Distributed Enhanced Suffix Arrays: Efficient Algorithms for Construction and Querying

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

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

ABSTRACT

Suffix arrays and trees are important and fundamental string data structures which lie at the foundation of many string algorithms, with important applications in computational biology, text processing, and information retrieval. Recent work enables the efficient parallel construction of suffix arrays and trees requiring at most $O(n/p)$ memory per process in distributed memory.

However, querying these indexes in distributed memory has not been studied extensively. Querying common string indexes such as suffix arrays, enhanced suffix arrays, and FM-Index, all require random accesses into $O(n)$ memory - which in distributed memory settings becomes prohibitively expensive.

In this paper, we introduce a novel distributed string index, the Distributed Enhanced Suffix Array (DESA). We present efficient algorithms for the construction and querying of this distributed data structure, all while requiring only $O(n/p)$ memory per process. We further provide a scalable parallel implementation and demonstrate its performance and scalability.

CCS CONCEPTS

• **Theory of computation** → **Sorting and searching; Parallel algorithms; Distributed algorithms; Pattern matching.**

ACM Reference Format:

Patrick Flick and Srinivas Aluru. 2019. Distributed Enhanced Suffix Arrays: Efficient Algorithms for Construction and Querying. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19)*, November 17–22, 2019, Denver, CO, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3295500.3356211>

1 INTRODUCTION

The *suffix tree* (ST) of a string S is a compacted trie of all of its suffixes. For a string S of length n , its ST can be constructed in $O(n)$ time [36]. The ST allows searching for a pattern P in S in $O(|P|)$ time (independent of n), as well as reporting of all occurrences of P in additional time that is linear in the number of occurrences. Since its invention in 1973, suffix tree has become the de facto data structure for the design of numerous clever string algorithms [20]. While versatile and powerful, suffix trees have a large space requirement,

prompting the hunt for data structures with similar power but that are space-efficient.

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*. Manber and Myers introduced suffix arrays for space-efficiency [31], and showed it only requires an extra logarithmic cost to support pattern matching ($O(|P| + \log n)$ time). Suffix arrays can also be constructed in linear time [23, 27], although algorithms with higher worst-case complexity are often shown to outperform in practice. Puglisi *et al.* [34] give a good survey of suffix array construction algorithms. An associated data structure is the *LCP* array, which stores the length of the longest common prefix between every pair of consecutive suffixes in the suffix array. It can be constructed in linear time either as a byproduct of suffix array construction [23, 31], or directly from a given suffix array [25].

Enhanced suffix arrays (ESA) were introduced by Abouelhoda *et al.* as a space-saving replacement for suffix trees [3]. The ESA consists of the suffix array, LCP Array, an interval tree over the LCP array, and a child-table. Abouelhoda *et al.* [3] showed how their data structure supports a majority of operations supported by suffix trees.

To handle very large string collections, such as billions of DNA fragments sampled for (meta-)genomic studies, distributed representations and querying of string data structures are needed. Recent efforts focused on efficient parallel construction of suffix trees [7, 15, 17, 18, 30, 32, 35] and arrays [8, 14, 16, 28, 29, 33, 37].

Querying of distributed suffix arrays and trees has only been studied sporadically and to a lesser extent [5, 10, 13]. While these constitute interesting results, they suffer from significant limitations that restrict their usability. For example, [5, 10] are theoretical works that result in excessive communication when implemented in practice, and no implementations are available. The more recent work of Fischer *et al.* [13] restricts query strings to be smaller than a constant threshold size.

In this paper, we present the *Distributed Enhanced Suffix Array*, a novel distributed string index which improves the state-of-the-art for efficient querying in multiple aspects. We present algorithms for construction and querying that are both theoretically and practically efficient, achieve memory scalability, and exhibit good scaling results in practice when compared against well engineered sequential implementation. Specific details of contributions are described in Section 2.3, after sufficient background details are presented.

1.1 Definitions and Notations

A *String* or *Text* $S = s_0s_1 \dots s_{n-1}$ is a sequence of characters or symbols s_i . We denote the string length as $|S| = n$. The string consists of characters from a shared alphabet Σ with $\sigma = |\Sigma|$ unique

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '19, November 17–22, 2019, Denver, CO, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6229-0/19/11...\$15.00

<https://doi.org/10.1145/3295500.3356211>

symbols, e.g., a genome can be represented as a string with the alphabet $\Sigma = \{a, c, g, t\}$. The string S doubles as an array: we can access any character $S[i] = s_i$ in constant time.

