper, we will use the symbol D to represent a given constant value.

Definition 3. D-Substring and D-String: For a string S with length(S) $= n$ and its suffix subset SubSet, $\forall$Suf(i) $\in$ SubSet, its D-Substring D-Sub(i) is the longest prefix of Suf(i) that satisfies two requirements: (1) the length of D-Sub(i) cannot exceed D; (2) D-Sub(i) cannot have any overlapping with other suffixes in SubSet. The D-String D-Str(SubSet) is defined as the concatenated string of all D-Substrings of suffixes in SubSet according to their indices in S.

Fig. 1 provides a simple example illustrating how a suffix is mapped to a much shorter D-Substring and how a subset of suffixes is mapped to a much shorter D-String. For a given string S = "bananananana" and a suffix subset $SubSet = \{1, 2, 5, 11\}$, based on Definition 3, D-$Sub(1)$ = "a" as it has only one character since the next suffix is also in the subset. D-$Sub(2)$ = "na" has two characters since the maximum length of a D-Substring is D which sets to 2 in the example.

Definition 4. Partial Suffix Array: Given a suffix subset SubSet of a string S with $|\text{SubSet}| = N_s$, the partial suffix array (PSA) is an array of integers that provides the indices of suffixes in SubSet in lexicographical order. This means that $\forall\ 0 \leq i < j < N_s$, we have Suf(PSA[$i$]) $<$ Suf(PSA[j]). At the same time, Suf(PSA[i]) and Suf(PSA[j]) must be two suffixes in SubSet.

A partial suffix array indicates the order of suffixes within a suffix subset. However, the order of suffixes not included in the subset remains unknown.

In this paper, we generate different D-strings to represent the reduced sub-problems. D-strings significantly reduce the size of the original problem.

Definition 5. Order of Suffix Sets: Given two non-empty suffix sets Set_1 and Set_2 of a string S, if for all $x \in Set_1$ and $y \in Set_2$, their lexicographical order satisfies $x < y$ (or $x > y$), then we define $Set_1 < Set_2$ (or $Set_1 > Set_2$).

Ensuring the correct order of a group of suffix sets is an important step in our workflow strategy. This step plays a crucial role in achieving the total order of the suffix set.

String	:	b	a	n	a	n	a	n	a	n	a	n	a
Index	:	0	1	2	3	4	5	6	7	8	9	10	11
Suffix Subset	:		1,	2,			5,						11
D-Substring	:		a,	na,			an,						a
D-String	:		a	n	a	a	n	a					

Fig. 1. An example of D-Substrings and D-String with $D = 2$ based on the given string S = "bananananana" and a suffix subset = $\{1, 2, 5, 11\}$.

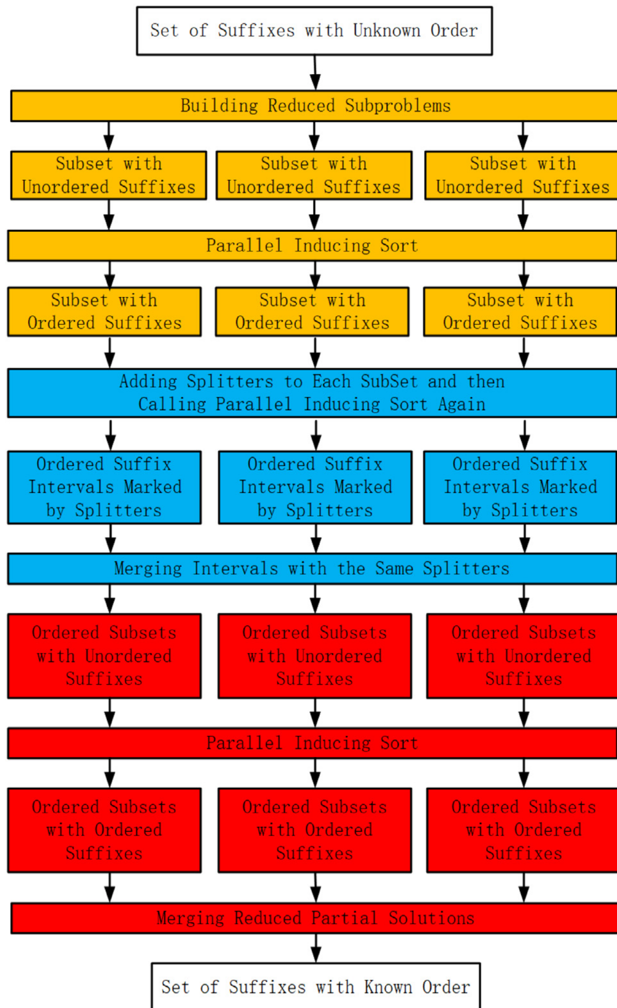

Fig. 2. Key idea and the high-level workflow description of the proposed method.

3.1. High-level algorithm overview

Our method, based on the concepts of the D-string and the order of suffix sets, is succinctly illustrated in Fig. 2. The figure presents the core idea of our approach.

The problem is formulated as a sorting challenge for a set of suffixes within a given string. To tackle this challenge, we employ three major sorting steps. In each step, we divide the complete set of suffixes into multiple subsets of roughly equal size by using different D-strings to replace the original string. This not only reduces the number of suffixes but also shortens their length. Our algorithm features a novel parallel-inducing sort method, which operates on all the subsets and ensures that the suffixes are correctly ordered in each subset. This parallel-inducing sort is the heart of our algorithm and will be thoroughly described in Section 4. The parallel-inducing sort transforms the subsets with an unknown suffix order into subsets with ordered suffixes, effectively establishing many partial order relations over the complete suffix set. By applying the parallel-inducing sort on different subsets three times, we can establish the total order relation over the complete set of suffixes.

The first step of our method, shown in yellow in Fig. 2, generates the order of suffixes for given suffix subsets. Once the order of suffixes in each subset is determined, we divide them into equal-sized intervals, with the suffixes between connected intervals serving as "splitters". By adding the splitters from other subsets and applying our parallel-inducing sort method once more, we can further refine the intervals, enabling us to distinguish the order of suffixes within different intervals based on the order of the splitters. This is the second step, marked in blue in Fig. 2.

In the third step, marked in red in Fig. 2, we merge the intervals with the same splitters to create new groups of ordered suffix subsets. However, the order of suffixes within each subset remains unknown. To determine this order, we once again apply our parallel-inducing sort method on the subsets whose order is already known. This results in both the ordering of the subsets and the suffixes within each subset. Finally, we merge these ordered subsets to obtain the final ordered suffixes or suffix array.

Algorithm 1: Framework of *Tunnel* Algorithm

1 **Function** Tunnel(*String, p*)
2 Step (1) Sort unordered subsets
3 1.1 Evenly divide all suffixes of *String* into p suffix subsets $SubSet_0,...,SubSet_{p-1}$, $\forall 0 \le i \le p-1$, $|SubSet_i| = \mathcal{O}(\frac{n}{p})$ and add them into the $SetList_1$
4 1.2 Call the parallel-inducing sort function to generate the partial suffix array for each set $SubPSA=PIS(SetList_1)$
5 Step (2) Sort the subsets with splitters
6 2.1 Select p splitters from each processor p_i based on *SubPSA*; add them into the splitter set *SplitSet*; add all splitters into each subset
7 2.2 Call the parallel-inducing sort function again to generate the partial suffix array of all subsets, including the splitters $SplPSA=PIS(SetList_2)$
8 Step (3) Sort ordered subsets
9 3.1 Assign all suffixes into p ordered subsets that meet $OSubSet_0 <...< OSubSet_{p-1}$ according to *SplPSA*; add them into $SetList_3$
10 3.2 Call the parallel-inducing sort function to generate the partial suffix array meeting total order $TolPSA=PIS(SetList_3)$
11 3.3 Generate the final *SA* based on *TolPSA*
12 **return** *SA*
13 **end**

We have developed a novel algorithm, *Tunnel*, to efficiently construct the suffix array of a given string. In Alg. 1, we present the framework of this parallel suffix array construction method. This framework transforms the large-scale problem, where the size of the problem is represented by n and the number of processors by p on a PRAM machine, into p smaller parallel problems of size $\mathcal{O}\left(\frac{n}{p}\right)$ that can be handled by one processor each.

The framework in Alg. 1 follows the three big steps outlined in Fig. 2. The first step is described in lines 2 to 4, where all n suffixes of the string S are evenly divided into p subsets. The *PIS* function is then applied in line 4 to generate the partial suffix arrays for each of the p subsets. The second step is described in lines 5 to 7, where $(p-1)$ splitters together with the largest suffix are selected from the partial suffix arrays and merged with the existing subsets. The *PIS* function is then applied again to generate the *SpliPSA* partial suffix array that reflects the order of suffixes in each subset, including the splitters.

In the third step, described in lines 8 to 11, the splitters are used to assign all suffixes into p ordered subsets. The *PIS* function

Table 1
Example string “bananabananaanannana” with its suffix indices.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	a	n	a	n	a	b	a	n	a	n	a	a	n	a	n	n	a	n	a

Table 2
An input string of step 1 and the output partial order results.

P 0	suffix index	0	1	4	5	8	9	12	13	16	17
	D-substring	b	an	n	ab	n	an	a	na	n	an
	D-string	bannaabnananan									
	partial suffix array	5	17	9	1	12	0	4	16	8	13
P 1	suffix index	2	3	6	7	10	11	14	15	18	19
	D-substring	n	an	b	an	n	aa	a	nn	n	a
	D-string	nanbannaaannna									
	partial suffix array	19	11	3	7	14	6	18	10	2	15

is then applied again to generate the *TolPSA* partial suffix array that meets the total order requirement. Finally, the partial suffix arrays are merged into one array, *SA*, in line 11.

