Parallel Suffix Sorting for Large String Analytics

Zhihui Du[1], Sen Zhang[2], and David A. Bader[1]

[1] New Jersey Institute of Technology,Nework, New Jersey, US
{zhihui.du,bader}@njit.edu
[2] State University of New York, College at Oneonta, New York, US
zhangs@oneonta.edu

Abstract. The suffix array is a fundamental data structure to support string analysis efficiently. It took about 26 years for the sequential suffix array construction algorithm to achieve $\mathcal{O}(n)$ time complexity and in-place sorting. In this paper, we develop the $DLPI$ (D Limited Parallel Induce) algorithm, the first $\mathcal{O}(\frac{n}{p})$ time parallel suffix array construction algorithm. The basic idea of DLPI includes two aspects: dividing the $\mathcal{O}(n)$ size problem into p reduced sub-problems with size $\mathcal{O}(\frac{n}{p})$ so we can handle them on p processors in parallel; developing an efficient parallel induce sorting method to achieve correct order for all the reduced sub-problems. The complete algorithm description is given to show the implementation method of the proposed idea. The time and space complexity analysis and proof are also given to show the correctness and efficiency of the proposed algorithm. The proposed $DLPI$ algorithm can handle large strings with scalable performance.

Keywords: Suffix Array · String Algorithm · Parallel Sorting · String Analysis · Optimal Algorithm.

1 Introduction

Suffix arrays were initially introduced by Manber and Myers [21] as a space efficient alternative to suffix trees [24, 3, 16]. Suffix arrays can be widely used in string processing, data compression, text indexing, information retrieval and computational biology. Since the volume of string data is increasing constantly, high performance suffix array construction algorithms (SACAs) have been a challenging problem.

Thirteen years after the suffix array was proposed, the first linear time algorithm for suffix sorting over integer alphabets was achieved by three research groups, Ko and Aluru [13, 14], Kärkkäinen and Sanders [10] and Kim *et al.* [12] at almost the same time. They reduced the time complexity of suffix array construction algorithms from original $\mathcal{O}(nlog(n))$ to $\mathcal{O}(n)$. These sequential algorithms are optimal in terms of asymptotic time complexity. Furthermore, many lightweight algorithms [1, 23, 22, 9] with small working space were developed. Especially, Nong *et al.* [25] can achieve $\mathcal{O}(1)$ space complexity for constant alphabets and Li *et al.* [20] can achieve $\mathcal{O}(1)$ in-place sorting for read-only integer alphabets. This also took about thirteen years to reduce the working space complexity from $\mathcal{O}(n)$ to $\mathcal{O}(1)$.

Besides the great advance in sequential SACAs, many parallel SACAs have also been developed. For examples, Futamura *et al.* [5] gave a very early effort to implement a parallel SACA based on the sequential prefix-doubling method. Shun's problem-based benchmark suite (PBBS) [30] leveraged the task-parallel Cilk Plus programming model in its parallel multicore skew algorithm implementation. Osipov [27] and Deo and Keely [2] implemented the parallel Difference Cover 3 [11] or skewed algorithm on GPU. Homann *et al.* [7] introduced the mkESA tool on multithreaded CPUs that could parallelize the sequential induce copy method. Lao *et al.* [18, 17] implemented their parallel recursive algorithm on multicore computers. All the parallel methods can significantly improve the practical performance compared with the corresponding sequential methods. However, none of them can handle very large string on many (p) processors in $\mathcal{O}(\frac{n}{p})$ time.

To achieve scalable performance, we need a parallel SACA with $\mathcal{O}(\frac{n}{p})$ time complexity. The major contributions of this paper are as follows.

- A high level parallel suffix sorting framework is proposed. This framework aims to divide a large string's suffix sorting problem ($T(n,p)$) into many even size reduced sub-problems ($T(\frac{n}{p},1)$) and the large problem can be solved by handling the many reduced sub-problems on p processors in parallel, or $T(n,p) = T(\frac{n}{p},1)$.
- The first parallel suffix array construction algorithm $DLPI$ with $\mathcal{O}(\frac{n}{p})$ time is presented. $DLPI$ is optimal in terms of asymptotic time complexity.