Given a *pattern* string $P = p_0p_1 \dots p_{m-1}$ of length m , we are interested in finding if and where the string P occurs within a larger string S , where commonly $n \gg m$. This problem is referred to by multiple names, such as *String Matching*, *Pattern Matching*, or *Substring Search* problem. Classic algorithms such as the *Rabin-Karp* algorithm [24], the *Knuth-Morris-Pratt* algorithm [26], and the *Boyer-Moore* algorithm [6] pre-process the pattern P and subsequently scan the larger string S in linear time to identify all occurrences of P in S .

(Sub)string Indexes are data structures created by pre-processing the string S , subsequently allowing much faster pattern search. Examples of such string indexes include the k -mer index, suffix tree, suffix array, and FM-Index.

Suffix trees and *suffix arrays* index every *suffix* of the given string S , and allow querying for any substring of S . The i^{th} suffix of S is defined as

$$\text{Suf}(i) = S[i..(n-1)] = s_i s_{i+1} \dots s_{n-1},$$

i.e., the suffix of S starting at position i . For simplifying presentation and to avoid ambiguity, an additional terminating character $\$$ is often appended to the input string $S = s_0 s_1 \dots s_{n-1} \$$ (i.e., $s_n = \$$). This ensures that no suffix is a prefix of another suffix. Typically, $\$$ is treated as the lexicographically smallest character, making $\text{Suf}(n)$ the lexicographically smallest suffix.

The *suffix tree* (ST) of a string S is a (compacted) trie of all the suffixes of S . The leaves of the tree are the lexicographically sorted suffixes $\text{Suf}(i)$ of S , represented by i - their offset in S . Each edge in the tree is labeled by a substring of S , such that any root to leaf path spells out the whole suffix corresponding to the leaf. Edges are compacted, such that each internal node has at least two children, and no two outgoing edges can start with the same character.

The *suffix array* (SA) of a string S is an array of length n containing the lexicographically sorted order of all suffixes of S . Each suffix $\text{Suf}(i)$ is represented by its starting position i , so that $\text{SA}[j] = i$, if and only if the suffix $\text{Suf}(i)$ has rank j among the lexicographically sorted suffixes.

The *longest common prefix* (lcp) of two strings is the maximum length for which the prefixes of the two strings are identical. The *Longest Common Prefix* (LCP) array is an array of length n , which contains the lcp between each pair of consecutive suffixes as they appear in the suffix array.

$$\text{LCP}[i] = \text{lcp}(\text{Suf}(\text{SA}[i-1]), \text{Suf}(\text{SA}[i]))$$

Figure 1 illustrates the relationship between suffix trees, suffix arrays, and LCP arrays. A subtree of the ST starting at node v corresponds to a range $[l, r]$ of suffixes $\text{SA}[l], \dots, \text{SA}[r]$. This subtree of string-depth t contains suffixes sharing a prefix of t or more common characters, thus the LCP array in this range contains all values $\geq t$ and is surrounded by values $< t$. The d children of v correspond to d intervals in the LCP array of all values $> t$ separated by $d-1$ positions where $\text{LCP}[i] = t$. In the context of (enhanced) suffix arrays, each node v is represented as its $[l, r]$ interval, and we refer to the children as the *child-intervals* of the interval $[l, r]$.

Figure 1: Relationship between suffix trees, suffix arrays, and LCP arrays: a subtree starting at an internal node v with string-depth t and child nodes $u_0, u_1, \dots, u_{d(v)-1}$.

A *Range-Minimum-Query* (RMQ) over an array A of size n allows efficiently finding the minimum value (as well as its position) in any sub-array of A . Given an index range $[a, b]$ in A find:

$$\arg \min_{i \in [a, b]} A[i]$$

To support multiple queries efficiently, the array A is pre-processed to create a suitable data structure. Fischer and Heun introduced a linearly constructible succinct RMQ data-structure which can answer any query in $O(1)$ time using only $2n + o(n)$ extra bits [12].

Distributed Arrays. We call a sequence $A = [a_0, a_1, \dots, a_{n-1}]$ (equally) *block distributed* onto p processors, iff processor i contains the contiguous segment $[a_{i \frac{n}{p}}, \dots, a_{(i+1) \frac{n}{p}-1}]$. For the sake of simplifying the presentation, we assume here that n is divisible by p . Further, an array or sequence A is *cyclically distributed* if element a_i is contained in processor $(i \bmod p)$. This is also sometimes called a *round-robin* or *multiplexed* distribution of A .

