Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays

Patrick Flick
Georgia Institute of Technology
Atlanta, Georgia, USA
patrick.flick@gatech.edu

Srinivas Aluru
Georgia Institute of Technology
Atlanta, Georgia, USA
aluru@cc.gatech.edu

ABSTRACT

Suffix arrays and trees are fundamental string data structures of importance to many applications in computational biology. Consequently, their parallel construction is an actively studied problem. To date, algorithms with best practical performance lack efficient worst-case run-time guarantees, and vice versa. In addition, much of the recent work targeted low core count, shared memory parallelization. In this paper, we present parallel algorithms for distributed memory construction of suffix arrays and longest common prefix (LCP) arrays that simultaneously achieve good worst-case run-time bounds and superior practical performance. Our algorithms run in $O(T_{sort}(n,p) \cdot \log n)$ worst-case time where $T_{sort}(n,p)$ is the run-time of parallel sorting. We present several algorithm engineering techniques that improve performance in practice. We demonstrate the construction of suffix and LCP arrays of the human genome in less than 8 seconds on 1,024 Intel Xeon cores, reaching speedups of over 110X compared to the best sequential suffix array construction implementation *divsufsort*.

1. INTRODUCTION

A *Suffix Tree (ST)* of a string is a (compacted) trie of all its suffixes. For a given string S of length n, its *Suffix Tree* can be constructed in $O(n)$ time [31]. *Suffix Trees* are useful for many applications, such as exact and approximate pattern matching, identification of longest common substrings, data compression, and many more.

A *Suffix Array (SA)* is an array containing the lexicographically sorted order of all suffixes of a string, and as such, compactly represents the leafs of a *Suffix Tree*. Suffix arrays were first introduced by Manber and Myers [21] as a space-efficient alternative to suffix trees. They further introduced the first construction algorithm which takes $O(n \log n)$ time, as well as first algorithms making use of the compact representation for exact pattern matching [21]. Subsequently, a great variety of different suffix array construction algorithms have been developed. The survey paper by Puglisi *et al.* [28] gives a good overview of the different approaches. Subsequent algorithms improved the suffix array construction time to $O(n)$ [18, 15].

Pairing the suffix array with the *Longest-Common-Prefix (LCP)* array yields a more powerful data structure. Abouelhoda *et al.* showed that suffix arrays combined with *LCP* arrays have the same capabilities as suffix trees [2]. The *LCP* array can be constructed either during the construction of the suffix array [21, 15], or from a given suffix array [17].

Much recent work on suffix arrays and trees is motivated by their ubiquitous presence in computational biology applications. The advent of high-throughput DNA sequencing is generating billions of short reads per experiment, necessitating the design of parallel algorithms. Note that suffix array algorithms can be trivially extended to a set of strings by concatenating them with appropriate delimiter characters. While the human genome serves as a useful benchmark, particularly for the purpose of comparison with prior state-of-the-art, current genomic datasets can reach sizes that are a hundred times larger, for example in metagenomics. Hence, development of space-efficient distributed memory algorithms are of considerable interest.

In this work, we present fully distributed, parallel suffix and LCP array construction algorithms. In contrast to most previous work, in our approach the input string, as well as all working data, and the output are fully distributed into blocks of size $O(\frac{n}{p})$ across the p processors. Furthermore, our algorithm provides a worst case run-time guarantee of $O(T_{sort}(n,p) \cdot \log(n))$ where $T_{sort}(n,p)$ is the run-time of parallel sorting. Additionally, our approach constructs the LCP array alongside the suffix array, also in a fully distributed fashion.

We provide an efficient, scalable implementation of our algorithm, which constructs the suffix and LCP arrays of the human genome in 7.3 seconds on 1024 Intel Xeon cores. We reach speedups of over 110× compared to *divsufsort* [25], the fastest sequential suffix array construction implementation, commonly used as comparison [27] [30]. We are not aware of any other parallel suffix array or suffix tree construction algorithms which achieve speedups this high.

The rest of the paper is organized as follows: In Section 2, we review the current state of the art. In Section 3, we introduce notation and relevant concepts that will be used throughout the paper. We then describe our suffix array and LCP array construction algorithms in Section 4 and Section 5, respectively. Section 6 contains detailed experimental results, followed by conclusions in Section 7.

SC '15, November 15 - 20, 2015, Austin, TX, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807609

2. RELATED WORK

There are multiple prior approaches for parallelizing suffix array construction. The *Futamura-Aluru-Kurtz (FAK)* [9] algorithm was the first parallel, distributed memory suffix array construction algorithm. In this algorithm, the authors first bucket suffixes according to a w-length prefix, distribute buckets among processors, and then sort each bucket using sequential *multi-key quicksort (MKQS)* [3]. Drawbacks of this approach are the worst-case run-time of $O(n^2)$, and the fact that the input string needs to be in-memory on each processor. A more recent work by Abdelhadi et al. [1] provides a MPI based implementation of the *FAK* algorithm, adapted to cloud computing on *AWS (Amazon Web Services)*.

Another parallel distributed memory algorithm was introduced by Kulla and Sanders [19]. This algorithm is based on the *skew/DC3* linear time, recursive construction algorithm of Kärkkäinen and Sanders [15]. We denote the parallel DC3 algorithm by *pDC3* from here on. The implementation of *pDC3* uses parallel samplesort in each recursive step and thus ceases to be linear time. An additional drawback of this approach is the additional memory consumption required by the recursive decomposition of the string.

Some shared memory algorithms and implementations have been proposed. Homann *et al.* [13] introduced the *mkESA* algorithm, which is a parallelized variant of the *Deep-Shallow* algorithm of Manzini and Ferragina [23]. The authors of *mkESA* report a speedup of less than 2 when using 16 threads. Mohamed and Abouelhoda [24] introduced a parallel bucket pointer refinement *(pBPR)* algorithm based on the algorithm of Schürmann and Stoye [29]. According to the authors, their parallel shared memory implementation beats *mkESA*. However, their results show poor scalability, as they reach a maximum relative speedup of less than 1.7 when using a maximum of 8 threads. Recently, Shun *et al.* ([30]; SC 2014) presented a parallel algorithm for constructing the LCP array given the suffix array and reported a runtime of 105.7 seconds for constructing both the suffix array and LCP array of the human genome in shared memory.