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 $suf(i') < suf(j')$, where $i' = SA[i]$, $j' = SA[j]$ and $suf(k)$ is the suffix $S[k..n-1]$.*

Definition 2. *Read-only integer alphabets: The alphabets Σ is a set of characters 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 $S[i] \in \Sigma$, where $|\Sigma| = \mathcal{O}(n)$. At the same time, the given string S cannot be changed during the procedure of building its suffix array. Since different characters can be encoded as different integers. So, we assume $\forall S[i]$, we have $S[i] \in \{x | 1 \leq x \leq |\Sigma|\}$.*

In this paper, our problem is based on read-only integer alphabets instead of constant alphabets, which have only constant characters, or integer alphabets, whose input strings can be updated during the sorting procedure. The constant or integer alphabets is a particular case of our problem.

The proposed problem is as follows. Given a very large string S built from a read-only integer alphabets Σ with length n and a parallel random access machine (PRAM) with p processors, can we have a parallel algorithm to build the suffix array of S in $\mathcal{O}(\frac{n}{p})$ time?

3 Algorithm Design

Unlike the existing parallel SACAs, we do not try to find parallelism in the framework of sequential SACAs. Instead, we first build a parallel framework that aims to divide the whole problem into many reduced sub-problems. Then we develop a parallel induce method to solve all the reduced sub-problems.

Definition 3. *Order of Suffix Sets: Given two non-empty suffix sets* Set_1 *and* Set_2 *of a string* S*, if* $\forall x \in Set_1, \forall y \in Set_2$*, their lexicographical order meets* $x < y$ *(or* $x > y$*), then we define* $Set_1 < Set_2$ *(or* $Set_1 > Set_2$*).*

In this section, we propose an idea to sort the suffixes of a long string in two steps. First, we construct many (p) suffix subsets to cover all the suffixes. The suffix subsets are ordered, but suffixes in each suffix subset are not sorted. Then, we sort each suffix subset in parallel into its own sub-suffix array and achieve the complete suffix array by combining the different sub-suffix arrays corresponding to different suffix subsets together.

Algorithm 1: $DLPI$ Algorithm

```
1  Function DLPI(String, p)
2      Step (1) Build parallel reduced subproblems
3      1.1 Divide all suffixes of S into p suffix subsets
         SubSet_1, ..., SubSet_p, ∀1 ≤ i ≤ p, |SubSet_i| = O(n/p)
4      1.2 Call Parallel Suffix SubSets Sorting function SA = PSSS(SubSet_1, ..., SubSet_p)
5      1.3 Evenly select (p − 1) splitters from each processor p_i's returned suffix array SA[i]
6      1.4 Add the (p − 1) × p splitters into each subset to get SpSubSet_i, 1 ≤ i ≤ p
7      1.5 Call Parallel Suffix SubSets Sorting function SA = PSSS(SpSubSet_1, ..., SpSubSet_p)
8      1.6 According to the returned SA, divide all suffixes into p ordered subsets that meet
         OSubSet_1 < ... < OSubSet_p
9      Step (2) Sort reduced subproblems in parallel
10     2.1 Call Parallel Suffix SubSets Sorting function SA = PSSS(OSubSet_1, ..., OSubSet_p)
11     2.2 return SA
12 end
```

3.1 Algorithm Framework

In Alg. 1, we present the framework of our parallel suffix array construction algorithm $DLPI$ (D Limited Parallel Induce). This framework transforms a large $T(n,p)$ problem, which means that the problem size is n and the parallel random access machine has p processors, into many parallel $T(\frac{n}{p},1)$ problems, which means that the single problem size is $\frac{n}{p}$ and it can be handled with one processors.

In line 3 of Alg.1, for all the n suffixes of a given string S, we assign them into p subsets evenly. The $PSSS$ function will generate different sub-suffix arrays corresponding to different subsets and we can select $(p-1)$ different splitters [6] to divide each subset evenly (line 5). When we add the $p \times (p-1)$ splitters into the previous subsets (we will use a special array to mark the additional splitters added to different subsets) and call PSSS again, we can take advantage of the ordered $p \times (p-1)$ splitters to organize all the suffixes into p ordered subsets,

$OSubSet_1, ..., OSubSet_p$ (lines 6-8). Here, all suffixes are assigned into p ordered subsets with the same size $\mathcal{O}(\frac{n}{p})$.