2 RELATED WORK AND CONTRIBUTIONS

2.1 Distributed SA and ST construction

A variety of algorithms have been developed for the parallel construction of suffix trees and arrays. Parallel suffix array construction has been studied for shared memory [8, 29, 33], as well as distributed memory [2, 16]. Many parallel algorithms for suffix tree construction exist: theoretical algorithms for the PRAM [4, 21, 22], practical algorithms for shared-memory machines [35], and algorithms for external or distributed memory [7, 17, 18, 32].

State-of-the-art PRAM algorithms for suffix tree, suffix array, and FM-index construction were recently introduced by Shun and Blelloch [35] and Labeit et al. [29]. These algorithms perform extremely well on shared-memory systems, but do not generalize to distributed memory. Whereas the shared-memory parallel algorithms assume random access to the entire string, suffix tree, or array, in distributed memory each processor generally contains only a part of the whole input or index.

Our earlier work allows the efficient distributed construction of suffix arrays [14] and suffix trees [15] in distributed memory, while using at most $O(n/p)$ memory per processor. Not only do these algorithms achieve superior run-time complexity, they are also designed to be memory-scalable. Previous approaches required $O(n)$

memory per process [2, 7, 16, 18, 32], thus limiting the problems that could be solved by the memory size of a single compute node. With a memory requirement of $O(n/p)$ per processor, potentially any problem can be solved using a sufficient number of processors and compute-nodes.

2.2 Distributed string search

The traditional algorithms for querying SA, ST, ESA, or compressed indexes such as FM-index, all require random accesses to $O(n)$ sized data structures, which in distributed memory becomes highly inefficient and prohibitively expensive.

Given that the input text is split equally across processors, a suffix array could be constructed by each processor independently using only its local text segment. Executing a large number of queries on these *local indices* however does not scale since every query needs to be executed by every processor. Using a distributed suffix array instead allows to parallelize over the queries. Arroyuelo et al. studied the theoretical communication complexities of executing the traditional SA binary search on these suffix array representations [5]. Further, they introduced and analyzed multiple other suffix array representations, e.g., by distributing the SA round-robin. These communication intensive search methods are based on binary searching the different suffix array representations and are therefore still dependent on $\log(n)$.

Ferragina and Luccio proposed an index which is created by distributing suffixes round-robin across processors and building local tries on each processor [10]. The characters of each query are also multiplexed (round robin) across all processors, and processors work together to answer queries. The query algorithm achieves a constant number (9) of BSP super steps with optimal communication and computation complexities each, but only if the query length exceeds the number of processors $m > p$ – a key limitation that greatly affects its usefulness in practice.

The key to designing a practically efficient suffix index lies in reducing the required communication steps without compromising the overall complexity. For our approach, we create a two level index. A top level index is shared by all processors, and for a given pattern, points toward which PE is capable of processing it using its local index. Our solution allows answering a set of queries in bulk using only two super steps, each with optimal complexity.

The distributed patricia trie (DPT) introduced by Fischer et al. follows a similar overall approach [13]. The DPT uses the equally block distributed suffix and LCP arrays as input (the output of our implementation of our SA and LCP construction algorithm [14]) and builds a top level trie of all pairs of those suffixes which lie directly at processor boundaries. There is no guarantee of how many characters those suffixes may share in their prefix (worst case is bounded only by $O(n)$). The authors propose to arbitrarily truncate these suffixes to a constant length k ($k = 30$ in their paper). The local indices are then also constructed only up to a maximum depth of k characters. Consequently, queries searching for patterns of length greater than k are not supported by the DPT – a severe limitation.

Figure 2: DESA, Top-Level Index (TLI) and distribution of subtrees.

2.3 Contributions

In this paper, we propose the *Distributed Enhanced Suffix Array (DESA)*, a novel distributed string index with the following contributions and advantages.

- A two-level hierarchical efficient distributed suffix index.
- A novel and space efficient representation of local subtrees – the key component of the local index.
- A novel top level index, the *Top Level Dynamic Trie*, which enables balanced partitioning of subtrees onto processors.
- An efficient distributed bulk string search algorithm, supporting arbitrary pattern lengths and requiring only two communication super steps.
- An efficient distributed memory algorithm for parallel construction of the DESA.
- Both the data structure layout, and the construction and query algorithms, are space-efficient, i.e., use at most $O(n/p)$ memory per processor. This is essential for solving large-scale problems for which it is unrealistic to assume the input and/or data structures fit into the memory of each node.
- An open source distributed memory parallel implementation, with demonstrable performance and scalability to over thousand cores.