3.2. An illustration example

Consider the example of our proposed algorithm applied to an input string, “bananabananaanannana”, with 20 suffixes. This algorithm employs two processors and has a value of $D = 2$ (here, we let $p = D$ just for simplicity. Under large-scale parallel scenarios, we have $p \gg D$). Table 1 lists the suffix indices for this input string.

The proposed framework begins by assigning the suffixes to two processors. In this example, we employ the *CYCLIC(D)* distribution (see Section 6.1) to assign suffixes to different processors, simplifying the subsequent parallel-inducing sort procedure.

In our example, processor 0 is assigned the following suffixes: [0, 1, 4, 5, 8, 9, 12, 13, 16, 17], while processor 1 is assigned: [2, 3, 6, 7, 10, 11, 14, 15, 18, 19].

Based on their respective suffix subsets, both processors compute the corresponding *D*-substrings and generate their *D*-strings to represent their suffixes.

Based on Definition 3, the *D*-substrings on processor 0 are “b”, “an”, “n”, “ab”, “n”, “an”, “a”, “na”, “n”, “an”, and they will form a *D*-string “bannabnanananan”. Processor 1’s *D*-substrings are “n”, “an”, “b”, “an”, “n”, “aa”, “a”, “nn”, “n”, “a” and the *D*-string is “nanbannaaannna”. Table 2 presents the input suffixes, *D*-substrings, *D*-strings, and the partial suffix array of suffixes on each processor after executing the parallel-inducing sort procedure.

The first round of parallel-inducing sort results in the ordered suffixes on processor 0 as [5, 17, 9, 1, 12, 0, 4, 16, 8, 13], and on processor 1 as [19, 11, 3, 7, 14, 6, 18, 10, 2, 15]. However, the final ordering of the total suffix set can only be determined after the subsequent processing, as the ordering between suffixes from different processors has not been established yet.

In the second step, suffix 12 serves as the splitter on processor 0, dividing the ordered suffixes into two approximately equal parts, while suffix 14 serves as the splitter on processor 1. Together with the two largest suffixes [13, 15] on the two processors, suffixes in [12, 13, 14, 15] are added to the original suffix set on both processors 0 and 1.

On processor 0, the resulting suffixes are [0, 1, 4, 5, 8, 9, 12, 13, 14, 15, 16, 17] with the *D*-substrings “b”, “an”, “n”, “ab”, “n”, “an”, “a”, “n”, “a”, “n”, “na”, “an” and the *D*-string “bannabnanananannaan”.

On processor 1, the suffixes are [2, 3, 6, 7, 10, 11, 12, 13, 14, 15, 18, 19] with the *D*-substrings “n”, “an”, “b”, “an”, “n”, “a”, “a”, “n”, “a”, “nn”, “n”, “a” and the *D*-string “nanbannaananna”.

Table 3
An input string of step 2 and the output partial order results.

P 0	suffix index	0	1	4	5	8	9	12	13	14	15	16	17
	D-substring	b	an	n	ab	n	an	a	n	a	n	na	an
	D-string	bannabnanananannaan											
	partial suffix array	5	17	9	1	12	14	0	4	16	8	13	15
P 1	suffix index	2	3	6	7	10	11	12	13	14	15	18	19
	D-substring	n	an	b	an	n	a	a	n	a	nn	n	a
	D-string	nanbannaananna											
	partial suffix array	19	11	3	7	12	14	6	18	10	2	13	15

Table 4
Ordered suffix intervals after step 2.

Processor	Unordered Suffixes	Splitter	Splitter	Un-ordered Suffixes	Splitter	Splitter
P 0	5,17,9,1	12	14	0,4,16,8	13	15
P 1	19,11,3,7	12	14	6,18,10,2	13	15

Another round of parallel-inducing sort is performed on the two *D*-strings, resulting in the sorted suffixes [5, 17, 9, 1, 12, 14, 0, 4, 16, 8, 13, 15] on processor 0 and [19, 11, 3, 7, 12, 14, 6, 18, 10, 2, 13, 15] on processor 1.

Table 3 shows the input suffixes, *D*-substrings, *D*-strings, and partial suffix arrays for the two processors after the parallel-inducing sort procedure is applied. By using the splitters [12, 13, 14, 15], the suffixes are divided into six ordered intervals (see Table 4), enabling the assignment of the suffixes into two ordered subsets [5, 17, 9, 1, 19, 11, 3, 7, 12, 14] and [0, 4, 16, 8, 6, 18, 10, 2, 13, 15]. Here, the subsets are ordered, but the suffixes in each subset may not be fully ordered.

The parallel-inducing sort is applied again to the two processors in the third step. On processor 0, the suffixes [1, 3, 5, 7, 9, 11, 12, 14, 17, 19] are associated with the *D*-substrings “an”, “an”, “ab”, “an”, “an”, “a”, “an”, “an”, “an”, “a”. The *D*-string for this step is “ananabananaananana”. On processor 1, the suffixes are [0, 2, 4, 6, 8, 10, 13, 15, 16, 18] with the *D*-substrings “ba”, “na”, “na”, “ba”, “na”, “na”, “na”, “n”, “na”, “na”. The *D*-string for this step is “bananabananananannana”.

The result of the parallel-inducing sort on these two *D*-strings yields the ordered suffixes [19, 11, 5, 17, 9, 3, 7, 1, 12, 14] on processor 0 and [6, 0, 18, 10, 4, 16, 8, 2, 13, 15] on processor 1. The final suffix array is the combination of these two ordered sets: [19, 11, 5, 17, 9, 3, 7, 1, 12, 14, 6, 0, 18, 10, 4, 16, 8, 2, 13, 15].

Table 5 displays the input suffixes, *D*-substrings, *D*-strings, and the final order of the suffixes after the third step of the parallel-inducing sort procedure.

In summary, this example demonstrates a step-by-step process to determine the final order of suffixes through the division of suffixes into two processors, the calculation of *D*-substrings and *D*-strings, and multiple rounds of the parallel-inducing sort.

4. Algorithm kernel development

In this section, we will explore the inner workings of our innovative method called *Parallel-Inducing Sort*, which serves as the cornerstone of our *Tunnel* algorithm. This powerful technique enables efficient parallel sorting of large strings and is invoked multiple times during the execution of our algorithm.

To grasp the essence of our approach, we will begin by introducing the underlying data structure employed in our method. Understanding this structure is crucial for comprehending the

Table 5
Input string of step 3 and the output total order results.

P 0	suffix index	1	3	5	7	9	11	12	14	17	19
	D-substring	an	an	ab	an	an	a	an	an	an	a
	D-string	ananabananaananana									
	partial suffix array	19	11	5	17	9	3	7	1	12	14
P 1	suffix index	0	2	4	6	8	10	13	15	16	18
	D-substring	ba	na	na	ba	na	na	na	n	na	na
	D-string	bananabanananannana									
	partial suffix array	6	0	18	10	4	16	8	2	13	15

inner workings of our algorithm. Subsequently, we will provide a comprehensive explanation of the kernel algorithm and its various components. By delving into the details of these components and their interactions, we can develop a clear understanding of how the *Parallel-Inducing Sort* method functions and how it significantly contributes to the overall efficiency of the *Tunnel* algorithm.

Before delving into the key data structures, let us establish some concept definitions to provide a clear expression of our ideas.

4.1. Definition

Definition 6. *D*-Prefix: For a suffix $Suf(i)$ of a string S with length n, where $0 \leq i < n$, the D-prefix, denoted as D-Pre(i), is defined as the longest prefix of $Suf(i)$ that contains no more than D characters.

Our algorithm is driven by the rationale that when the *D*-prefixes of specific suffixes differ, we can utilize these *D*-prefixes instead of the entire suffixes to establish their relative order. By leveraging this approach, we simplify the sorting problem and reduce the size of the suffixes involved, which forms the core concept underpinning our algorithm.

Definition 7. Distinguishable Suffix and Equivalent Suffix: Let SufSet be a set of suffixes in string S. For any Suf(x) in SufSet, if the *D*-prefix of Suf(x) differs from the *D*-prefixes of all other suffixes in SufSet, then Suf(x) is referred to as a distinguishable suffix in SufSet. Otherwise, it is classified as an equivalent suffix in SufSet.

According to Definition 7, it is possible for the same suffix to be considered distinguishable within one set of suffixes, while being regarded as equivalent within another set of suffixes. This is a challenge for our algorithm, and we will delve into the specific details of how we intend to address this issue.

Definition 8. Modulus-Equal Suffixes and Modulus-Equal Suffix Set: Suffixes Suf(i) and Suf(j) are considered modulus-equal suffixes if mod(i,D) = mod(j,D). Given a suffix set SufSet, the modulus-equal suffix subset with modulus d is defined as $M_d(\text{SufSet}) = \{\text{Suf(k)} \mid \text{Suf(k)} \in \text{SufSet} \wedge \text{mod(k,D)} = \text{d}\}$, where $0 \leq d < D$.

An important property of modulus-equal suffixes is that their *D*-prefixes do not overlap with each other. Consequently, we can concatenate their *D*-prefixes to restore a longer suffix. By leveraging this property, we can break down a large suffix sorting problem into multiple smaller sub-problems, each involving the sorting of significantly smaller *D*-prefixes. This approach allows us to induce the relative order of multiple long suffixes based on the sorting of their corresponding short *D*-prefixes.