Deo and Keely [7], and Osipov [27] developed shared memory GPU algorithms for suffix array construction. Deo and Keely implemented and tuned a parallel variant of the DC3 algorithm for GPUs and reached significant speedups of up to 35. Additionally, they provided a parallel algorithm for constructing the LCP array, given the input string and the suffix array [7]. Osipov follows a prefix-doubling approach similar to Larsson and Sadakane [20] and provides an efficient GPU implementation. The author reports relative speedups of up to 18 and absolute speedups of up to 6 compared to *divsufsort* [27]. Another GPU implementation utilizing prefix-doubling is available in the *Nvidia NVBIO* library.

Parallel suffix tree construction has been studied extensively. Hariharan gave the first optimal $O(n)$ work CRCW-PRAM algorithm [12]. In this work, however, we are interested in distributed memory parallel and scalable algorithms, for which prior work is more sparse. Ghoting and Makarychev introduced the *WaveFront* algorithm [11, 10], which they developed for both the *External-Memory (EM)* model and for distributed memory parallel machines. The authors report a run-time of 15 minutes for building the suffix tree for the human genome on a 1024 processor IBM Blue Gene/L. Their algorithm has the drawback of having a worst-case quadratic run-time, since each processor needs up to $O(n/k)$ many passes through the input data, where n is the size of the input and k is some constant block size. Mansour *et al.* [22] improve upon *WaveFront* with their *Elastic-Range (ERA)* algorithm. Their algorithm follows a similar principle to *WaveFront* and suffers from a super quadratic worst-case run-time. The authors provide implementations for *EM*, shared-memory and distributed memory architectures. In their publication, the authors report a run-time of 13.7 minutes for constructing the suffix array of the full human genome on 32 cores, thus improving over the 15 minutes of *WaveFront*. A more recent work by Comin and Farreras improves further on *ERA* and *WaveFront* [5]. Their algorithm, called *Parallel Continuous Flow (PCF)*, has a similar time complexity as *Wavefront*, according to the authors. The authors report a run-time of 7 minutes for building the suffix tree of the human genome. To achieve this run-time, they used 172 cores – more than 5 times as many as used for the results of *ERA* for less than 2 times improvement in run-time.

3. DEFINITIONS

We define the suffix array SA, inverse suffix array ISA, and LCP array as commonly done in previous work.

The input string is denoted by $S = s_0s_1s_2 \cdots s_{n-1}$ and its length by $n = |S|$. Each character s_i of the string is an element of a common alphabet $s_i \in \Sigma$ with a total of $\sigma = |\Sigma|$ unique characters. The two notations $S[i]$ and s_i represent the i^{th} character and are used interchangeably. Further, we designate $S_i = S[i..(n-1)] = s_is_{i+1} \cdots s_{n-1}$ as the suffix of S starting at position i.

The *Suffix Array* of S, denoted by SA, is a length n array which contains the lexicographically sorted order of suffixes represented as offsets/indexes into the string. We set $SA[j] = i$, if and only if the suffix S_i has rank j among the lexicographically sorted suffixes. For the construction of the suffix array, we add a virtual character `$` to the end of the string S, such that $S = s_0s_1s_2 \cdots s_{n-1}\$$. This extra character is defined to have the smallest value among all characters of Σ, and hence leads to a unique lexicographical ordering of suffixes.

Conversely, the *Inverse suffix array* ISA (sometimes called the *Rank Array*) contains the rank for each suffix, indexed in the same order as the suffixes appear in the string, hence, $ISA[i] = j$ iff $SA[j] = i$.

The *longest common prefix (lcp)* of two strings is the maximum length of a common prefix in which these two strings are equal. The *Longest Common Prefix* Array (LCP) contains the lcp between each pair of consecutive suffixes of the suffix array. Hence we define:

$$LCP[i] = lcp(S_{SA[i-1]}, S_{SA[i]})$$

Position 0 of the LCP array can be set arbitrarily, we define it as $LCP[0] = 0$.

We denote a prefix of length h of a suffix as its h-prefix. Furthermore, we call the ordering of suffixes by their h-prefixes an h-ordering of all suffixes. Such an h-ordering consists of multiple h-groups, where each h-group consists of suffixes which share a common h-prefix.

The k-mers of a string S are defined as the k-prefixes of all suffixes of S, or in other words, all length k substrings of S.

4. PARALLEL SUFFIX ARRAY CONSTRUCTION

4.1 Prefix Doubling

Our approach is motivated by *Prefix Doubling*, which was first introduced by Karp [16], and then first utilized by Manber and Myers [21] for suffix array construction. The idea behind prefix doubling is as follows.

Given that the suffixes of a string are already sorted by their h-prefix (i.e., a h-ordering of suffixes), we can deduce their $2h$-ordering. Consider any two suffixes with identical h-prefix, say S_i and S_j. The ordering of these two suffixes according to their $2h$-prefix can be deduced by using the current relative ordering of suffixes S_{i+h} and S_{j+h}, which are already sorted according to their h-prefix. Applying this prefix doubling to all suffixes of an h-ordering then yields the $2h$-ordering. Since the longest suffix has size n, all suffixes will be sorted after at most $\log_2(n)$ iterations.

Manber and Myers *(MM)* [21] algorithm induces the $2h$-ordering from the h-ordering with a single linear scan of the current Suffix Array SA. During the scan, MM accesses the ISA in a random access fashion, with no locality. In MM, the SA has to be scanned in linear order, since this is a necessary condition for the correct placement of suffixes to the front of their respective h-groups. These two properties make MM both cache-inefficient and hard to parallelize in a straightforward fashion, especially targeting a distributed memory architecture. Larsson and Sadakane (LS) [20] instead use ternary split quicksort (TSQS) to create the $2h$-ordering from a h-ordering. They sort each h-group separately using $ISA[i+h]$ as the key for each suffix S_i in the h-group. The ISA saves the h-group rank during the construction process. Accesses to the ISA follow a random, non-local order.

Our prefix doubling algorithm follows a similar approach to LS, as in that we use sorting to induce a $2h$-ordering from a h-ordering. In the following, we introduce two approaches to suffix sorting by prefix doubling. The first approach uses global sorting in each iteration in order to induce the $2h$-ordering. The parallelization of this approach reduces to parallel sorting and other common parallel primitives. The second approach improves upon the first by avoiding global sorts, and sorting only non-singleton h-groups, leading to large improvements in run-time.

4.2 Prefix doubling using global sorting

Our algorithm for suffix array construction using global sorting follows multiple steps (see Algorithm 1). We keep the following invariants at the beginning of each iteration. The suffix array SA represents the suffixes, sorted according to their h-prefix. The B array is laid out in the same order as the suffix array SA, and each position i in B contains an identifier for the h-group of suffix $S_{SA[i]}$. We represent each h-group by its leftmost element. We set $B[i] = i$ for the first position i of the h-group and $B[j] = i$ for all further elements $j > i$ of the same h-group. We denote the values in B as h-group ranks of the corresponding suffixes.