The second step is straightforward, and we just call $PSSS$ again to generate the order of suffixes in different subsets and then combine them together as the complete suffix array SA (lines 10-11).

3.2 Parallel Induce Method

In Alg.2, we describe the essential function $PSSS$ that can support parallel induce on all reduced sub-problems. The basic idea of this function is that we first construct p much smaller strings to express the different sub-problems. The suffixes without long repeated prefixes can be sorted easily and the novel parallel induce method is used to achieve the order of suffixes with long repeat prefixes.

Algorithm 2: Parallel Suffix SubSets Sorting Algorithm

```
1  Function PSSS(SubSet_1, ..., SubSet_p)
2      Step (1) Sort suffixes of each subsets and distinguish Fixed and Changeable suffixes
3      Build D limited shrunk strings DS_S_1, ..., DS_S_p according to different subsets
4      forall (i in 1..p) do
5          ESA[i][] = SeqOptSA(DS_S_i)
6          Remove all indices ESA[i][j] that are not in Set_i and get SA[i][] corresponding to SubSet_i
7          var mg=-1
8          for (j in 0..|SubSet_i|-1) do
9              if (Suf(SA[i][j]) and its closest suffix in SA have the same D prefix) then
10                 Flag[i][j]=Changeable
11                 if (Suf(SA[i][j]) is the first Changeable suffix of a new group then
12                     mg++
13                     ChgGrp[i][mg].head=j
14                 end
15                 ChgGrp[i][mg].num++
16             end
17             else
18                 Flag[i][j]=Fixed
19             end
20         end
21     end
22     Step (2) Induce the order of Changeable suffixes in each Changeable suffix group
23     2.1 Build aligned subsets AliSubSet_1, ..., AliSubSet_p for Changeable suffix groups
24     2.2 Generate the new suffix array AliSA of the suffixes just like the previous step (1)
25     2.3 Generate the distinguishable tail suffix array DTA for suffixes in the Changeable suffix groups
26     2.4 Induce the correct order of all Changeable suffixes in SA based on DTA and AliSA
27     return SA
28 end
```

We introduce the first step of $PSSS$ function at first.

Definition 4. *D limited substring and D limited shrunk string: Given a constant D, a string S with length n and one of its suffix subset $SubSet$, if two suffixes $suf(i) \in SubSet$ and $suf(j) \in SubSet$, where $i < j$ and no other suffix sits between i and j in $SubSet$(we will let $j = n$ if no such $suf(j)$ in $SubSet$), then the D limited substring of $suf(i)$ is substring $S[i..j-1]$ if $j - i \leq D$ or $S[i..i+D-1]$ if $j-i > D$. The D limited shrunk string DS of S is the string by concatenating all D limited substrings from $SubSet$ together according to their original order in S.*

Building Reduced Strings The first step of this function is building p much smaller D limited shrunk strings $DS_1, ..., DS_p$ so each processor can handle one

smaller string in parallel (line 3). We use D limited substrings to replace the original suffixes.

We will call the existing optimal sequential SACA $SeqOptSA$ [20] to generate the extended suffix array for the given shrunk string. Since we do not need to compare the suffixes not included in the given subset, we may remove the indices of such suffixes in the extended suffix array and get the exact suffix array SA (lines 5-6). It is a two-dimension array. The first dimension stands for the number of processors and the second dimension stands for the maximum number of suffixes assigned to different processors.

For the suffix whose order can be decided based on its D prefix, its rank in the suffix array is correct. If there are two or more suffixes whose D prefixes are exactly the same, their ranks in SA should be induced based on their complete suffixes instead of their D limited substrings in the shrunk string. We use a two-dimension array $Flag$ to mark the correct rank as Fixed and the rank to be induced as Changeable. At the same time, we use a two-dimension array $ChgGrp$ to manage the clustered Changeable suffixes by their D prefix. $ChgGrp[i][mg]$ keeps the current group of Changeable suffixes on processor i. $ChgGrp[i][mg].head$ is the rank of the first suffix in the corresponding suffix array and $ChgGrp[i][mg].num$ is the total number of suffixes in the current group (lines from 7 to 20).