Definition 9. Suffix Predecessor and Successor: For two modulus-equal suffixes Suf(x) and Suf(y), where $x < y$, Suf(x) is considered a predecessor of Suf(y). Similarly, Suf(y) is referred to as a successor of Suf(x). The distance between Suf(x) and Suf(y) is denoted as $y - x$, and if we let $z = \frac{y-x}{D}$, then Suf(y) is the zth successor of Suf(x).

To determine the relative order of two suffixes with shared *D*-prefixes, we can compare the order of their first successors. This comparison process can be carried out recursively until the successors have distinct *D*-prefixes. By employing this approach, we can establish the order of long common prefix suffixes along with their successors that possess different *D*-prefixes. However, this straightforward method is not efficient and does not guarantee linear time complexity. To overcome this limitation, we have devised a highly efficient inducing method that replaces the step-by-step successor checking method. This approach significantly improves the overall efficiency of the algorithm while achieving the desired results.

Definition 10. Equivalent Group: Given a suffix set *Grp* with $|Grp| \geq 2$, if all suffixes in *Grp* have the same *D*-prefix, then *Grp* is named as an equivalent group.

To handle suffixes with common prefixes longer than size *D*, we employ equivalent groups to organize such suffixes. For these suffixes, we utilize a *Tunnel* mechanism (see Section 4.3) to determine their relative order based on the *D*-prefixes of their successors.

Definition 11. Inducing Set and Unique Inducing Set: Given an equivalent group (EG) of a string S with length n, for any positive integer z, the set $\text{IS}_z(\text{EG}) = \{\text{Suf(x)} \mid \forall\ \text{Suf(y)} \in \text{EG}, \text{Suf(x)}$ is the zth successor of Suf(y)$\}$ represents the zth inducing set of EG. Similarly, the set $\text{UIS}_z(\text{EG}) = \text{IS}_z\text{-EG-} \cup_{x=1}^{z-1} \text{UIS}_x$ represents the zth unique inducing set of EG. Here, z refers to the index of the corresponding inducing or unique inducing set. All inducing sets or unique inducing sets form an inducing set series or unique inducing set series.

To determine the order of suffixes within an equivalent group, we utilize the *D*-prefix order of their corresponding suffixes in their first inducing set. If a new equivalent group emerges in the first inducing set, we repeat the procedure until no more equivalent groups are found. This procedure creates an inducing tunnel through which we can determine the order of suffixes in the equivalent group. However, if two suffixes within the equivalent group are modulus-equal suffixes, the successor suffix may appear again in an inducing set of the equivalent group, leading to duplicates in the tunnel. These duplicates can increase the time complexity of our algorithm. To mitigate this issue, we introduce a unique inducing set (*UIS*) that excludes any suffixes already present in their equivalent group or their previous unique inducing set.

By definition, UIS_z is a subset of IS_z, at the same time, it cannot contain any element already present in $UIS(z-k)$ and EG, where $1 \leq k < z$. Consequently, any two unique inducing sets will have no overlapping elements. Our parallel-inducing method employs unique inducing sets instead of inducing sets to reduce time complexity.

Definition 12. Inducing Successor: Given an equivalent group (EG), for every $\text{Suf(x)} \in \text{EG}$, its inducing successor is denoted as its closest successor $\text{Suf(y)} = \text{Suf}(x + z \times \text{D})$, where Suf(y) is a distinguishable suffix in $\text{IS}_z(\text{EG})$.

The relative order of suffixes within an equivalent group can be determined based on the order of their inducing successors in the corresponding inducing sets instead of sorting all inducing sets. Therefore, accurately identifying the inducing successor is crucial for inducing the order of suffixes in an equivalent group efficiently. In Section 4.3.3, we will explore how our proposed *Tunnel* mechanism effectively identifies the inducing successors.

4.2. Data structure design

We propose the following data structure design based on the definitions provided above.

Given an input string S with a size of $n = length(S)$ and $Suf(e) = S[e..n-1]$, where $0 \leq e \leq n-1$ is the eth suffix, the goal is to sort the suffixes in parallel using p processors.

To keep track of the global indices of suffixes assigned to each processor, we utilize a two-dimensional array called *ParSuf*. The corresponding suffix array is represented by another two-dimensional array called *ParSA*. Specifically, if *ParSuf[i][j]* $= k$, it means that the jth local suffix assigned to processor i is the kth global suffix *Suf(k)*. Similarly, if *ParSA[i][l]* $= t$, it implies that the lth sorted suffix from smallest to largest on processor i is the tth global suffix *Suf(t)*.

The algorithm consists of three main steps to process different subsets of suffixes. Each step requires a pair of *ParSuf* and *ParSA* arrays to maintain the corresponding suffix subsets and their partial suffix arrays.

In addition to the *ParSuf* and *ParSA* arrays, another two-dimensional array, *ParEG*, is utilized to store the equivalent groups. For each processor i with N_g different equivalent groups, *ParEG[i][g]* represents the gth equivalent group on processor i. *ParEG[i][g].size* indicates the number of suffixes in group g, and *ParEG[i][g].head* $= t$ specifies that *ParSA[i][t]* is the first suffix in the group. The suffixes from *ParSA[i][t]* to $ParSA[i][t+ParEG[i][g].size-1]$ have the same D-prefix on processor i.

In the output of the first and second step of our parallel inducing algorithm, different *ParEG* arrays are used to store different equivalent groups based on the input suffixes.

To generate the inducing successor array of an equivalent group *ParEG*, a corresponding two-dimensional array, *ParIS*, is used to hold the inducing successors. If *ParIS[i][j]* $= t'$ and *ParSuf[i][j]* $= t$, it means that the suffix *Suf(t')* is the inducing successor of the suffix *Suf(t)* on processor i.

To generate the *ParIS* array, we require additional data structures. A two-dimensional array, *ParFlag*, is used to indicate if a particular suffix can be an inducing successor. Specifically, if *ParFlag[i][j]* $=$ *True*, it means that suffix *Suf(ParSuf[i][j])* can be used to determine the order of suffixes in an equivalent group. Otherwise, *ParFlag[i][j]* $=$ *False*. Additionally, a one-dimensional array, *ShaFlag*, is used to store the shared flag information, where *ShaFlag[s]=True* if there exists at least one suffix *ParSuf[i][j]* such that *ParSuf[i][j]=s* and *ParFlag[i][j]=True*. Conversely, if no such suffix exists, *ShaFlag[s]* is set to *False*.

4.3. Parallel-inducing sort

In this subsection, we will present the key components of our *Tunnel* algorithm's parallel-inducing sort method. We will begin by outlining the general structure of the method, followed by a detailed explanation of the various functions that constitute the method.

Alg. 2 introduces the *PIS* function, which implements parallel-inducing sort on reduced sub-problems. The core concept of this function is to construct smaller D-strings for each of the p sub-problems and utilize the D-prefixes to categorize the suffixes into two groups: distinguishable suffixes with unique D-prefixes and equivalent suffixes with identical D-prefixes. While the order of distinguishable suffixes can be established unambiguously, the challenge lies in determining the order of equivalent suffixes, which is the primary focus of the function. To overcome this challenge, we generate unique inducing sets for each equivalent group and identify the inducing successor for each equivalent suffix. Finally, we use these results to induce the order of equivalent suffixes within each equivalent group. By combining the order of both distinguishable and equivalent suffixes, we obtain the final suffix array.

The *PIS* function, outlined in line 2 of Alg. 2, sorts all the suffixes in the *SetList* based on their D-prefixes by calling the sub-function *D_PIS*. This operation yields two outputs: *D_ParSA* with the *ParSA* data structure, and *D_ParEG* with the *ParEG* data structure. In line 3, the function *BldIT* follows the inducing tunnel for suffixes in equivalent groups *D_ParEG*, generating the unique inducing sets for identifying the inducing successor of each equivalent suffix.

The unique inducing sets are sorted using the *D_PIS* function again, resulting in a new partial suffix array, *U_ParSA*, and the equivalent groups, *U_ParEQ* for the unique inducing sets. In line 5, the *LocateIS* sub-function is called on these structures, as well as the original equivalent groups, *D_ParEQ*, to obtain the inducing successor array, *ParIS*. The order of the suffixes in the equivalent groups *D_ParEG* can be induced by calling the sub-function *E_PIS* in line 6, utilizing the inducing successor array, *ParIS*. Finally, the final partial suffix array is obtained by updating *D_ParSA* with *E_ParSA* in line 7.

Algorithm 2: Parallel-Inducing Sort

```
1 Function PIS(SetList)
2   (D_ParSA,D_ParEG)=D_PIS(SetList)
3   UniIndSetList=BldIT(D_ParEG)
    /* Build unique inducing set list
       UniIndSetList=[UniIndSet_0,...,UniIndSet_{p-1}] for all
       equivalent groups                                        */
4   (U_ParSA,U_ParEG)=D_PIS(UniIndSetList)
5   ParIS=LocateIS(D_ParEG,U_ParSA,U_ParEG)
    /* Get the inducing successor array to induce the order of
       suffixes in equivalent groups.                           */
6   E_ParSA=E_PIS(D_ParEG,ParIS)
7   Build ParSA based on D_ParSA and E_ParSA
8   return ParSA
9 end
```

In the subsequent subsections, we will delve into the key components of the *PIS* algorithm, providing a more in-depth understanding of its functioning and effectiveness.

4.3.1. Sorting D-prefixes

The initial phase of this function shown in Alg. 3 involves constructing p significantly smaller D-strings, referred to as DS_S_0, ..., DS_S_{p-1}, from the original string S. This division allows each processor to process one of these D-strings concurrently (as seen in line 2). The original suffixes are then substituted with their D-substrings.