Example.

We show an example of this algorithm in Figure 1 for the input string $S =$`mississippi`. The example illustrates the data dependencies and movements for the different steps of the algorithm using gray arrows.

ALGORITHM 1: Suffix Array Construction by global sorting

```
Input: string $S$
Output: Suffix Array $SA$ and inverse suffix array $ISA$
$B = k$-mers of $S$
$\left[\dots, \binom{B[i]}{SA[i]}, \dots\right] = \texttt{sort}([\binom{B[i]}{i} | i = 0 \dots n-1]$ by $B[i])$
$B = \texttt{rebucket}(B)$ // assign $h$-group rank
for $h = k, 2k, 4k, 8k, \dots$ do
    // reorder to string order (in-place)
    for $i = 0, \dots, n-1$ do
        $B'[SA[i]] = B[i]$
    end
    $B = B'$
    if $done$ then
        $ISA = B$
        break
    end
    // shift $B$ by $h$:
    for $i = 0, \dots, n-1$ do
        $B_2[i] = B[i+h]$
    end
    // sort tuples $(B, B_2)$ and keep origin index
    $\left[\dots, \begin{pmatrix} B[i] \\ B_2[i] \\ SA[i] \end{pmatrix}, \dots\right] = \texttt{sort}([\begin{pmatrix} B[i] \\ B_2[i] \\ i \end{pmatrix} | i = 0 \dots n-1]$
    by $B[i], B_2[i])$
    $B = \texttt{rebucket}(B, B_2)$ // assign $2h$-group rank
    $done = \texttt{check-all-singleton}(B)$
end
```

k-mer sorting.

The first step of the algorithm is to generate the k-mers of S. These are the length k prefixes of all suffixes of S. We choose k according to the alphabet size, such that k characters fit into one machine word. Let l be the bit-length of the machine word (e.g. 32 or 64). We then chose k as:

$$k = \lfloor \frac{l}{log_2(\sigma)} \rfloor$$

We store all k-mers of S into an array B, and then sort them lexicographically. To keep track of each k-mer's original position, we treat the k-mer and its string index as a tuple $\langle k\text{-mer}, i \rangle$. After sorting these tuples, the second index yields the SA according to a k-ordering.

Re-bucketing.

After sorting the k-mers, we need to assign the k-group ranks to each position in B. Since each k-group appears in a contiguous block in B and has identical k-mers, we can assign the new k-group ranks using a simple linear scan through all positions i. We keep a single variable g, which keeps the current k-group rank. The group index g gets updated to i whenever we encounter a new k-group, i.e., when $B[i-1] \neq B[i]$. Otherwise, we assign $B[i] = g$ and continue. The re-bucketing within the main loop follows the same procedure, but here we check for new $2h$-groups, which begin whenever $B[i-1] = B[i]$ and $B_2[i-1] \neq B_2[i]$.

Reorder to string order.

For prefix doubling of each string position i, we need the bucket number (h-group rank) for the string position $i+h$. To efficiently pair each string position with this index, we first transform B from the current suffix array ordering into

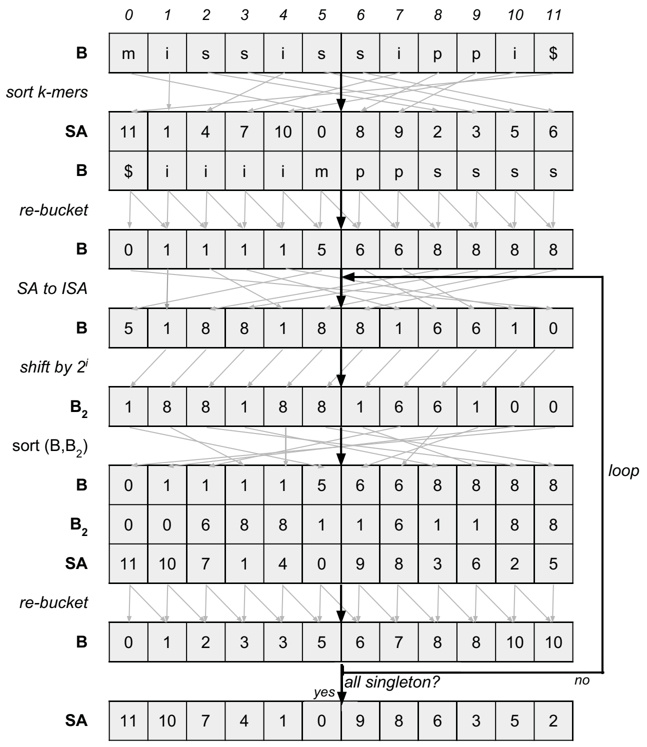


Figure 1: Example of suffix array construction by global sorting for the input string S =mississippi\$ and $k = 1$.

the ordering according to the string indexes. This is effectively equal to the ISA, which for each suffix S_i contains its h-group rank $B[i] = ISA[i]$. Since we know the string position for each position in B via SA, we can perform this reordering in a simple linear scan. If all h-groups are singletons, this reordering of B is equal to the final ISA, which assigns the overall rank for each suffix S_i in $ISA[i]$.

Shifting by h.

In order to perform prefix doubling, we now need to pair each $B[i]$ with the corresponding $B[i + h]$. We do so by creating a new array B_2 and filling each position i with the value in $B[i + h]$. In other words, we are shifting the array B by h positions to the left. This creates spacial locality, necessary for efficient parallelization.

Tuple sorting.

Next, we sort B and B_2 by interpreting each position i as a tuple of size three $\langle B[i], B_2[i], i\rangle\rangle$. After sorting, the third tuple position represents the SA of a $2h$-ordering. We can now re-assign $2h$-group ranks using the re-bucketing procedure explained above.

Termination.

The algorithm terminates if all h-groups are singletons. We check for this condition during the re-bucketing step. To do so, we keep track if $B[i-1] \neq B[i]$ is true for all positions i. In case the ISA is needed alongside the SA, we perform a final reordering prior to terminating the loop. Termination is necessarily reached after at most $\log_2(n)$ iterations.

4.2.1 Parallelization

We parallelize our suffix array construction algorithm using *MPI*. Most steps of the algorithm map to common parallel primitives such as parallel sorting, `all-to-all`, `all-reduce`, `scan`, and `exscan`.

Data distribution.