Based on the $ChgGrp$ data structure, the induce sorting method is as follows. When we know the smallest suffix in the group mg, we just need to switch the rank of the smallest suffix with that of the head suffix, advance $ChgGrp[i][mg].head$ by one, and reduce $ChgGrp[i][mg].num$ by 1. If a suffix can split the suffixes into two ordered subsets, we will put the suffix at the correct position in its SA and split its Changeable suffix group into two smaller groups. In this way, we can induce one suffix at its correct position. When $ChgGrp[i][mg].num$ is one, all suffixes in the Changeable group are correctly sorted. The suffixes in different groups can be induced in parallel.

The major work of the second step is inducing the correct ranks of Changeable suffixes (line 22). The basic idea is building induce chain for all the Changeable suffixes; then identifying the tail suffix that can distinguish the Changeable suffix from other suffixes; inducing the order for each Changeable suffix based on the tail suffixes. It includes four substeps and we will present the detailed descriptions as follows.

Definition 5. *Aligned suffix set: Given a Changeable suffix group CG and a non negative integer k, the set $\{suf(x)|\forall e \in CG, e = suf(y) \wedge x = y + D \times k \wedge x < n\}$ is the k aligned suffix set of CG.*

Building Aligned Suffix Sets In the first substep (line 23) we build p completely new suffix subsets $AliSubSet_1$, ..., $AliSubSet_p$ that are used to induce the correct order of all the Changeable suffixes. Suffixes in an aligned suffix set will be assigned to the same processor so we can get their order based on each processor's suffix array.

For all the Changeable suffix groups, we can generate all of their k aligned suffix set. If $|i-j|\%D = 0$, then $\forall suf(i), suf(j) \in CG$, they may have the same suffixes in different k aligned suffix sets. We will merge two such overlapping k aligned suffix sets as one big set ($k > 0$). Then we evenly distributed these sets into p processors and form p suffix subsets $AliSubSet_1$, ..., $AliSubSet_p$.

Generating SA for Aligned Suffix Sets In the second substep, we may employ the similar method as before (lines from 2 to 21) to generate the suffix array of each aligned suffix subset. Here we use $AliSA$ to express the new suffix array corresponding to the aligned suffixes. $AliFlag$ has the similar meaning as before to mark the Fixed and Changeable suffixes.

Building the Distinguishable Tail Suffix Array In the third substep, we will build an array DTA to store the suffixes that can be used to distinguish one Changeable suffix from other suffixes in the same Changeable group.

Definition 6. *Distinguishable Tail Suffix: For any Changeable suffix $suf(x)$ in a Changeable suffix group $ChgGrp$, its distinguishable tail suffix $suf(DTA(x))$ is the suffix that can distinguish the order of $suf(x)$ from the other Changeable suffixes according to $suf(DTA(x))$'s D limited substring.*

We will transfer the index t of suffix $suf(t)$ whose flag is Fixed to its left suffix $suf(t-D)$ and let $DTA[t-D] = t$ if $suf(t-D)$ exists and it is a Changeable suffix. This procedure will continue to the head of the string along the induce chain of $suf(t)$. The challenge here is that we should do it in parallel. The basic idea is as follows.