The parallel execution of D-prefix based sorting will be carried out across all processors (from lines 3 to 20). On each processor, we will utilize the existing optimal sequential SACA algorithm, *SeqOptSA* [11], to generate the suffix array for the corresponding D-string. Since the suffix array will include some suffixes that are not included in the assigned subset, we call the original suffix array an extended suffix array. We need to eliminate unnecessary suffix entries from the extended suffix array to obtain the exact suffix array *SA* (as seen in line 5).

Suffixes whose order can be determined based on their D-prefixes will have the correct rank in the partial suffix array. However, in cases where multiple suffixes have identical D-prefixes,

their ranks in *SA* may not be accurate. In such scenarios, the order of these equivalent suffixes must be determined based on their full suffixes. To cluster equivalent suffixes based on their *D*-prefixes, we use a two-dimensional array, *EquGrp*, to store these groups. The equivalent suffixes of the current group *cg* for processor *i* will be stored at *EquGrp[i][cg]*.

Next, we will identify the equivalent groups whose suffixes may not be correctly ordered based on their *D*-prefixes (from lines 6 to 19). To start, we initialize the equivalent group number to -1 and set the new group flag, *NewGroup*, to be *True*. Then, we compare each suffix with its previous suffix in the order of the partial suffix array *SA*, examining if their *D*-prefixes are the same (from lines 7 to 19). If they are equal (from lines 8 to 15), for a new equivalent group (from lines 9 to 13), we increment the group ID *EqG* by 1 and set the new group flag to *False*. At the same time, we set the head position of the current equivalent group to $j-1$ and initialize the total number of elements in the group to 1. Afterward, we increase the total number of elements in the current equivalent group by 1 (line 14). If the *D*-prefix of the current suffix is different from the *D*-prefix of its previous suffix, we reset the new equivalent group flag to be *True* (line 17). Finally, the function returns the partial suffix array *SA* and the equivalent group array *EquGrp* (line 21).

The *EquGrp* feature plays a crucial role in efficiently traversing all equivalent suffixes within the current processor. Additionally, it can be used to build a valuable one-to-one mapping from the ranks of a partial suffix array to global suffix indices. Leveraging this information, we can construct a reverse one-to-one mapping, enabling us to locate the local suffix rank when provided with a global suffix index. This reverse mapping capability proves to be highly advantageous in various scenarios, facilitating quick and precise access to local suffix information. By harnessing the power of both mappings, we enhance the effectiveness of our approach and achieve more efficient and accurate processing of suffixes within the system.

Algorithm 3: Sorting *D*-Prefixes

```
1  Function D_PIS(SetList)
2      Build p D-strings DS_S0,...,DS_Sp−1 according to different subsets
3      forall (i in 0..p-1) do
4          ESA[i][] = SeqOptSA(DS_Si)
5          Remove the entries not in Set_i from ESA[i][] and build SA[i][]
            corresponding to SubSet_i
6          EqG = -1; NewGroup = True
7          for (j in 1..|SubSet_i|-1) do
8              if (Suf(SA[i][j]) and Suf(SA[i][j-1]) have the same D-prefix) then
9                  if (NewGroup == True) then
10                     EqG++; NewGroup = False
11                     EquGrp[i][EqG].head = j-1
12                     EquGrp[i][EqG].size = 1
13                 end
14                 EquGrp[i][EqG].size++
15             end
16             else
17                 NewGroup = True
18             end
19         end
20     end
21     return SA, EquGrp
22 end
```

4.3.2. Building inducing tunnel

The primary objective of this procedure is to generate unique inducing suffix sets and evenly distribute them among processors. While the unique inducing sets within the same equivalent group do not overlap, different unique inducing sets from distinct equivalent groups may have some overlap with each other. Assigning these overlapping unique inducing sets to the same processors can effectively reduce the total number of duplicated suffixes. However, achieving a balance in workload distribution while minimizing the duplication of suffixes poses a significant challenge.

Based on Definition 11 about the unique inducing set, the unique inducing set with a larger index number z may have a much less number of suffixes than the unique inducing set with a smaller z. So, we can divide the z indices into three intervals based on the potential largest number of suffixes in those unique inducing sets. We can find a constant C that meets the following requirements.

$$\sum_{x \in EG_i} |\mathrm{UIS}_z(\mathrm{x})| \leq \begin{cases} \mathcal{O}(\frac{n}{p}) : z \leq C \\ \mathcal{O}(\frac{n}{p^2}) : C < z \leq C \times p \\ \mathcal{O}(1) : z > C \times p \end{cases}$$

where EG_i is the set of equivalent groups in processor i. The number of unique inducing sets with an index $z \leq C$ is very limited. However, these sets may have a large number of suffixes, typically up to the order of $\mathcal{O}\left(\frac{n}{p}\right)$ in the worst case. On the other hand, for the unique inducing sets with an index ranging from $C < z \leq C \times p$, the unique inducing sets sharing the same index z may have approximately $\mathcal{O}(\frac{n}{p^2})$ suffixes at most. In both cases, when considering each processor, the sum of all suffixes within the range of unique inducing sets ($1 \leq z \leq C \times p$) cannot exceed $\mathcal{O}\left(\frac{n}{p}\right)$. Therefore, it is practical to assign these unique inducing sets to the same processors as their corresponding equivalent groups.

To optimize the processing of unique inducing sets with an index (z) exceeding $C \times p$, we can follow a series of steps. Firstly, we identify the starting and ending suffixes of these sets, ensuring that they share the same value for modulo D. Next, we align these sets based on the degree of overlap they have with subsequent unique inducing sets. Finally, by distributing the aligned unique inducing sets across multiple processors, we can achieve load balancing and effectively reduce redundancy.

Alg. 4 presents the procedure for constructing the inducing tunnel for all equivalent groups. We leverage parallel processing to handle all the suffixes concurrently in different processors (lines 2 to 9). Firstly, in lines 3 to 5, we allocate the unique inducing suffix sets with ranks lower than $C \times p$ to their corresponding equivalent group processors. This step is simple and we will focus on the second part. It is worth noting that while the sizes of the remaining unique inducing suffix sets, identified by indices $z > C \times p$, might be small, their total count can be substantial.

To ensure efficient assignment of these small sets, we have developed a three-step approach:

Step 1 (Line 6): Determining bounds for unique inducing set series. We start by computing the lower and upper bounds of the suffix indices for each unique inducing set series. Due to different modulus values, these series may have up to D pairs of bounds.

Step 2 (Line 7): Computing the alignment solution for unique inducing sets. Using the information from the “Bound” array, we calculate the alignment solution for different unique inducing sets. This enables us to identify which sets should be aligned and assigned to the same processor, effectively reducing duplicated suffixes. We focus on aligning only those unique inducing set series with overlapping suffixes exceeding $\frac{n}{p}$. This ensures that each processor handles suffixes no more than $\mathcal{O}(\frac{n}{p})$. However, aligning additional sets can further reduce the total number of duplicated suffixes.

Step 3 (Line 8): Ensuring load balance through even distribution. In this crucial step, we tackle load balancing by mapping distinct unique inducing sets to individual processors (the mapping method can be used to determine which processor is responsible for handling a particular unique inducing set later). We distribute the aligned unique inducing sets evenly across

Algorithm 4: Building Inducing Tunnel

1 **Function** `BldIT`(*ParEG*)
2   **forall** *(i in 0..p-1)* **do**
3   **forall** z *in 1..C×p* **do**
4   For any equivalent group g in *ParEG[i]*, Add suffixes of $UIS_z(g)$ and g to $UniIndSet_i$
5   **end**
6   Calculate the Bound array for all $z > C \times p$, any equivalent group g in *ParEG[i]*, and $0 \le d < D$; *Bound[i][g][d].l* represents the lower bound of modulus-equal suffixes in group g's unique inducing sets, and *Bound[i][g][d].u* represents the upper bound
7   Calculate alignment solutions based on the major overlapping range of the Bound array. An alignment solution Align = $\{<g_1,...,g_t>, <z_1,...,z_t>\}$ means that groups $g_1, \ldots, g_t$'s unique inducing sets will be aligned from indices $z_1, \ldots, z_x$
8   Distribute aligned unique inducing sets among different processors (assign suffixes on processor i to $UniIndSet_i$)
9   **end**
10   **return** *UniIndSet*
11 **end**

processors using a cyclic approach. This strategic distribution ensures that computational load remains well-balanced across the entire system, enabling each processor to efficiently handle its assigned workload.

By following this three-step approach, we achieve an efficient assignment of small sets, significantly reducing duplication and ensuring that each processor handles an appropriate number of suffixes. This method leads to improved performance and streamlined processing for the given task.

Unique inducing sets that remove the duplicated suffixes from its inducing sets can potentially result in our *Tunnel* mechanism not finding the correct inducing successor. To address this, we introduce a quick check in the "Locating Inducing Successors" *LocateIS* function (set Section 4.3.3) to rectify this problem.

4.3.3. Locating inducing successors

Identifying the inducing successors of an equivalent group serves a crucial purpose and offers a substantial advantage. It allows us to distinguish a set of suffixes that can be utilized to determine the order of suffixes within the given equivalent group. When the suffixes within an equivalent group share long common prefixes, inducing successors can dramatically decrease the search time. The method to locate the inducing successors is outlined in Alg. 5.

In Alg. 5, we assign values to the *ParGFlag* flag array, using the *ParFlag* data structure, based on the information from the partial suffix array (*UniIndSA*) and its equivalent group (*UniIndEG*), as well as the original equivalent group (*ParEG*) which we need to sort. Specifically, *ParGFlag[i][j]* is set to *True* if the suffix *ParSuf[i][j]* is a distinguishable suffix, and *False* otherwise.