All data including input, output, and working data is distributed equally among all p processors. Each processor contains $\frac{n}{p}$ contiguous elements of each array (if n is not divisible by p, then either $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$). For the sake of simplifying the presentation, we assume here that p equally divides n. A processor with rank $r \in \{0, \dots, p-1\}$ contains array elements with indexes $r\frac{n}{p}, \dots, (r+1)\frac{n}{p}-1$. Conversely, an element with global index i is located on processor $\lfloor i\frac{p}{n} \rfloor$. The full distribution of all data sets our approach apart from most others, which at the least require the input string S to be accessible from each processor. Our approach thus scales to much larger inputs for which the string S does not fit into the main memory of a single node or processor.

Reordering.

Reordering of the B array from the suffix array order to its string order is done in three steps. First, the elements are bucketed according to the target processor to which they will be sent. Then an `all-to-all` collective communication is used to exchange the buckets between processors. Finally, the ISA order is achieved locally by reassigning elements by their SA index.

Parallel Sorting.

There are many different approaches for sorting on parallel distributed memory architectures. Blelloch *et al.* [4] give a good review. We implement parallel sample sort with regular sampling in our implementation.

Shifting.

Shifting the B array by h positions is a straightforward communication pattern, where one processor communicates with at most two others. Based on the value of h, each processor determines the ≤ 2 processors to which it has to send data and the ≤ 2 processors from which it will receive data. Two point-to-point communications are performed to shift the data.

Re-bucketing.

A h-group might span across multiple processors. Thus we cannot use the same approach, which we used for the sequential algorithm. We instead perform two passes over the data. In the first pass, we set each entry which is at the beginning of its h-group (i.e., $B[i-1] \neq B[i]$ or $B_2[i-1] \neq B_2[i]$) to its own index, while setting all other elements to 0. We then perform a prefix-scan with the max operation as the combination operator. The result of this operation is that $B[i] = i$ only for the first element of each h-group. For all other positions within the h-group, $B[i]$ is the index of the first element of said h-group. Hence, this operation re-establishes the invariant for B.

Complexity.

The overall complexity of Algorithm 1 is $O(T_{sort}(n, p) \cdot \log(n))$, since the doubling approach requires at most $O(\log n)$

iterations and sorting is the most expensive operation in each iteration. For samplesort, the sorting complexity is given by $O(\frac{n}{p}\log\frac{n}{p} + \mu\frac{n}{p} + \tau\log(p))$

Memory Consumption.

If n is the number of characters in the input string, and assuming a word- size of 4 bytes, then our algorithm requires $3 \times 4n = 12n$ bytes for the SA, B, and $B2$ arrays. Since all data (input, working, and output) is fully distributed, the memory consumption per process for these arrays is $12\frac{n}{p}$. For the all-to-all MPI communication, we further allocate receive buffers of 12n additional bytes. Hence, Algorithm 1 requires a total of $25\frac{n}{p}$ bytes per process (compare to the output size of $8n$ for the SA and ISA arrays).

4.3 Avoiding global sorting

For many real world inputs, a significant fraction of h-groups will be fully resolved (become singletons) after only a few iterations. Sorting all positions in each iteration thereafter thus becomes unnecessary overhead. Further prefix doubling needs to be performed only for non-singleton h-groups. For each global position i within such an h-group, we require the current h-group rank for the suffix starting at $SA[i]+h$. When considering the array B in the same order as the SA, this information is not simply accessible and may be localized on any processor. We thus use the ISA to represent this information. For any string position i, $ISA[i]$ contains the current h-group rank of the corresponding suffix S_i. After termination, this is equivalent to the rank of the suffix and thus the complete ISA.

Our second approach, Algorithm 2, improves upon Algorithm 1. Algorithm 1 is still used for the initial k-mer sorting, and for the first few iterations of prefix doubling, as long as the number of elements in non-singleton h-groups is less than some predefined fraction of n. We then switch to Algorithm 2, which takes partially solved SA, B and ISA arrays as input. Here, we explain only the parallel version of Algorithm 2. Hence, we assume that all three arrays, ISA, SA, and B are already equally distributed among all processors using a block decomposition.

Determine non-singleton h-groups.

The first step of this algorithm is to determine all non-singleton h-groups in the B array. We do so by using a local property of B. A position i is part of a non-singleton h-group if either $B[i] \neq i$ (this element is not the representative of its h-group) or $B[i+1] = i$ (the next element is in the same h-group). We can perform this check in a single linear scan. Each processor needs to send its first local element of B to the previous processor on the left, such that each processor can check $B[i+1]$ for its last element $i = r(\frac{n}{p}+1)-1$. We keep track of the indexes of all non-singleton elements in an additional array W (see Algorithm 2).

Prefix Doubling.

Consider any non-singleton h-group G with global indexes $[g, g+1, \ldots, g+|G|-1]$, thus starting on processor $\lfloor g\frac{p}{n}\rfloor$. In order to further resolve this bucket, we need to apply prefix doubling to the items of this h-group. For an element $i \in G$ of this h-group, i.e., suffix $S_{SA[i]}$, we need the corresponding h-group rank for the suffix starting at $SA[i]+h$. This information is contained at $ISA[SA[i]+h]$. Since the

ALGORITHM 2:

```
Input: partially solved $SA$, $ISA$ and $B$, resolved up to prefix
       length $h$
Input: each processor contains $\frac{n}{p}$ contiguous elements of each
       array
Output: final suffix array $SA$ and inverse suffix array $ISA$
// let $r$ be the processors rank
// we use $i$ as a global index into the distributed arrays
// keep track of all non-singleton $h$-groups
$W = \left[i|B[i] \neq i \text{ or } B[i+1] = i, i = r\frac{n}{p} \ldots r(\frac{n}{p}+1)-1\right]$
while $h \leq n$ do
    // request the group rank needed for doubling
    $M = \left[\binom{SA[i]+h}{i}|i \in W\right]$
    $M = \texttt{all-to-all}(M, \texttt{target-processor} = \frac{p}{n}(SA[i]+h))$
    // send responses (read from $ISA$)
    $M = \left[\binom{ISA[i]}{j}|\binom{i}{j} \in M\right]$
    $M = \texttt{all-to-all}(M, \texttt{target-processor} = \frac{p}{n}j)$
    // sort each unresolved $h$-group by using the keys returned
    in $M$
    $W_{new} = \emptyset$
    for each non-singleton $h$-group $G$ do
        // the $h$-group $G$ is a contiguous section of $SA$ and $B$
        sort($G$ by the $ISA$ returned in $M$)
        rebucket(elements of $G$)
        // keep track of non-singletons $2h$-groups
        $W_{new} = W_{new} \cup [i|B[i] = i \text{ and } B[i+1] \neq i, i \in G]$
    end
    // update the $ISA$
    $M = \left[\binom{SA[i]}{B[i]}|i \in W\right]$
    $M = \texttt{all-to-all}(M, \texttt{target-processor} = \frac{p}{n}SA[i])$
    for $\binom{i}{b} \in M$ do $ISA[i] = b$
    // update $W$ and check for convergence
    $W = W_{new}$
    if $W = \emptyset$ then break
end
```