We first assign all suffixes (not including the additional splitters if there are such suffixes) to different processors based on the indices of different suffixes evenly and each processor only checks about $\frac{n}{p}$ suffixes. For suffixes assigned to the current processor i, we will cluster them into D classes based on their indies' modulo D values. Each processor will scan every class from its end suffix to its start suffix. The index of the Fixed suffix $suf(f)$ will be passed to its left suffix $suf(f-D)$ one by one until the new Fixed suffix is met. Then the new Fixed suffix will replace the old one and be passed to the left suffix. If one end suffix $suf(e)$ is Changeable, then we use a temporary array $tmp[D][p]$ with $(D \times p)$ elements to pass the value across processors. We let $tmp[e\%D][i-1] = -1$ and $DTA[e] = -(i)$ that means that $suf(e)$'s DTA value $DTA(e)$ is unknown and it will get its value from $tmp[e\%D][i-1]$. All Changeable suffixes of the current processor that cannot get its distinguishable tail suffix from its last suffix will point to the same element $tmp[e\%D][i-1]$. After this, we will scan the temporary array from end to start for different modulo values. For current temporary element $tmp[d][i]$ that is corresponding to the $(i+1)$ processor and d^{th} class, if $suf(r)$ is first suffix of its right processor, $r\%D = d$, and $DTA[r] > 0$, we will let $tmp[d][i] = DTA[r]$. If not, we will let $tmp[d][i] = tmp[d][i+1]$. The temporary array update will start from $tmp[d][p-2]$ and end with $tmp[d][0]$ (for all d in $[0..D-1]$). Finally, each processor will check all of its suffixes that

their DTA value is negative and update them with the corresponding temporary value. In this way, we can pass the distinguishable tail suffix from the end to start in parallel and quickly.

Inducing the Order of Changeable Suffixes In the fourth substep, we have known the distinguishable tail suffix of each Changeable suffix. We can use such information to induce the order of suffixes in each Changeable suffix group. For each Changeable suffix group, first we will use its closest distinguishable tail suffixes to distinguish the corresponding Changeable suffixes from others. Then, we will induce the correct order of all Changeable suffixes based on their distinguishable tail suffixes' indices from small to large. The order of different Changeable suffix groups can be induced in parallel.

4 Complexity Analysis

In this section, we adopt the widely used Parallel Random Access Machine (PRAM) model [8] to analyze our parallel algorithm. The time complexity of the proposed algorithm is $\mathcal{O}(\frac{n}{p})$ and the space complexity is $\mathcal{O}(n)$. We will prove that every step of our algorithm can be done in $\mathcal{O}(\frac{n}{p})$ time and at most $\mathcal{O}(n)$ working space (the space except the input string S and the returned suffix array) is needed to generate the complete suffix array.

The $DLPI$ function gives the framework of our algorithm. For substep 1.1 of step 1, we can assign the suffixes of the given string S with length n into p parts and each has about $\frac{n}{p}$ elements using block or cyclic distribution in $\mathcal{O}(\frac{n}{p})$ time. The p D limited shrunk strings will need $\mathcal{O}(p \times D \times \frac{n}{p}) = \mathcal{O}(D \times n) = \mathcal{O}(n)$ space. For substep 1.2, we will give the time and space complexity of the parallel induce function $PSSS$ later. Selecting $(p-1)$ splitters for each processor based on its returned suffix array and adding them into different subsets are straightforward and can also be done in $\mathcal{O}(\frac{n}{p})$ time. Here we assume $p^3 < n$, when we add $p \times (p-1)$ elements to each subset, each subset will have $\mathcal{O}(\frac{n}{p}) + \mathcal{O}(p \times (p-1)) \leq \mathcal{O}(\frac{n}{p}) + \mathcal{O}(\frac{n}{p}) = \mathcal{O}(\frac{n}{p})$ elements. So, the total working space will also be $\mathcal{O}(p \times \frac{n}{p}) = \mathcal{O}(n)$.

For substep 1.5, just like before, we will discuss the time complexity of $PSSS$ later. From substeps 1.3 to 1.6, we know that the total number of elements between two closest splitters cannot be larger than $\mathcal{O}(\frac{n}{p^2})$. So, when we combine p parts of elements divided by the same splitters together into one subset, its size cannot be larger than $\mathcal{O}(\frac{n}{p})$. At the same time, the elements of each subset will be no larger than $\mathcal{O}(\frac{n}{p})$. Based on this conclusion, it is feasible for us to build p ordered subsets according to the $p \times (p-1)$ splitters.

Hence, we can claim that $DLPI$ function can generate the complete suffix array of a given string S with length n in $\mathcal{O}(\frac{n}{p})$ time on p processors using $\mathcal{O}(n)$ space if the parallel induce function $PSSS$ can return the suffix array for each suffix subset in $\mathcal{O}(\frac{n}{p})$ time on p processors using $\mathcal{O}(n)$ working space.