At the same time, we establish the shared flag array *ShaGFlag*, implemented using the *ShaFlag* data structure, based on the values present in *ParGFlag* (line 2). This array serves as a shared representation of the flags across all processors. In order to address the potential occurrence of *False* flags in the *ShaGFlag* arising from two suffixes belonging to different inducing sets on the same processor, we invoke the *Remark* subfunction to reset those elements within *ShaGFlag* that were erroneously marked as *False* to their correct state of *True* (see Alg. 7 in details).

To initialize the *ShaIS* array (line 3) in Alg. 5, we assign each element with the index of its corresponding suffix, encompassing all the suffixes in *ParEG* and their respective successors. The *ShaIS* array contains the suffixes to be sorted and their successors. It is shared among all processors.

In line 4, we invoke the *TunnelTrans* subfunction (see details in Alg. 6) to update the *ShaIS* array using the *ShaGFlag*. This update process is in parallel, and it can make sure that even if the size of *ShaIS* is $\mathcal{O}(n)$ (the worst case), it can be handled in $\mathcal{O}\left(\frac{n}{p}\right)$ time.

However, it is important to note that the same suffixes on different processors may have different inducing successors. This may happen for two reasons: the shared splitters and the same suffixes due to being successors of the suffixes in different *EG*s on different processors. Therefore, in line 5, we call the *Refine* subfunction (see details in Alg. 7) to refine the information on the inducing successors. The refined information is then stored in a new array called *ParIS*.

In Alg. 5, by leveraging the information from *UniIndSA*, *UniIndEG*, *ParGFlag*, and *ShaGFlag*, and performing the necessary initialization, updating, and refining steps, we accurately determine the inducing successors of the suffixes in *ParEG*, which are stored in the *ParIS* array (line 6) for the following sorting on suffixes in *ParEG*.

Algorithm 5: Locating Inducing Successors

1 **Function** `LocateIS`(*ParEG,UniIndSA,UniIndEG*)
2   Build *ParGFlag* and *ShaGFlag* arrays; Call *Remark(ParEG, UniIndSA, UniIndEG)* to update *ShaGFlag*
3   Initialize *ShaIS* with the current suffix itself to include all suffixes in *ParEG* and their successors
4   *ShaIS=TunnelTrans(ShaIS,ShaGFlag)*
5   *ParIS=Refine(ShaIS,UniIndEG)*
6   **return** *ParIS*
7 **end**

Alg. 6 describes the details of the *TunnelTrans* function. It plays a crucial role in identifying inducing successors. Since inducing successors can only exist within an equal-modulus suffix set, our first step is to construct these sets (line 2). Subsequently, lines 3 to 24 perform the generation of inducing successors for each equal-modulus suffix set.

To enable parallel execution, we divide each set into p parts (line 4). The key idea behind identifying inducing successors is to transfer the suffixes with a *True* flag to their predecessors with *False* flags. This transfer operation can be performed in parallel (lines 5 to 8).

To facilitate the parallel transformation across different processors, we employ a temporary array *tmp* to store the suffix from the subsequent processor. This enables us to update the inducing successors of the current processor (lines 9 to 17).

By utilizing the inducing successor information from the next processor, each processor can independently update its local portion of inducing successors in parallel (lines 18 to 22). Finally, in line 23, we update the inducing successor information in *ShaIS* for the suffixes belonging to the current equal-modulus suffix set.

Through the *TunnelTrans* function, we efficiently identify inducing successors by constructing equal-modulus suffix sets, performing parallel transfers of suffixes, updating local inducing successor information, and ultimately updating the inducing successor information in *ShaIS* for the current equal-modulus suffix set.

In Alg. 7, the *Refine* function is introduced to ensure that each processor possesses the correct inducing successors. There are three issues that need to be addressed in this function.

The first issue arises when dealing with two suffixes, s_x and s_y, which belong to different unique inducing sets. Although their *D*-prefixes can be used to differentiate them from other suffixes within the same unique inducing set, s_x and s_y share the same *D*-prefix. Consequently, if both suffixes are assigned to the same processor, they will be incorrectly identified as equivalent suffixes instead of distinct ones. To resolve this problem, we present a solution that entails the generation and sorting of the initial

Algorithm 6: Building Shared Inducing Successors

```
1  Function TunnelTrans(ShaIS,ShaGFlag)
     /* building connection between the suffix in an equivalent
        group and its inducing successor.                        */
2    Build Modulus-Equal Suffix Set List MESSL based on ShaIS
3    for (ms in MESSL) do
4      Divide ms into p even blocks
5      forall (i in 0..p-1) do
6        for all the elements in the current block whose flag is True,
           transfer its value to all of its predecessors whose flag is False
7        for all the elements in the current block whose flag is False
           and has not been updated, assign its value with -1
8      end
9      tmp[p − 1] ← the first element of block p − 1
10     for (i from p-2 to 0) do
11       if (the first element fe of block (i+1) is not -1) then
12         tmp[i]=fe
13       end
14       else
15         tmp[i]=tmp[i+1]
16       end
17     end
18     forall (i in 0..p-1) do
19       if (block i has element whose flag is False and its value is -1)
           then
20         Let its value be tmp[i]
21       end
22     end
23     Update ShaIS based on the ms value of each block
24   end
25   return ShaIS
26 end
```

inducing set for every equivalent group. By examining whether only s_x or s_y is present in the inducing set, we can determine whether they should be marked as distinguishable suffixes. To incorporate this update, we call the *Remark* function after constructing the *ShaGFlag* arrays in Alg. 5 (line 2). Placing the *Remark* subfunction within Alg. 7 helps to illustrate the different refining methods clearly.

The second issue pertains to the usage of unique inducing sets instead of inducing sets to identify the inducing successors. Unique inducing sets assume that for two modulus-equal suffixes *Suf(a)* and *Suf(b)*, where $a < b$, if they are equivalent suffixes, then *Suf(b)* and *Suf*$(2\times b - a)$ are also equivalent suffixes. To validate this assumption, we only need to individually examine the successor suffixes (lines 3 to 12 in Alg. 7). Specifically, line 8 states that if *Suf(a)* is a predecessor of *Suf(b)* and they are in one equivalent group, but *Suf(b)* and *Suf*$(2\times b - a)$ have different *D-prefix*, or they are not in an equivalent group, then if *ShaIS[a]* $> b$, we know it is not correct, and we should let *ShaIS[a]* $= b$ because *Suf(b)* can distinguish itself from others in its inducing set. Similarly, line 9 indicates that if *ShaIS[b]* $> 2\times b - a$, we should let *ShaIS[b]* $= 2\times b - a$ because *Suf*$(2\times b - a)$ can distinguish itself from others in its inducing set. These updates serve to rectify the inducing successors if our initial assumptions were incorrect.

By incorporating these updates, we can ensure that the inducing successors are accurately determined, even if the unique inducing sets do not include duplicate suffixes compared with the corresponding inducing sets.

The third issue relates to the fact that different processors may have different inducing successors rather than sharing the same ones. Currently, the construction of the *ShaIS* array is done conservatively, marking a suffix's shared flag as *True* as long as any processor identifies it as such. However, it is possible for different processors to have different inducing successors if one suffix's successor is marked as *True* on one processor but *False* on another. To address this, we propose updating the value of *ParIS[t]* when the local flag *ParGFlag[i][j]* is *False*, but the corresponding suffix *ParSuf[i][j]* $= t$ and the shared flag *ShaFlag[t]* is *True* (lines 13 to 20). This update allows each processor to maintain its own inducing successors.

In lines 14 to 20, we iterate through all the suffixes simultaneously on each processor. This iterative process handles each suffix one by one. In lines 15 to 16, we initially assign the inducing successor from *ShaIS* to *ParIS*. The variable t keeps track of the index of the current suffix.

In lines 17 to 19, we check if the current suffix is marked as an equivalent suffix (*False*) in *ParGFlag* but as a distinguishable suffix (*True*) in *ShaGFlag*. If this condition is met, we update the value of *ParIS[i][j]* using the inducing successor indicated by the next successor *ShaIS*$[t + D]$.

Once *ParIS* is assigned from *ShaIS*, we can further optimize the inducing successors using the elements in *ParIS*. For any two equivalent suffixes *ParSuf[i][j]* and *ParSuf* $[i'][j']$, if *ParSuf*$[i'][j']$ is the first successor of *ParSuf[i][j]*, then *ParIS[i][j]* can directly use the suffix in *ParIS*$[i'][j']$ as its inducing successor.

Suffixes with the same long common prefixes can form a long tunnel. To efficiently transfer the inducing successor in parallel, we can use a method similar to *TunnelTrans* (line 21). This method allows for efficient parallel transfer of the inducing successor in cases where long common prefixes exist among the suffixes.

By implementing these updates, we ensure that each processor can maintain its own set of inducing successors, thereby improving the overall efficiency of the sorting algorithm in the subsequent steps.