ISA is block decomposed across processors, this may not be available locally. Hence, we send a message to the processor containing the global index $SA[i]+h$ and request the corresponding ISA value. Instead of sending single messages for each non-singleton element, we generate an array M of tuples $\langle SA[i]+h, i\rangle$ for each non-singleton position i. Then we exchange all tuples by bucketing the tuples by their target processor $\lfloor\frac{p}{n}(SA[i]+h)\rfloor$ and using an `all-to-all` communication. The received messages are processed by a linear scan, in which we replace the first tuple element $SA[i]+h$ by $ISA[SA[i]+h]$. This corresponds to the $B_2[i]$ value of Algorithm 1. In a second step, the messages are returned to their origin using another `all-to-all` communication. We can now pair each received value with its SA position using the second tuple element.

Bucket sorting.

The next step consists of sorting each non-singleton h-group using the received ISA values as keys. Since a h-group might span more than one processor, we first sort all buckets which are local to the processor. Then we sort the remaining h-groups (at most two per processor) using up to two parallel steps. In each step, each processor participates in a parallel sort either with its left, right, or both neighboring processors. The sorting induces the $2h$-ordering of the suffix array. Next, we re-bucket each h-group into their new corresponding $2h$-group ranks. Simultaneously we create a new

W_{new} containing only those indexes from W which remain non-singleton.

Update ISA.

As a final step in each iteration, we need to update the ISA with the newly determined $2h$-group ranks for all updated values in B. To do so, we reuse the M array, and fill it with tuples $\langle SA[i], B[i]\rangle$ for each $i \in W$. All messages are exchanged in yet another round of `all-to-all` communication, where each tuple is sent to the processor containing the global index $SA[i]$. The receiving processor can then update its local ISA according to $ISA[SA[i]] = B[i]$.

Memory Consumption.

Algorithm 2 has a higher memory consumption than Algorithm 1 due to the additional W and M arrays needed to keep track of the non-singleton positions and the messages. We switch from Algorithm 1 to Algorithm 2 only when the number of remaining non-singleton elements falls to or below εn, where $\varepsilon < 1$ is a tuning parameter. For example, ε can be set such that the additional memory consumption for W and M remains less than n additional bytes.

Influence of Alphabet Size.

The size of the alphabet $\sigma = |\Sigma|$ is relevant only in the initial k-mer sorting stage. Afterwards, all values are in the form of indexes, each requiring $O(\log n)$ bits independent of σ. The chosen value of k depends on the alphabet size, since we are packing as many characters as possible into a single machine word (i.e., the k-mer), prior to sorting. As such, a smaller alphabet means that more characters can be sorted in the initial stage and thus more iterations can be skipped. However, σ does not influence the overall worst-case complexity.

5. LCP CONSTRUCTION

5.1 Calculating the LCP

Manber and Myers [21] first introduced how to construct the *Longest-Common-Prefix (LCP)* array alongside the suffix array. Our approach follows a similar strategy and makes use of observations from [21] in order to construct the LCP in parallel during the parallel SA construction. The key observation is that whenever new $2h$-groups are formed from a single h-group, the LCP value for the first position of each $2h$-group can be determined.

When the initial 1-groups are formed, the LCP array can be set to a value of 0 for each first position of the 1-groups, i.e., all positions where $S[SA[i-1]] \neq S[SA[i]]$. All other positions in the LCP array are initialized to ∞.

During the construction of the LCP array, we keep the invariant, that after each prefix doubling step from $h \to 2h$, all LCP values with $LCP[i] < 2h$ are set to their final value.

Take two suffixes from any, but different, h-groups, say at positions i and j in the SA, then $lcp(S_{SA[i]}, S_{SA[j]}) < h$. Furthermore, the *lcp* is given by the minimum value of the LCP array within the range between the two h-groups [21]. Since we represent h-groups inside the B array by the index of their first element, the *lcp* between these two suffixes is given by:

$$lcp(S_{SA[i]}, S_{SA[j]}) = \min_{q \in [b_{min}+1, b_{max}]} LCP[q]$$

where $b_{min} = min(B[i], B[j])$ and $b_{max} = max(B[i], B[j])$.

Now, consider any prefix doubling step from $h \to 2h$. The suffixes of a single h-group all share an identical h prefix, and two such suffixes of different but adjacent $2h$-groups $B[i]$ and $B[j]$ $(i < j)$, say $S_{SA[i]}$ and $S_{SA[j]}$, do not have a common prefix of length $2h$. Thus these suffixes must have a LCP value with: $h \leq lcp(S_{SA[i]}, S_{SA[j]}) < 2h$. Hence, we can determine the *lcp* between the two $2h$-groups as:

$$lcp(S_{SA[i]}, S_{SA[j]}) = h + lcp(S_{SA[i]+h}, S_{SA[j]+h})$$

Since $lcp(S_{SA[i]+h}, S_{SA[j]+h}) < h$, we can use the above invariant and property to calculate the LCP directly. Furthermore, we have the h-group ranks for the strings $S_{SA[i]+h}$ and $S_{SA[j]+h}$ available locally during the prefix doubling step of the suffix array construction, since these are used as the key for sorting each h-group. We can thus calculate the LCP value for new $2h$-group boundaries via:

$$LCP[B[j]] = h + \min_{q \in [a_{min}+1, a_{max}]} LCP[q]$$

where $a_{min} = min(B_2[i], B_2[j])$ and $a_{max} = max(B_2[i], B_2[j])$.

5.2 LCP construction during SA construction

LCP of k-mers.

We incorporate the LCP construction into our prefix doubling approach. Initially, our approach sorts by k-mers, and as such, 1-groups are never present (unless $k = 1$). Thus, the LCP array has to be initialized from the sorted order of k-mers. We linearly scan the sorted k-mers and determine positions where new k-groups begin (two consecutive but different k-mers). Since the k characters of a k-mer are stored contiguously in a single machine word, we can determine the *lcp* between two k-groups using bitwise comparisons. To do so efficiently, we use bitwise `xor` between the two bordering k-mers and then use intrinsics to determine the highest set bit.

Doubling.