Theorem 1. *For a string S with length n, if its suffixes are assigned to p given subsets with size $\mathcal{O}(\frac{n}{p})$, then the corresponding D limited substrings of each subset can be sorted in $\mathcal{O}(\frac{n}{p})$ time on p processors with $\mathcal{O}(n)$ space.*

Proof. We can build D limited shrunk strings based on given p suffix subsets of the string S in parallel. The shrunk stings can be done by directly concatenating all the D limited substrings corresponding to the suffixes in each subset directly. This work will take $\mathcal{O}(\frac{n}{p})$ time with $\mathcal{O}(n)$ space. Then, we may employ the existing in-place sequential linear suffix array algorithm $SeqOptSA$ to directly return their corresponding extended suffix arrays in $\mathcal{O}(\frac{n}{p})$ time. The extended suffix arrays will contain more indices than each subset's elements. So, we need to remove the additional indices. This can also be done in at most $\mathcal{O}(\frac{n}{p})$ time. Totally, $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space will be needed to sort the D limited substrings of suffixes in all the given subsets.

Lemma 1. *All suffixes can be marked as Fixed or Changeable suffixes and clustered into groups in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.*

Proof. To mark all suffixes as Fixed or Changeable, a $Flag[1..p][]$ array with $\mathcal{O}(n)$ space will be needed. To store the Changeable group information, at most $\mathcal{O}(\frac{n}{2})$ space for a Changeable suffix array $ChgGrp[1..p][]$ will be needed because the suffixes can be divided into at most $(\frac{n}{2})$ groups. Based on the returned suffix array, each processor can compare any suffix's D prefix with its neighbor to check if they are the same. The different D prefix means the suffix can be marked as Fixed; otherwise, it will be marked as Changeable. The entire character comparison operations for any processor should be $\mathcal{O}(D \times \frac{n}{p}) = \mathcal{O}(\frac{n}{p})$. It is similar to clustering the Changeable suffixes based on their D prefixes and storing the group information into $ChgGrp$. So, the marking and clustering operations can be done in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.

Lemma 2. *The aligned subsets $AliSubSet_1, ..., AliSubSet_p$ that each is no more than $\mathcal{O}(\frac{n}{p})$ elements can be built in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.*

Proof. The total number of aligned suffixes cannot be larger than $\mathcal{O}(n)$. Since we evenly assign aligned suffix sets to different processors, the total number of suffixes assigned to one processor cannot be larger than $\mathcal{O}(\frac{n}{p})$. The total number of suffixes in the Changeable suffix groups cannot be larger than $\mathcal{O}(n)$, and the total number of suffixes in all aligned suffix sets cannot be larger than $\mathcal{O}(n)$ either. So, for the first substep, totally at most $\mathcal{O}(n)$ space will be needed to store all the suffixes. Generate at most $\mathcal{O}(\frac{n}{p})$ suffixes $AliSubSet_1, ..., AliSubSet_p$ for each processor from the Changeable suffix groups is straightforward and it can be done at most in $\mathcal{O}(\frac{n}{p})$ time.

Corollary 1. *The D limited substrings of subsets $AliSubSet_1, ..., AliSubSet_p$ can be sorted in $\mathcal{O}(\frac{n}{p})$ time on p processors with $\mathcal{O}(n)$ space.*

Proof. $AliSubSet_1, ..., AliSubSet_p$ are p suffix subsets and each of them have at most $\mathcal{O}(\frac{n}{p})$ suffixes. Based on theorem 1, we can get the corollary and the third substep can be done in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.