Algorithm 7: Inducing Successor Refining

```
1  Function Refine(ShaIS, ParEG)
2    forall (i in 0..p-1) do
3      Let NumEqG ← Number of equivalent groups on processor i based
         on ParEG[i]
4      for (j in 0..NumEqG-1) do
5        Let g be the set of suffixes in ParEG[i][j] group
6        forall (Suf(a), Suf(b) ∈ g and Suf(b) is the closest successor of
           Suf(a)) do
7          if (the D-prefix of Suf(b) and Suf(2 × b-a) are different)
             then
8            ShaIS[a]=min(b,ShaIS[a])
9            ShaIS[b]=min(2×b-a,ShaIS[b])
10         end
11       end
12     end
13     Num_i ← total number of suffixes in ParSuf on processor i;
         ParIS[i]=-1
14     for j in 0..Num_i -1 do
15       t=ParSuf[i][j]
16       ParIS[i][j]=ShaIS[t]
17       if ((ParGFlag[i][j]==False) && (ShaGFlag[t]==True) ) then
18         ParIS[i][j] = ShaIS[t+D]
19       end
20     end
21     Update ParIS from tail to head using the method as in TunnelTrans
         for all equivalent suffixes
22   end
23   return ParIS
24 end
25 Function Remark(ParEG,UniIndSA,UniIndEG)
26   forall (i in 0..p-1) do
27     Building and sorting IndSet_i based on ParEG_i and UniIndEG_i
28     forall (s_x ∈ IndSet_i) do
29       if (s_x is a distinguishable suffix) then
30         Remark ShaGFlag(s_x) as True
31       end
32     end
33   end
34 end
```

4.3.4. Parallel sorting equivalent groups

In this section, we will demonstrate how we can leverage inducing successors to efficiently induce the order of suffixes in equivalent groups simultaneously using Alg. 8.

The *E_PIS* function will produce the partial suffix array for the suffixes belonging to the equivalent groups *ParEG*. The equivalent groups will be processed in parallel by each processor (lines 2 to 13). Firstly, the processor will retrieve the total number of equivalent groups (line 3). Then, for each equivalent group, we will sort its suffixes based on their inducing successors (lines 5 to 8).

By leveraging the inducing successors of each equivalent group, the sorting process becomes highly efficient and straightforward. In line 5, we calculate the z value for the inducing sets that contain the corresponding inducing successors. A higher z value indicates a longer common prefix for the suffix.

To effectively sort the equivalent group in line 6, we first identify such a suffix based on the given inducing successor: (1) It must have the same z value as the inducing successor. (2) It represents a group of suffixes that share the same equivalent group as the inducing successor in the $z-1$ unique inducing set. We refer to such a suffix as a "group suffix" since it signifies a group of suffixes that differ from the inducing successor.

We can sort the inducing successors and their group suffixes just based on their *D-Prefix* (line 7). Consequently, in line 8, we build the partial suffix array for suffixes in the same group. If the suffixes form different groups, then we need to determine the order of different groups. We proceed to sort them recursively in lines 9 to 11. For this, we select one suffix from each group, add them to the set of the current processor, and perform the sorting process again in lines 14–16.

When all independent subgroups have been sorted but the order between different subgroups is unknown, we utilize the suffixes selected from each independent subgroup to call *PIS* recursively (lines 14–16). This step ensures that the suffix order between different subgroups is determined.

Once the suffix order between different subgroups is known, we merge them and construct the final suffix array in line 17. Finally, in line 18, we return the completed suffix array.

These steps collectively enable a highly efficient and streamlined sorting process, optimizing the organization of suffixes based on inducing successors of suffixes with long common prefixes.

Algorithm 8: Parallel Sorting Equivalent Groups

```
1  Function E_PIS(ParEG, ParIS)
2      forall (i in 0..p-1) do
3          Let NumEqG ← Number of equivalent groups on processor i based
             on ParEG[i]
4          for (j in 0..NumEqG-1) do
5              Calculate the z value (the rank of an inducing set) of each
                 suffix's inducing successor
6              Identify the group suffix of each inducing successor with the
                 same z value
7              Sort all inducing successors and group suffixes based on their
                 D-Prefix
8              Build partial suffix array ESA[g]_{i,j} for suffixes in each group g
9              if (there are multiple groups) then
10                 Choose any one suffix from each group and add it to
                     GSetList[i]
11             end
12         end
13     end
14     if (GSetList is not empty) then
15         GSA=PIS(GSetList)
16     end
17     Build RSA based on ESA and GSA
18     return RSA
19 end
```

5. Complexity analysis

In this section, we evaluate the performance of our parallel algorithm using the widely recognized Parallel Random Access Machine (PRAM) model [20]. The proposed algorithm has a time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$ and a space complexity of $\mathcal{O}(n)$. We will prove that each step of the algorithm can be executed in $\mathcal{O}\left(\frac{n}{p}\right)$ time, utilizing at most $\mathcal{O}(n)$ of working space.

Lemma 1. *All substeps of Algorithm 1, excluding the PIS function, can be performed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.*

Proof. Regarding substep 1.1 of step 1, dividing the n suffixes of the given string S into p parts, each with $\mathcal{O}\left(\frac{n}{p}\right)$ elements, can be done straightforwardly in $\mathcal{O}\left(\frac{n}{p}\right)$ time. The space required for the resulting p D-strings will be $\mathcal{O}(p \times D \times \frac{n}{p}) = \mathcal{O}(D \times n) = \mathcal{O}(n)$.

For substep 2.1, selecting $(p-1)$ splitters for each processor based on its returned partial suffix array and grouping them (together with the largest suffix on each processor) into different subsets can also be accomplished in $\mathcal{O}\left(\frac{n}{p}\right)$ time. Here, it is assumed that $p^3 < n$. When (p^2-p) new suffixes are added to each subset, each subset will contain $\mathcal{O}\left(\frac{n}{p}\right) + \mathcal{O}(p^2-p) \leq \mathcal{O}\left(\frac{n}{p}\right) + \mathcal{O}\left(\frac{n}{p}\right) = \mathcal{O}\left(\frac{n}{p}\right)$ elements, resulting in a total working space of $\mathcal{O}(p \times \frac{n}{p}) = \mathcal{O}(n)$.

The number of elements between the two closest splitters cannot be larger than $\mathcal{O}\left(\frac{n}{p^2}\right)$. Therefore, when p intervals of elements divided by the same splitters are combined into one subset, its size cannot be larger than $\mathcal{O}\left(\frac{n}{p}\right)$. At the same time, the elements of each subset will be no larger than $\mathcal{O}\left(\frac{n}{p}\right)$. Based on this conclusion, it is possible to construct p ordered subsets according to the p^2 selected suffixes in substep 3.1, in $\mathcal{O}\left(\frac{n}{p}\right)$ time and with $\mathcal{O}(n)$ working space.

Therefore, the conclusion holds. □

Therefore, we can conclude that the *Tunnel* algorithm has a time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$ and a space complexity of $\mathcal{O}(n)$ when executed on p processors. This holds true provided that the parallel-inducing function *PIS* can generate the partial suffix array for each subset of size $\mathcal{O}\left(\frac{n}{p}\right)$ in $\mathcal{O}\left(\frac{n}{p}\right)$ time while using $\mathcal{O}(n)$ working space.

We will analyze the time and space complexity of the *PIS* function as follows.

Corollary 1.1. *Consider a string S of length n, where its suffixes are divided into p subsets, each with a size not exceeding $\mathcal{O}\left(\frac{n}{p}\right)$. Then, the D-prefixes of each subset can be sorted in $\mathcal{O}\left(\frac{n}{p}\right)$ time using p processors and with a working space of $\mathcal{O}(n)$.*

Proof. To sort the D-prefixes of the p suffix subsets of string S with length n, we can first build D-strings by concatenating the D-substrings corresponding to the suffixes in each subset. This can be done in parallel and will take $\mathcal{O}\left(\frac{n}{p}\right)$ time with $\mathcal{O}(n)$ space. Then, we can apply the optimal sequential suffix array algorithm *SeqOptSA* to compute the corresponding extended suffix arrays in $\mathcal{O}\left(\frac{n}{p}\right)$ time. The extended suffix arrays will contain extra indices, which can be removed in $\mathcal{O}\left(\frac{n}{p}\right)$ time. Thus, the D-prefixes of the given suffix subsets can be sorted in $\mathcal{O}\left(\frac{n}{p}\right)$ time and with $\mathcal{O}(n)$ space. □

Corollary 1.2. *All equivalent groups can be generated in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space in Alg. 3.*

Proof. The information on the equivalent groups can be stored in an equivalent group array *EquGrp[p][$\frac{n}{2}$]* with $\mathcal{O}(\frac{n}{2}) = \mathcal{O}(n)$ space, as the suffixes can be grouped into a maximum of $\frac{n}{2}$ groups. Using the returned partial suffix array, each processor can compare the *D*-prefix of a suffix with its neighbor to determine if they are the same. The total time required for these comparison operations by any processor is $\mathcal{O}(D \times \frac{n}{p}) = \mathcal{O}(\frac{n}{p})$. Building the reverse one-to-one mapping based on *EquGrp* will also need at most $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}\left(\frac{n}{p}\right)$ space for each processor in parallel. Hence, generating all equivalent groups can be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and using $\mathcal{O}(n)$ space. □

Lemma 2. *Alg. 3 can be executed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.*

Proof. Corollaries 1.1 and 1.2 demonstrate that sorting *D*-prefixes and generating all equivalent groups can be performed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space. Therefore, Alg. 3 can also be executed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space. □

Lemma 3. *Alg. 4 generates p unique inducing subsets $UniIndSet_0$,..., $UniIndSet_{p-1}$, with each set containing no more than $\mathcal{O}\left(\frac{n}{p}\right)$ suffixes in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.*

Proof. Generating all unique inducing subsets is straightforward, but ensuring that the size of each set assigned to a processor is no more than $\mathcal{O}\left(\frac{n}{p}\right)$ and that the work is done in $\mathcal{O}\left(\frac{n}{p}\right)$ time, even with overlap between sets generated from different equivalent groups, requires further consideration.

We can divide the task into two parts to analyze the time complexity and space requirements.