During the prefix doubling procedure, we have to determine the minimum over ranges of the LCP for each new $2h$-group boundary. Manber and Myers [21] use a size n search tree. We implement the succinct *Range-Minimum-Query (RMQ)* from Fisher and Heun [8] for this purpose. The RMQ requires only $2n + o(n)$ additional bits, is constructed in $O(n)$ time, and querying for the minimum of a range takes constant time $O(1)$.

5.3 Parallel LCP construction

LCP of k-mers.

Parallelizing this step is straightforward, since it requires a simple linear scan. We need to send the last element of each processor to the next processor on the right, since this is needed for determining the *lcp* for the first element on each processor.

Doubling.

During the doubling step, we need to solve multiple (one for each new $2h$-group) *Range-Minimum-Queries*, where the range might span across multiple processors. We propose the following, hierarchical method for *bulk-parallel RMQs*.

Bulk-parallel Range-Minimum-Queries.

Each processor first constructs the RMQ for the local LCP array of size $O(\frac{n}{p})$. Then each processor determines its local minimum, and sends it to all other processors using a collective `all-gather` operation. We then create a RMQ of the processor minimas on each processor.

In order to solve a minimum query for a range $[a, b]$ ($a < b$), we first need to determine the processors on which a and b are located. Given an equal block decomposition with $\frac{n}{p}$ elements per processor, the corresponding processors are $p_a = \lfloor a\frac{p}{n} \rfloor$ and $p_b = \lfloor b\frac{p}{n} \rfloor$. We distinguish between three cases:

(a) $p_a = p_b$: In this case the query can be answered by a single processor. We thus send a message (i, a, b) to processor p_a, where i is the global position which generated this query. This field is used as a return address.

(b) $p_a + 1 = p_b$: In this case, we need to send queries to two processors. We send $(i, a, (p_a+1)\frac{n}{p} - 1)$ to processor p_a and $(i, p_b\frac{n}{p}, b)$ to processor p_b. In order to find the total minimum, the minimum of the two answers is used.

(c) $p_a + 1 < p_b$: In this case the query spans across more than two processors. Here, we send the same queries as in case *(b)*. However, we need to combine the answers to these queries with the minimum of the intermediate processors. We thus use the RMQ of processor minimums to get the minimum over the range $[p_a+1, p_b-1]$ in constant time. Taking the minimum of this and the received minimums yields the overall minimum of the requested range.

Instead of sending single messages, we first generate all tuples locally and then exchange messages using a single `all-to-all` collective communication. Each processor then executes all received queries and replaces the entries (i, a, b) by the minimum position and minimum value in the range. Each query tuple is updated with the results as:

$$(i, \min_{j\in[a,b]} LCP[j], argmin_{j\in[a,b]} LCP[j])$$

A final collective `all-to-all` returns the results to those processors which posted the queries, i.e., the processor containing global index i.

Complexity.

We construct the local RMQs in each iteration of the prefix doubling. This takes $O(\frac{n}{p} + p)$ time, since there are $\frac{n}{p}$ local elements. Adding this to each iteration of the suffix array construction algorithm does not change its total complexity. The number of total queries is bound by positions in the LCP, since each LCP position gets set only once. Thus there can be at most $O(n)$ queries throughout the construction, each taking constant time. Overall, the complexity of suffix array construction dominates the LCP construction.

Memory Consumption.

If the LCP array is computed along the suffix array, an additional $4n$ bytes are required for the LCP array itself. During the LCP construction, additionally $2n + o(n)$ bits are needed for the succinct range-minimum-query, in addition to the memory for the packed message exchange. The number of these messages depends on the number of new bucket boundaries in each iteration and could potentially reach $3 \times 4n = 12n$ bytes total in the worst case. However, note that the $12n$ bytes receive buffer required by the SA construction is needed only temporarily during the update of the SA, whereas the LCP update happens after the buffer memory is already freed. Thus the total memory consumption for constructing the LCP increases the total memory consumption by no more than the $4n$ bytes required for the LCP plus $2n + o(n)$ bits, which is just insignificantly more than the memory required for storing the LCP array itself.

6. EXPERIMENTS AND RESULTS

6.1 Systems and Data sets

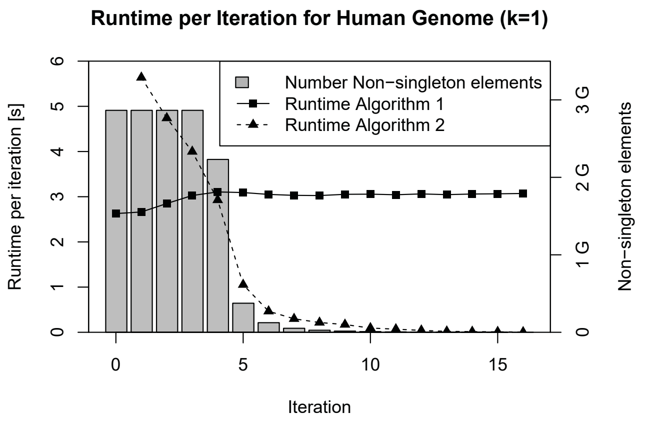
We implemented our suffix array and LCP array construction algorithms using *C++11* and *MPI*. Our code is compiled with `MVAPICH2 v 1.7` and `gcc 4.8` using the optimization flags `-O3 -march=native`. We perform our experiments on an Intel Xeon Infiniband cluster. Each node consists of two 2.0 GHz 8-Core Intel E5 2650 processors and 128 GB main memory. The nodes are interconnected with QDR (40Gbit) InfiniBand. For the experimental evaluation, we use up to 100 identical nodes, corresponding to 1600 cores. All reported run times are averaged over multiple executions.

Since our work is motivated by biological applications, we evaluate the performance of our algorithm on the human (*H.sapiens*) genome. The human genome has a size of approximately 3 billion nucleotides with an alphabet size of 4 (A, C, T, and G). We use the reference genome from the 1000 Genomes Project [6] version `GRCh37`. Additionally, we show performance results for the much larger pine (*Picea abies*) genome, which has a length of over 12 billion nucleotides. We use the *P.abies* assembly version 1.0 published by Nystedt *et al.* [26].