Lemma 3. *The distinguishable tail suffix array DTA can be generated in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.*

Proof. We can allocate the DTA array with size n to cover all suffixes. So, $\mathcal{O}(n)$ space is enough. The basic idea of distinguishable tail suffix generation is passing the closest Fixed suffix to the current Changeable suffix and storing the Fixed suffix's index in DTA. The short passing path will be easy to implement. In order to reduce the passing time for a very long passing path, our implementation method divides the long passing path into multiple parallel subpaths. The suffix passing can be done on different subpaths in parallel. We allocate at most p temporary memory space to transfer the index across different processors. Since all the suffixes assigned to one processor cannot be larger than $\mathcal{O}(\frac{n}{p})$, the first scan procedure can be done in $\mathcal{O}(\frac{n}{p})$ time for all the processors. Then we let one processor pass the value in the temporary memory one by one from end to start. So, at most $\mathcal{O}(p)$ time is needed. Finally, during the last scan, every processor will assign the suffixes with the value of the temporary memory space if they point to this memory space. The third substep will need at most $\mathcal{O}(\frac{n}{p})$ time. So, totally, $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space are needed to generate DTA.

Lemma 4. *Inducing the order of all Changeable suffixes based on DTA and $AliSA$ can be done in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.*

Proof. Generating the relative order of suffixes in each Changeable group based on its DTA can be done in $\mathcal{O}(\frac{n}{p})$ time because the length of each suffix to be sorted will be less than D, and we have at most $\mathcal{O}(\frac{n}{p})$ such suffixes for each processor. It will need to scan all the corresponding distinguishable tail suffixes to induce the order of Changeable suffixes. The total number of distinguishable tail suffixes is the same as the total number of Changeable suffixes that is no more than $\mathcal{O}(\frac{n}{p})$ on each processor. So, the induce procedure also can be done in $\mathcal{O}(\frac{n}{p})$ time. The total space to keep the $AliSA$ and the temporary string is no more than $\mathcal{O}(n)$. So the fourth substep can also be done in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space.

Theorem 2. *For a string S with length n, its suffix array can be generated in $\mathcal{O}(\frac{n}{p})$ time on p processors with $\mathcal{O}(n)$ space in parallel.*

Proof. Based on the above theorem and the lemmas, every algorithm step can be done in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ space. So, after we add them together, we will get the conclusion.

5 Related work

There have been many works on the suffix array construction algorithm since suffix array was invented in 1990 by Manber and Myers[21]. "Induce", which means that we can take advantage of the existing order of some suffixes to induce the order of other suffixes, is an essential technique in suffix sorting.

Although prefix-doubling [28] adopts the induce technique, it cannot reduce the problem size step by step. This is why it cannot achieve $\mathcal{O}(n)$ time complexity. The following works [13, 10, 12] recursively solves the problem by constructing a reduced problem and employing the induce technique to sort the suffixes.

All existing parallel suffix array construction algorithms were trying to parallelize one or combined sequential algorithm. Futamura *et al.* [5] gave the early effort to parallel the prefix-doubling method. Larsson *et al.* [19] implemented optimized methods based on the previous prefix-doubling technology and improved its performance in parallel. Osipov *et al.* [27] implemented prefix-doubling algorithm on GPUs. Flick and Aluru [4]'s parallel MPI-based implementation of prefix-doubling method can achieve very high practical performance on human genome datasets.

Kulla *et al.* [15] parallelized the sequential DC3 method, which regularly samples the string to build a smaller $\frac{2}{3}n$ problem. Deo *et al.* [2] further implement the DC3 method on GPUs. Shun [29]'s parallel skew (DC3) algorithm could achieve good performance on shared-memory multicore computers. Wang *et al.* [31] implemented a hybrid prefix-doubling and DC3 method on GPUs to improve the existing GPU methods significantly.

Lao *et al.* [18, 17] employed pipeline technology to parallelize their previous sequential linear algorithm [26, 25] on multicore computers.

The existing parallel suffix array construction algorithms can significantly improve the practical performance compared with the corresponding sequential algorithms. However, they cannot achieve $\mathcal{O}(\frac{n}{p})$ time complexity. The existing sequential algorithm framework is the major barrier to the existing parallel methods of achieving scalable performance. We develop a parallel framework and propose a parallel induce method to achieve $\mathcal{O}(\frac{n}{p})$ time complexity.