2.4 Structure of the DESA

The *Distributed Enhanced Suffix Array (DESA)* is represented in two levels. A small *Top Level Index (TLI)* is shared as a copy on each processor and splits the lower levels into disjoint subtrees of the (virtual) suffix tree (see Figure 2). Note, we do not explicitly construct the suffix tree structure, but introduce space efficient data structures which allow operations equivalent to those on a suffix tree. Nonetheless, it is useful to use the more natural tree vocabulary, and visualize more intuitively the operations on the tree it represents.

A key difference to previous approaches [5, 13] lies in the way we partition the subtrees. Instead of distributing the suffix array into pieces of exactly $\lfloor \frac{n}{p} \rfloor$ elements, we propose to re-partition subtrees such that each subtree is fully contained within a processor. In Section 4 below, we discuss variants of the *Top Level Index* and how the novel *Top Level Dynamic Trie* can be used to find a balanced partitioning of subtrees.

The subtrees are represented space efficiently as contiguous segments of the suffix array, LCP array, and a character array. A subtree corresponds to a range $[l, r)$ in these arrays. Conventional methods

for querying similar space efficient tree representations require random access into $O(n)$ memory even when querying within a small subtree. In the next section below, we discuss enhanced suffix array query methods and show how they can be augmented to allow querying within a subtree of size k accessing only local data in a size k data range.

3 LOCAL SUBTREE STRUCTURE AND QUERYING

3.1 Background: Enhanced Suffix Arrays

Top-down traversal of a suffix tree finds many applications, most notably exact string pattern matching for a given pattern P . Starting at the root, the characters of the pattern are matched one-by-one to the characters on the unique matching path (if one exists) from the root to a leaf. If the node representation of the ST allows constant time lookup of outgoing edges by character, the pattern search has complexity $O(m)$.

Using a suffix array, the pattern search can be performed by a binary search in $O(m \log(n))$ time [31]. By additionally using the LCP array, the time complexity can be improved to $O(\log(n) + m)$ [31] – still dependent on n and thus worse than a suffix tree. Abouelhoda et al. introduced the *enhanced suffix array*, consisting of the *suffix array*, *LCP array*, an *LCP interval tree*, and a *child-table* [3]. Using the additional *child-table*, Abouelhoda et al. show how to achieve $O(m)$ query time for constant alphabets.

Fischer and Heun proposed a succinct *Range-Minimum-Query (RMQ)* data-structure built on top of the LCP array in order to replace the child-table [12], yielding a more space efficient ESA.

ALGORITHM 1: ESA query algorithm [3, 12]

Input: Pattern $P = p_0 p_1 \dots p_{m-1}$
Output: interval of P in SA

```

1  $[l, r] \leftarrow [0, n]$ 
2  $c \leftarrow 0, \text{ found} \leftarrow \text{true}$ 
3 while found  $\wedge c < m \wedge l < r$  do
4    $[l, r] \leftarrow \text{getChild}(l, r, P[c])$ 
5   if  $[l, r] == \emptyset$  then
6     return 'not found'
7   end
8    $\ell \leftarrow \min_{i \in [l+1, r]} \text{LCP}[i], m$ 
9   found  $\leftarrow (S[SA[l] + c \dots SA[l] + \ell - 1] == P[c \dots \ell - 1])$ 
10   $c \leftarrow \ell$ 
11 end
```

The pattern search algorithms for (enhanced) suffix arrays are also called *forward-search*, as they search the array structures by starting with the first character of the pattern P and then iteratively matching more and more characters, while narrowing a search interval $[l, r]$ in the suffix and LCP arrays. Algorithm 1 illustrates the top-down query algorithms as performed by both Abouelhoda et al. [3], and Fischer and Heun [12]. The search starts from the root represented as the interval $[0, n]$ and $c = 0$ matched characters. In each iteration of this search, the current query interval $[l, r]$ corresponds to those suffixes $SA[l], \dots, SA[r-1]$ in the suffix array which have a c -length common prefix with P .