6.2 Performance of Our Algorithms

We introduced two methods for parallel suffix array construction. The first (Algorithm 1) performs global sorting in each iteration, whereas Algorithm 2 improves upon the first algorithm by continuing to sort only non-singleton elements. Figure 2 shows why this is of significant advantage. Both graphs show the run-time spent in each individual iteration for constructing the suffix array of the human genome on 1024 cores. Additionally, we show how many non-singleton elements are left in each iteration. For the *left* Figure, we initially sort suffixes only by their first character, i.e., $k = 1$. Thus, suffixes are sorted by their prefix of size 2^i after iteration i. Initially, Algorithm 2 is more than a factor of 2 slower than Algorithm 1 due to its additional overhead involved in identifying and avoiding unnecessary sorting of singleton elements. However, since the number of non-singleton elements decreases drastically after iteration $i = 5$, the approach taken by Algorithm 2 is far superior in later iterations.

This observation motivated us to design a combined approach: Only a single iteration of Algorithm 1, which sorts pairs of k-mers for $k = 21$, is required until less than $\frac{1}{10}^{\text{th}}$ of elements are non-singletons. Hence, Algorithm 2 is used for all subsequent iterations. The second graph in Figure 2 shows the run-time per iteration for the combined approach. The first iteration (≈ 3.2 seconds) is the most expensive step



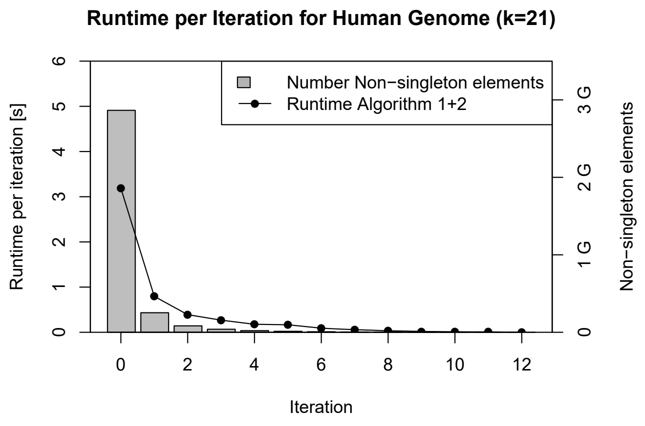


Figure 2: Run-time of each iteration of Algorithm 1 and Algorithm 2 and the number of non-singleton elements in each iteration. The *left* graph shows the run-time for both algorithms, given that the first sorting is performed on k-mers with $k = 1$. The *right* graph shows the time per iteration for our combined algorithm, which first sorts pairs of k-mers with $k = 21$, and then switches to Algorithm 2. Timing results are for the human genome on 1024 cores (64 nodes).

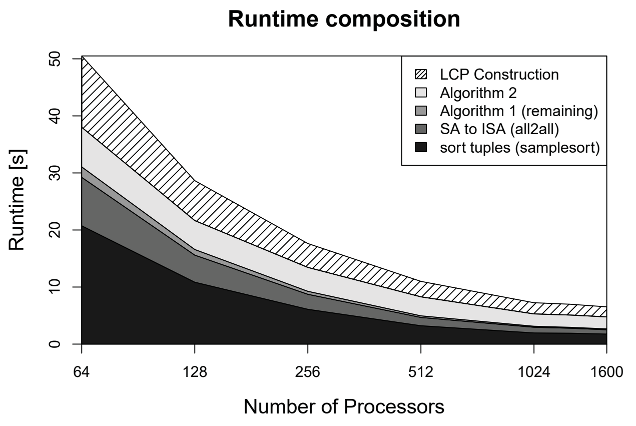


Figure 3: Run-time composition for suffix array and LCP construction of the human genome using up to 1600 cores (100 nodes).

of the full suffix array construction, which completes after a total of 5.3 seconds.

Figure 3 shows the time spent in each part of the algorithm, including the additional time needed for the construction of the LCP array. This composition of the total runtime is shown for different numbers of cores, up to 1600 (100 nodes). The initial sorting of tuples and the consecutive reordering of elements from the SA to the ISA ordering is the largest fraction of the total run-time. The optional LCP construction increases the total cost by at most 30% of the time required for construction of the suffix array.

6.3 Comparison with Prior State-of-the-art

We compare our algorithm and implementation against multiple other methods. The *divsufsort* suffix array implementation [25] serves as the sequential comparison, since it is open source and is known as one of the fastest and lightweight suffix array construction implementations. Furthermore, we run the shared-memory parallel suffix array and LCP construction tool *mkESA* [13]. Additionally, we directly compare our implementation with the *FAK* implementation *cloudSACA* by Abdelhadi *et al.* [1]. We ran all aforementioned suffix array construction algorithms on the same hardware described earlier, thus allowing direct comparison. Due to the non-availability of code from other implementations [19, 11, 22, 5], we could not directly compare with their performance. However, all of these approaches report run-times many times larger than ours.

Method	H 2G	H 3G	P 12G
divsufsort	424.5	586.4	X
mkESA (1)	586.6	1,123.0	X
mkESA (4)	462.6	759.0	X
cloudSACA (128)	40.6	X	X
Our method (128)	16.3	22.1	142.6
Our method (1600)	3.5	4.8	14.8

Table 1: Run-times of different methods in seconds. The numbers in parentheses denote the number of threads or cores used. The times are for constructing the suffix array for the first 2 billion nucleotides of the human genome (H 2G), the entire human genome (H 3G), and the pine genome (P 12G). The character `X` denotes failure to execute due to running out of memory or not supporting large inputs.

In Table 1, we show the measured run-times of *divsufsort*, *mkESA*, *cloudSACA*, and our approach. The MPI-based *cloudSACA* implementation of the *FAK* algorithm fails for inputs larger than 2GB, which is due to a limitation in the algorithm and implementation. The algorithm requires the input string to be available in memory of every process and thus cannot scale to inputs larger than main memory. The *cloudSACA* implementation fails for inputs larger than 2GB, since it tries to send the complete input string to all pro-

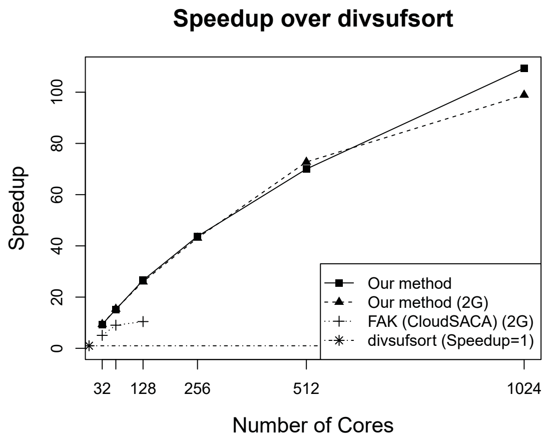


Figure 4: Speedup achieved by our suffix array construction algorithm and the *FAK* implementation of Abdelhadi [1]. Speedup is calculated with respect to the sequential algorithm and implementation *divsufsort*. Results are for the human genome. The *(2G)* denotes that the input is limited to only the first 2 billion nucleotides of the human genome.

cesses using an MPI_Scatter operation. In order to compare the performance of our algorithm with this approach, we use the first 2GB of the human genome as an additional input case. *cloudSACA* reaches a speedup of a little over 10× when using 128 processes (see also Figure 4). In our experiments *cloudSACA* crashed for any number of processes ≥ 256. The *mkESA* tool displays only limited scalability, since it allows only a maximum of 4 threads, and when using this many threads, the improvement in run-time remains limited. Even when using all 4 threads *mkESA* still remains slower than the sequential *divsufsort*. All these methods fail for the pine genome due to its much larger size. The sequential program *divsufsort* runs out of memory, and *mkESA* shows an error that it can not build indexes for this input size, whereas *cloudSACA* is limited to 2GB of input as explained above.