6 Conclusion

The novel idea provided in this paper is that we propose the concept of D limited substring to divide the complete problem with size n into p reduced sub-problems with size $\mathcal{O}(\frac{n}{p})$. String data are increasing significantly, and an optimal parallel suffix array construction algorithm that can handle the problem in $\mathcal{O}(\frac{n}{p})$ is critical for us to handle large strings with scalable performance. The critical technology is parallel induce. The suffixes with long repeat prefixes can induce their order based on their distinguishable tail suffixes in parallel. We take advantage of the existing optimal sequential suffix array construction algorithm as an independent execution unit to generate the order of all D limited substrings that can be used to separate suffixes with long repeat prefixes from those with short unique prefixes.

The simplicity and the $\mathcal{O}(\frac{n}{p})$ time complexity make the proposed $DLPI$ (D Limited Parallel Induce) algorithm very promising to handle huge strings with scalable performance. $DLPI$ is the first parallel suffix array construction algorithm with $\mathcal{O}(\frac{n}{p})$ time complexity. It is optimal in terms of asymptotic time

complexity. We will focus on further reducing the total working space in the future work.

7 Acknowledgement

This research was funded in part by NSF grant number CCF-2109988.

References

1. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Annual Symposium on Combinatorial Pattern Matching. pp. 55–69. Springer (2003)
2. Deo, M., Keely, S.: 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. pp. 197–206 (2013)
3. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings 38th Annual Symposium on Foundations of Computer Science. pp. 137–143. IEEE (1997)
4. Flick, P., Aluru, S.: 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. pp. 1–10 (2015)
5. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting (2001)
6. Helman, D.R., JáJá, J., Bader, D.A.: A new deterministic parallel sorting algorithm with an experimental evaluation. Journal of Experimental Algorithmics (JEA) **3**, 4–es (1998)
7. Homann, R., Fleer, D., Giegerich, R., Rehmsmeier, M.: mkESA: enhanced suffix array construction tool. Bioinformatics **25**(8), 1084–1085 (2009)
8. JéJé, J.: An introduction to parallel algorithms. Reading, MA: Addison-Wesley **10**, 133889 (1992)
9. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science **387**(3), 249–257 (2007)
10. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: International colloquium on automata, languages, and programming. pp. 943–955. Springer (2003)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM (JACM) **53**(6), 918–936 (2006)
12. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. pp. 186–199. Springer (2003)
13. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. pp. 200–210. Springer (2003)
14. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. Journal of Discrete Algorithms **3**(2-4), 143–156 (2005)
15. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Computing **33**(9), 605–612 (2007)
16. Kurtz, S.: Reducing the space requirement of suffix trees. Software: Practice and Experience **29**(13), 1149–1171 (1999)

17. Lao, B., Nong, G., Chan, W.H., Pan, Y.: Fast induced sorting suffixes on a multicore machine. The Journal of Supercomputing **74**(7), 3468–3485 (2018)
18. Lao, B., Nong, G., Chan, W.H., Xie, J.Y.: Fast in-place suffix sorting on a multicore computer. IEEE Transactions on Computers **67**(12), 1737–1749 (2018)
19. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science **387**(3), 258–272 (2007)
20. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. In: International Symposium on String Processing and Information Retrieval. pp. 268–284. Springer (2018)
21. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. siam Journal on Computing **22**(5), 935–948 (1993)
22. Maniscalco, M.A., Puglisi, S.J.: Faster lightweight suffix array construction. In: Proc. of International Workshop On Combinatorial Algorithms (IWOCA). pp. 16–29. Citeseer (2006)
23. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1), 33–50 (2004)
24. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM (JACM) **23**(2), 262–272 (1976)
25. Nong, G.: Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Transactions on Information Systems (TOIS) **31**(3), 1–15 (2013)
26. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Transactions on Computers **60**(10), 1471–1484 (2010)
27. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: International Symposium on String Processing and Information Retrieval. pp. 379–384. Springer (2012)
28. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. acm Computing Surveys (CSUR) **39**(2), 4–es (2007)
29. Shun, J.: Fast parallel computation of longest common prefixes. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 387–398. IEEE (2014)
30. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: the problem based benchmark suite. In: Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures. pp. 68–70 (2012)
31. Wang, L., Baxter, S., Owens, J.D.: Fast parallel skew and prefix-doubling suffix array construction on the GPU. Concurrency and Computation: Practice and Experience **28**(12), 3466–3484 (2016)