In the first part, we consider the case where the maximum rank of the unique inducing sets is no more than $C \times p$. In this scenario, the total number of suffixes in the unique inducing sets of each processor cannot exceed $\mathcal{O}\left(\frac{n}{p}\right)$. This is because, for any unique inducing set with rank z, all the suffixes generated from the equivalent group within one processor will be no more than $\mathcal{O}(C \times \frac{n}{p}) + \mathcal{O}((C \times p - C) \times \frac{n}{p^2}) = \mathcal{O}(\frac{n}{p})$. The calculation can be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and requires $\mathcal{O}(n)$ space.

In the second part, we consider the worst-case scenario where the remaining unique inducing sets ($z \geq C \times p$) contain $\mathcal{O}(n)$ suffixes. The unique inducing sets with the same z on each processor will have a constant number of suffixes. Calculating the bound array for all unique inducing set series within a single processor requires traversing all the elements of the equivalent groups in that processor. This operation can be completed in $\mathcal{O}(1)$ time because each processor will have a constant number of suffixes. The space for each processor will also be $\mathcal{O}(1)$ because at most D such an array will be needed, and each array has two elements (lower bound and upper bound).

During the alignment solution calculation step, We only need to align the unique inducing set series whose total number of duplicated suffixes is large than $\frac{n}{p}$ (much more than p). The method is straightforward. If two unique inducing set series will have overlapping suffixes starting from indices z_1 and z_2, we just align the two series from z_1 and z_2. The aligned unique inducing sets will be merged together and assigned to the same processor. This will make sure the total number of suffixes in different unique inducing sets cannot be larger than $\mathcal{O}(n)$. Once alignment will increase the size of a unique inducing set at most D times. So, aligning unique inducing sets from p processors will cause the size of the merged unique inducing set to be at most $\mathcal{O}(p)$. The total number of unique suffixes cannot be large than n. So each processor will have at most $\mathcal{O}(\frac{n}{p})$ unique suffixes. The total number of duplicated suffixes across different processor cannon be large than $\frac{n}{p}$. So each processor will have at most $\mathcal{O}(\frac{n}{p})$ duplicated suffixes. This can make sure the total number of suffixes assigned to each processor will be no more than $\mathcal{O}\left(\frac{n}{p}\right)$. Therefore, the second part can also be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.

Therefore, the conclusion holds for both parts combined. □

Lemma 4. *The D-prefix substrings of subsets UIndSet0, ..., UindSetp − 1 can be sorted in $\mathcal{O}\left(\frac{n}{p}\right)$ time on p processors with $\mathcal{O}(n)$ space.*

Proof. Based on Lemma 3, *UindSet*0, ...,*UindSetp* − 1 are p suffix subsets, and each of them has at most $\mathcal{O}\left(\frac{n}{p}\right)$ suffixes. Based on Lemma 2, they can be sorted in $\mathcal{O}\left(\frac{n}{p}\right)$ time on p processors with $\mathcal{O}(n)$ space. □

Lemma 5. *For Alg. 5, the inducing successor array ParIS can be generated in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.*

Proof. Since each processor handles a subset of suffixes, the space required for *ParIS* on each processor is no larger than $\mathcal{O}\left(\frac{n}{p}\right)$. Therefore, $\mathcal{O}(n)$ space is sufficient. All the arrays, such as *ParGFlag* and *ShaGFlag*, also need $\mathcal{O}(n)$ space. The assignment to such arrays is straightforward and can be done in parallel in $\mathcal{O}\left(\frac{n}{p}\right)$ time.

For the function *TunnelTrans* in Alg. 6, the process involves passing the closest *True* flag suffix to the current suffix within each equivalent group. We optimize the passing path by dividing long paths into multiple parallel subpaths, which allows parallel processing. Each processor performs suffix passing on different subpaths, ensuring efficient utilization of resources.

During the process, we use temporary memory spaces to transfer indices across different processors. Since the suffixes assigned to each processor are at most $\mathcal{O}\left(\frac{n}{p}\right)$, the first scan procedure in *TunnelTrans* can be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time for all processors. The subsequent passing of values through temporary memory spaces can be done sequentially, requiring at most $\mathcal{O}(p)$ time. Finally, during the last scan, each processor assigns the suffixes with the values stored in the temporary memory space if they point to that memory space. This third substep requires at most $\mathcal{O}\left(\frac{n}{p}\right)$ time.

For the *Refine* function in Alg. 7, the first update *Remark* subfunction will build *IndSet* whose space is no more than *ParEG* and *UniIndSet*. So, $\mathcal{O}(n)$ space is enough. Sorting it in parallel will need $\mathcal{O}\left(\frac{n}{p}\right)$ time since the suffixes on each processor will be no more than $\mathcal{O}\left(\frac{n}{p}\right)$. The remarking procedure can also be done in parallel and each processor only need to check $\mathcal{O}\left(\frac{n}{p}\right)$ suffixes. So, *Remark* subfunction can be done in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space.

The second update will check every suffix in each equivalent group, which takes no more than $\mathcal{O}\left(\frac{n}{p}\right)$ time. The third update involves checking the suffixes of different unique inducing sets assigned to each processor. Since the total number of suffixes handled by one processor is at most $\mathcal{O}\left(\frac{n}{p}\right)$, the time complexity for each processor is also $\mathcal{O}\left(\frac{n}{p}\right)$. Therefore, the total time complexity to locate all inducing successors is $\mathcal{O}\left(\frac{n}{p}\right)$.

In conclusion, the inducing successor array *ParIS* can be generated in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space. □

Lemma 6. *For Alg. 8, the process of inducing the order of suffixes in all equivalent groups based on ParIS can be completed in* $\mathcal{O}\left(\frac{n}{p}\right)$ *time and* $\mathcal{O}(n)$ *space.*

Proof. The algorithm efficiently performs the operations required to calculate the rank z value of each inducing set, group the suffixes into equivalent groups, sort the suffixes within subgroups and determine the order of subgroups.

For each processor, calculating the rank z value of each inducing set can be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time because each processor handles at most $\mathcal{O}\left(\frac{n}{p}\right)$ inducing successors. Grouping the suffixes into equivalent groups based on the z value of their inducing successors also takes at most $\mathcal{O}\left(\frac{n}{p}\right)$ time. Sorting the suffixes within different subgroups independently requires $\mathcal{O}\left(\frac{n}{p}\right)$ time since each subgroup contains only a fraction of the total number of suffixes in the equivalent groups. For the recursive procedure, the size of the input will be at most half of the original input because we just select one suffix from each subgroup and each group has at least two suffixes. Merging two partial suffix arrays *ESA* and *GSA* together will need at most $\mathcal{O}\left(\frac{n}{p}\right)$ time. Based on the *Master Theorem* [21], the total time complexity is $\mathcal{O}\left(\frac{n}{p}\right)$.

All of these operations can be performed using $\mathcal{O}\left(\frac{n}{p}\right)$ space on each processor. Since each processor can handle its equivalent groups independently in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}\left(\frac{n}{p}\right)$ space, the overall execution time for all processors is no more than $\mathcal{O}\left(\frac{n}{p}\right)$, and the total space required is no more than $\mathcal{O}(n)$. Therefore, the conclusion holds. □

Theorem 7. *The execution of Alg. 2 can be completed in* $\mathcal{O}\left(\frac{n}{p}\right)$ *time on p processors while using* $\mathcal{O}(n)$ *space.*

Proof. The proof of Theorem 7 is a direct result of Lemmas 2, 3, 4, 5, and 6. All components of Alg. 2 can be executed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space, as proven in each respective lemma. Hence, by combining these components, we conclude that Alg. 2 can be completed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space. □

Theorem 8. *The suffix array of a string S of length n can be generated in parallel using Alg. 1 in* $\mathcal{O}\left(\frac{n}{p}\right)$ *time on p processors with* $\mathcal{O}(n)$ *space.*

Proof. The proof follows from the analysis in Lemma 1 and Theorem 7, which demonstrate that each step in the algorithm can be performed in $\mathcal{O}\left(\frac{n}{p}\right)$ time and $\mathcal{O}(n)$ space. The conclusion follows by aggregating these results. □

6. Practical performance optimization

In this section, we discuss some techniques that can improve the practical performance of the proposed *Tunnel* algorithm.

6.1. Optimized sampling technique

In substep 1.1 of Algorithm 1, we can utilize a specific sampling method to eliminate the need to generate unique inducing sets.

Using the constant D, we can assign suffixes to p processors using a special cyclic distribution called *CYCLIC(D)*. This distribution follows a specific pattern to allocate the suffixes among the processors.

First, we assign the suffixes *Suf(0), Suf(1), ..., Suf(D-1)* to processor 0. Then, the next D suffixes, *Suf(D), Suf(D+1), ...,* *Suf*$(2\times D-1)$, are assigned to processor 1. This pattern continues, assigning the subsequent D suffixes, *Suf*$(D\times(p-1))$, *Suf*$(D\times(p-1)+1)$, ..., *Suf*$(D\times p-1)$, to processor $(p-1)$. Finally, the distribution wraps back to processor 0, and this assignment process repeats until all suffixes have been allocated to the processors.

For any suffix set ss on processor i, where $0 \le i < p$, if ss is an equivalent group, then any of its inducing sets can be found in full on some other processors. This eliminates the need to generate unique inducing sets and assign them to different processors, as they are already available. So, in Alg. 2, we do not need to call the subfunction *BltIT* in line and *D_PIS* in line 4. The *LocatIS* function in line 5 can directly use the results from line 2 to calculate the *ParIS*. Obviously, this can save some execution time.