ALGORITHM 2: getChild function from Fischer and Heun [12]

```

1 Function getChild( $l, r, a$ )
2    $i \leftarrow \text{RMQLCP}(l+1, r-1)$ 
3    $\ell \leftarrow \text{LCP}[i]$ 
4   repeat
5     if  $S[SA[l] + \ell] == a$  then
6       return  $[l, i-1]$ 
7     end
8      $l \leftarrow i$ 
9      $i \leftarrow \text{RMQLCP}(l+1, r-1)$ 
10  until  $l = r \vee \text{LCP}[i] > \ell;$ 
11  if  $S[SA[l] + \ell] == a$  then
12    return  $[l, r]$ 
13  else
14    return  $\emptyset$ 
15 end
```

Each iteration of the search progresses by finding the child interval in $[l, r]$ which corresponds to the character $P[c]$. This getChild() function differs between the two approaches. The *child-table* of Abouelhoda et al. contains pre-computed child-intervals. Finding the correct child-interval thus resorts to scanning the *child-table* for a match of the character $P[c]$. Fischer and Heun iteratively perform the Range-Minimum-Query on the LCP array up to $|\Sigma|$ times (see Algorithm 2 above). Both methods access $S[SA[i] + c]$ for multiple positions i (up to $|\Sigma|$ many) while searching for the child interval corresponding to the character $P[c]$. Due to the nature of the suffix array, these are random accesses into the string S . Furthermore, $\ell - c$ characters of the pattern are compared to the suffix starting at $S[SA[l] + c \dots]$ in each iteration (Algorithm 1 Line 9) - another random access read into the string.

In a distributed representation of the string and index arrays, these query algorithms would require communication in every step of the search - making querying in the distributed representation prohibitively expensive. Next, we propose a novel distributed data structure and querying algorithm which avoids random accesses.

3.2 DESA: Local Index Querying

The local indices of our *Distributed Enhanced Suffix Array (DESA)* consist of the distributed suffix array SA and LCP array, a *Range-Minimum-Query (RMQ)* data structure, and an additional character array L_c , which is formally defined in the next section.

For efficient top-down traversal and pattern matching in the distributed representation, we need to remove the requirement for randomly accessing the string S in every step of the forward-search algorithm.

Instead of matching the full pattern to the suffixes during the top-down traversal, we match only those key-characters which correspond to the branching characters of the corresponding suffix tree, i.e, the first character on each edge along the path being traversed. By ignoring other characters during the traversal, we initially allow a false positive match, which is later verified by performing a string comparison after the top-down traversal is completed. Splitting the search algorithm into these two stages allows us to pre-compute the characters required in each traversal decision into locally available arrays.

We propose a query algorithm (Algorithm 3), a pre-computed character array, and a new getChild() function, which allow processing a query within a given search interval $[l, r]$ requiring no random accesses outside of this index range. Thus, if the query interval $[l, r]$ lies fully within a processor, it can be processed locally.

We modify the ESA query Algorithm 1 of Fischer and Heun [12] as follows. First, we remove the *found* check, which compares $\ell - c$ additional characters in each iteration (Algorithm 1, Line 9), and move it outside of the loop. Effectively, we are allowing false-positives during the iterative querying, by comparing only those characters which correspond to a branch in the suffix tree. If the querying algorithm finds a SA range for the pattern with $l < r$, we merely have to inspect the pattern against a single suffix of the range (e.g. $SA[l]$) in order to determine if the pattern matches. The resulting Algorithm 3 illustrates the steps of the forward-search algorithm.

ALGORITHM 3: Global view of DESA query algorithm

Input: Pattern $P = p_0p_1 \dots p_{m-1}$
Output: interval of P in SA

```

1  $c \leftarrow 0$ 
2 while  $c < m \wedge l < r$  do
3    $[l, r] \leftarrow \text{getChild}(l, r, P[c])$ 
4    $\ell \leftarrow \min(LCP[RMQ(l+1, r-1)], m)$ 
5    $c \leftarrow \ell$ 
6 end
7 if  $l < r$  then
8    $\text{found} \leftarrow (S[SA[l] \dots (SA[l] + m - 1)] == P[0 \dots (m - 1)])$ 
9 end
```

Next, we propose to eliminate the random access reads inside the getChild() function and replace them by pre-computed local lookups. We observe that the character accessed in Algorithm 2, Line 5: $S[SA[l] + \ell]$ is identical to $S[SA[j] + \ell]$ for all $j = l, l+1, \dots, i-1$. This is the case, because all these suffixes $SA[j]$ share a common longest prefix larger than ℓ : $LCP[j] \geq LCP[i] > \ell$. We can thus replace Line 5 in Algorithm 2 by $S[SA[i-1] + LCP[i]]$. This is still a random access into the string S . However, for any given i , this is always the same character and position in the string S .