Figure 4 shows the speedup of our algorithm and *cloudSACA* for constructing the suffix array for the first 2 billion nucleotides of the human genome. We calculate the speedup based on the sequential run-time of *divsufsort*. Additionally, we plot the speedup of our method for the full human genome. Our method reaches speedups of over 110× when using 1024 cores. As such, our algorithm outperforms *cloudSACA* by a large margin.

6.4 Scalability and Large Problems

Our approach scales to well above 1024 cores and can handle large genomes. Table 2 shows the run-times for constructing the suffix array with or without the LCP array for both the human genome and the pine genome. Construction of the suffix array for the human genome takes 4.8 seconds without and 6.5 seconds with the LCP array on 100 nodes. For the pine genome our algorithm runs in 14.8 seconds and in 20.2 seconds when the LCP array is constructed alongside. Going from 1024 to 1600 cores yields only a small improvement in run-time for the human genome, due to its already small local size. For 1024 cores, the local input for each process is a mere 3 MB. For the larger pine genome, however, we observe better scalability from 1024 to 1600 cores.

Cores	**H**	**H** (+LCP)	**P**	**P** (+LCP)
128	22.1	28.6	107.1	142.6
256	13.4	17.5	59.3	79.2
512	8.4	11.0	34.2	43.8
1024	5.4	7.3	19.8	25.9
1600	4.8	6.5	14.8	20.2

Table 2: Run-times for constructing the suffix array with (+LCP) and without the LCP array for the human genome (H) and the pine genome (P). The run-times are given in seconds for up to 1600 cores on 100 nodes.

7. CONCLUSIONS

We introduced new parallel distributed memory suffix array and LCP array construction algorithms that scales to large inputs and a large number of nodes and cores. Our implementation reaches speedups of above 110× over *divsufsort*, one of the fastest known sequential implementations. Our implementation indexes the full human genome in less than 8 seconds, many times faster than any previously reported run times. Additionally, our method scales to larger inputs than any previous published results, and indexes a large 12 billion nucleotide plant genome in less than 15 seconds.

The suffix array and LCP array give the same expressive power as suffix trees. Currently, distributed memory parallel methods for suffix tree construction have quadratic time worst case run-time, and the fastest approach still requires approximately 7 minutes to construct the suffix tree of the human genome. Using the suffix array and LCP array constructed with our algorithm, we can construct the suffix tree by solving the *All-Neighbor-Smallest-Value (ANSV)* problem [14].

Further improvements to our algorithms are possible by using hybrid parallelism. Local sorting and other operations could be performed by shared memory parallel implementations using *OpenMP*, or GPUs using *CUDA* or *OpenCL*.

Acknowledgments

This research is supported in part by the National Science Foundation under CCF-1360593 and IIS-1416259. The CyEnce cluster used for experiments was supported by the National Science Foundation under CNS-1229081.

8. REFERENCES

[1] A. Abdelhadi, A. Kandil, and M. Abouelhoda. Cloud-based parallel suffix array construction based on mpi. In *Biomedical Engineering (MECBME), 2014 Middle East Conference on*, pages 334–337. IEEE, 2014.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[3] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, volume 97, pages 360–369, 1997.

[4] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM, 1991.

[5] M. Comin and M. Farreras. Efficient parallel construction of suffix trees for genomes larger than main memory. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 211–216. ACM, 2013.

[6] . G. P. Consortium et al. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[7] M. Deo and S. Keely. Parallel suffix array and least common prefix for the gpu. In *ACM SIGPLAN Notices*, volume 48, pages 197–206. ACM, 2013.

[8] J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470. Springer, 2007.

[9] N. Futamura, S. Aluru, and S. Kurtz. Parallel suffix sorting. pages 76–81, 2001.

[10] A. Ghoting and K. Makarychev. Indexing genomic sequences on the ibm blue gene. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 61. ACM, 2009.

[11] A. Ghoting and K. Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 827–840. ACM, 2009.

[12] R. Hariharan. Optimal parallel suffix tree construction. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 290–299. ACM, 1994.

[13] R. Homann, D. Fleer, R. Giegerich, and M. Rehmsmeier. mkesa: enhanced suffix array construction tool. *Bioinformatics*, 25(8):1084–1085, 2009.

[14] J. JaJa and K. W. Ryu. Optimal algorithms on the pipelined hypercube and related networks. *Parallel and Distributed Systems, IEEE Transactions on*, 4(5):582–591, 1993.

[15] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[16] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136. ACM, 1972.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial pattern matching*, pages 181–192. Springer, 2001.

[18] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 200–210. Springer, 2003.

[19] F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

[20] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[21] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[22] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment*, 5(1):49–60, 2011.

[23] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[24] H. Mohamed and M. Abouelhoda. Parallel suffix sorting based on bucket pointer refinement. In *Biomedical Engineering Conference (CIBEC), 2010 5th Cairo International*, pages 98–102. IEEE, 2010.

[25] Y. Mori. libdivsufsort. https://github.com/y-256/libdivsufsort.

[26] B. Nystedt, N. R. Street, A. Wetterbom, A. Zuccolo, Y.-C. Lin, D. G. Scofield, F. Vezzi, N. Delhomme, S. Giacomello, A. Alexeyenko, et al. The norway spruce genome sequence and conifer genome evolution. *Nature*, 497(7451):579–584, 2013.

[27] V. Osipov. Parallel suffix array construction for shared memory architectures. In *String Processing and Information Retrieval*, pages 379–384. Springer, 2012.

[28] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4, 2007.

[29] K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[30] J. Shun. Fast parallel computation of longest common prefixes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 387–398. IEEE Press, 2014.

[31] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.