6.2. Splitter optimization

In step 2 of Alg. 1, we can adopt a more efficient approach to sort the splitters and arrange the suffixes accordingly.

One method to improve the efficiency is to directly sort the p^2 splitters gathered from various processors. Once all the splitters are sorted, each processor can sort the intervals that contain splitters from other processors, while skipping the intervals that do not include new splitters. This results in a reduced number of suffixes being sorted, saving valuable processing time.

Another optimization strategy is to decrease the total number of splitters used. While selecting p^2 splitters provides optimal load balancing, using a smaller number of splitters in practical scenarios can result in improved performance without sacrificing load balancing. This is because reducing the number of splitters also reduces the number of suffixes that need to be sorted.

7. Related work

The field of suffix array construction has seen numerous advancements since its inception in 1990 by Manber and Myers [1]. The use of "inducing" – leveraging the order of certain suffixes to induce the order of others – has proven to be a crucial technique in the sorting of suffixes. Although prefix-doubling [22] adopts the inducing technique, it cannot reduce the problem size step by step. This is why it cannot achieve $\mathcal{O}(n)$ time complexity. Subsequent works [3–5] have successfully tackled this issue by constructing a reduced problem and employing the inducing technique to sort the suffixes in a recursive manner.

All existing parallel suffix array construction algorithms were trying to parallelize one or combined sequential algorithms. Futamura et al. [12] gave the early effort to parallel the prefix-doubling method. Larsson et al. [23] implemented optimized methods based on the previous prefix-doubling technology and

improved its performance in parallel. Osipov et al. [14] implemented the prefix-doubling algorithm on GPUs. Flick and Aluru [24]'s parallel MPI-based implementation of the prefix-doubling method can achieve very high practical performance on human genome datasets. Kulla et al. [25] parallelized the sequential DC3 method, which regularly samples the string to build a smaller $\frac{2}{3}n$ problem. Deo et al. [15] further implement the DC3 method on GPUs. Shun [26]'s parallel skew (DC3) algorithm could achieve good performance on shared-memory multicore computers. Wang et al. [27] implemented a hybrid prefix-doubling and DC3 method on GPUs to improve the existing GPU methods significantly. Lao et al. [18,19] employed pipeline technology to parallelize their previous sequential linear algorithms [10,28] on multicore computers.

While these parallel algorithms have greatly improved their practical performance compared to their sequential counterparts, they still fall short of achieving $\mathcal{O}\left(\frac{n}{p}\right)$ time complexity. The conventional sequential algorithm framework proves to be a hindrance to attaining scalable performance in parallel methods. Our proposed framework and parallel-inducing method break this barrier and achieve $\mathcal{O}\left(\frac{n}{p}\right)$ time complexity.

8. Conclusion

This paper introduces the concept of *D*-strings and presents a novel parallel-inducing sort technology along with a three-phase workflow for efficiently sorting large strings. The proposed algorithm, named *Tunnel*, represents a significant advancement in the field of parallel suffix array construction. Notably, it achieves a remarkable time complexity of $\mathcal{O}\left(\frac{n}{p}\right)$, where n is the size of the input string and p denotes the number of parallel processors. Importantly, *Tunnel* outperforms existing parallel algorithms by being the first to achieve such time complexity, assuming $p^3 < n$.

The key strength of the *Tunnel* algorithm lies in its efficient handling of large read-only strings constructed from an integer alphabet. By leveraging the order of suffixes in their inducing set, the algorithm parallelizes the sorting of suffixes with long common prefixes. It tackles this complex problem through a meticulous step-by-step process, employing three well-defined phases. Moreover, it builds upon an optimal sequential suffix array construction algorithm as an independent execution unit to calculate the order of all *D*-prefixes, effectively separating suffixes with long common prefixes from those with short unique prefixes.

The simplicity and $\mathcal{O}\left(\frac{n}{p}\right)$ time complexity of the proposed *Tunnel* algorithm make it highly promising for efficiently handling massive strings while ensuring scalable performance. Notably, it achieves optimality in terms of asymptotic time complexity. As part of future work, our goal is to further reduce the total working space from $\mathcal{O}(n)$, thereby enhancing the algorithm's efficiency and applicability in even larger string contexts.

CRediT authorship contribution statement

Zhihui Du: Algorithm development, Provided proof of its effectiveness. **Sen Zhang:** Algorithm analysis, Provided example design. **David A. Bader:** Design of the parallel framework.

Declaration of competing interest

To the best of our knowledge, we do not have conflict of interest.

Data availability

Data will be made available on request.

Acknowledgment

This research was funded in part by National Science Foundation (NSF) grant number CCF-2109988.

References

[1] Udi Manber, Gene Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[2] Edward M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.
[3] Pang Ko, Srinivas Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.
[4] Juha Kärkkäinen, Peter Sanders, Simple linear work suffix array construction, in: International Colloquium on Automata, Languages, and Programming, Springer, 2003, pp. 943–955.
[5] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, Kunsoo Park, Linear-time construction of suffix arrays, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2003, pp. 186–199.
[6] Stefan Burkhardt, Juha Kärkkäinen, Fast lightweight suffix array construction and checking, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2003, pp. 55–69.
[7] Giovanni Manzini, Paolo Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (1) (2004) 33–50.
[8] Michael A. Maniscalco, Simon J. Puglisi, Faster lightweight suffix array construction, in: Proc. of International Workshop on Combinatorial Algorithms, IWOCA, 2006, pp. 16–29.
[9] Juha Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theoret. Comput. Sci. 387 (3) (2007) 249–257.
[10] Ge Nong, Practical linear-time O(1)-workspace suffix sorting for constant alphabets, ACM Trans. Inf. Syst. (TOIS) 31 (3) (2013) 1–15.
[11] Zhize Li, Jian Li, Hongwei Huo, Optimal in-place suffix sorting, Inform. and Comput. 285 (2022) 104818.
[12] Natsuhiko Futamura, Srinivas Aluru, Stefan Kurtz, Parallel suffix sorting, Electrical Engineering and Computer Science - All Scholarship. 64. 64 (2001).
[13] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, Kanat Tangwongsan, Brief announcement: the problem based benchmark suite, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2012, pp. 68–70.
[14] Vitaly Osipov, Parallel suffix array construction for shared memory architectures, in: International Symposium on String Processing and Information Retrieval, Springer, 2012, pp. 379–384.
[15] Mrinal Deo, Sean Keely, Parallel suffix array and least common prefix for the GPU, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013, pp. 197–206.
[16] Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[17] Robert Homann, David Fleer, Robert Giegerich, Marc Rehmsmeier, mkESA: enhanced suffix array construction tool, Bioinformatics 25 (8) (2009) 1084–1085.
[18] Bin Lao, Ge Nong, Wai Hong Chan, Jing Yi Xie, Fast in-place suffix sorting on a multicore computer, IEEE Trans. Comput. 67 (12) (2018) 1737–1749.
[19] Bin Lao, Ge Nong, Wai Hong Chan, Yi Pan, Fast induced sorting suffixes on a multicore machine, J. Supercomput. 74 (7) (2018) 3468–3485.
[20] Joseph JáJá, An Introduction to Parallel Algorithms, Vol. 10, Addison-Wesley, Reading, MA, 1992, 133889.
[21] Jon Louis Bentley, Dorothea Haken, James B. Saxe, A general method for solving divide-and-conquer recurrences, ACM SIGACT News 12 (3) (1980) 36–44.
[22] Simon J. Puglisi, William F. Smyth, Andrew H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2) (2007) 4–es.
[23] N. Jesper Larsson, Kunihiko Sadakane, Faster suffix sorting, Theoret. Comput. Sci. 387 (3) (2007) 258–272.
[24] Patrick Flick, 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, 2015, pp. 1–10.
[25] Fabian Kulla, Peter Sanders, Scalable parallel suffix array construction, Parallel Comput. 33 (9) (2007) 605–612.
[26] Julian Shun, Fast parallel computation of longest common prefixes, in: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2014, pp. 387–398.

[27] Leyuan Wang, Sean Baxter, John D. Owens, Fast parallel skew and prefix-doubling suffix array construction on the GPU, Concurr. Comput.: Pract. Exper. 28 (12) (2016) 3466–3484.

[28] Ge Nong, Sen Zhang, Wai Hong Chan, Two efficient algorithms for linear time suffix array construction, IEEE Trans. Comput. 60 (10) (2010) 1471–1484.

Zhihui Du received the BE degree in 1992 in the computer department from Tianjin University. He received the MS and Ph.D. degrees in computer science, respectively, in 1995 and 1998, from Peking University. From 1998 to 2000, he worked at Tsinghua University as a postdoctoral. From 2001 to 2019, he worked at Tsinghua University as an associate professor in the Department of Computer Science and Technology. Currently, he works at the Department of Data Science of New Jersey Institute of Technology as a principal senior researcher. His research areas include high-performance computing, large data analysis and parallel algorithm.

Sen Zhang is a professor of Computer Science in the Department of Mathematics, Computer Science and Statistics at State University of New York College at Oneonta. He received the MS degree in 1995 from South China University of Science and Technology and Ph.D. degree in 2005 from NJIT, both in computer science.

David A. Bader is a Distinguished Professor and founder of the Department of Data Science in the Ying Wu College of Computing and Director of the Institute for Data Science at New Jersey Institute of Technology. Prior to this, he served as founding Professor and Chair of the School of Computational Science and Engineering, College of Computing, at Georgia Institute of Technology. He is a Fellow of the IEEE, ACM, AAAS, and SIAM; a recipient of the IEEE Sidney Fernbach Award; and the 2022 Innovation Hall of Fame inductee of the University of Maryland's A. James Clark School of Engineering.