Using this property, we can define a character array which contains for each position i :

$$L_c[i] = S[SA[i-1] + LCP[i]]$$

The corresponding location in the LCP array $LCP[i]$ sits between two subtrees, each of which has a string depth $> LCP[i]$. All suffixes in the left of the two subtrees share $L_c[i]$ as the branching character.

Using the L_c character array, we arrive at the getChild() function as shown in Algorithm 4. Note that for a subtree with u children, there are $u-1$ LCP positions corresponding to that node. Thus, only $u-1$ characters are explicitly saved for that internal node. The last sub-tree/child-interval does not have its branching character stored. If the function does not find a character match in the first $u-1$ child intervals, we always continue the search in the last interval. The matching algorithm (Algorithm 3) still works as intended, because we allow false-positives during the traversal itself, and check against the underlying string at the end.

ALGORITHM 4: DESA getChild function

```

1 Function getChild( $l, r, a$ )
2    $i \leftarrow RMQLCP(l+1, r-1)$ 
3    $\ell \leftarrow LCP[i]$ 
4   repeat
5     if  $L_c[i] == a$  then
6       return  $[l, i-1]$ 
7     end
8      $l \leftarrow i$ 
9      $i \leftarrow RMQLCP(l+1, r-1)$ 
10  until  $l = r \vee LCP[i] > \ell$ ;
11  // return last child interval
12  return  $[l, r]$ 
13 end
```

For any given query range $[l, r]$, this algorithm accesses data only within the index range $[l, r]$, avoiding any non local accesses and thus allows its efficient use in distributed memory systems.

4 TOP LEVEL INDEX AND DISTRIBUTION OF SUBTREES

Our distributed suffix array and LCP array construction algorithms [14] return these arrays block distributed across processors. This type of distribution is useful during construction, but not so for querying. Any subtree might arbitrarily be split across processor boundaries - greatly complicating distributed querying.

For an efficient distributed index, we instead need to split the (virtual) suffix tree into a shared *top-level index*, and separate subtrees, where subtrees are distributed onto processors such that each subtree is fully contained within a processor, enabling it to be queried locally.

The *top-level index* (TLI) is used to match the first few characters of a pattern P , and then points to whichever processor contains the subtree required for continuing the forward-search. The top-level index is kept as a copy on every processor and thus cannot be allowed to exceed the size of $O(n/p)$ to achieve true memory scaling. Figure 2 illustrates this concept.

Here, we introduce two different variants/implementations of a top-level index with different partitioning schemes for subtrees: a simple Top-Level Lookup Table (TLLT) or a Top-Level Dynamic Trie (TLDT).

4.1 Top-Level Lookup Table

Using a Top-Level q -mer Lookup Table can accelerate top-down traversal by skipping an initial number of iterations. For each possible q -mer (length q substring) over the alphabet Σ , the suffix array index range is pre-computed and saved into this table. This type of lookup table of size $|\Sigma|^q$ has been proposed by Manber and Myers as a *bucket array* to be used for speeding up the binary search in their suffix array query algorithm [31].

For our distributed suffix array, the lookup table has another important use. Each subrange corresponds to a subtree of the associated suffix tree. If the subranges are small enough, they can be redistributed between processors so that each subrange is fully contained within a processor. Then, a query of the distributed enhanced suffix array can be fulfilled by a single lookup within the

top-level lookup table and solely local processing within the processor containing the corresponding target subrange.

We require that the size of the lookup table $|\Sigma|^q$ does not exceed our design constraint that each processor use at most $O(\frac{n}{p})$ local memory. At the same time, q should be chosen to be sufficiently large, such that the index ranges (subtrees) are expected to be small enough to be fully contained within a processor and not create a notable load-imbalance. This yields very rough bounds for useful choices of q to: $p < |\Sigma|^q < \frac{n}{p}$, the exact value of which depends on the application and can be chosen as a tuning parameter, with a trade-off between expected load imbalance, extra memory usage, and faster querying.

The Top-Level Lookup Table can be easily and efficiently constructed by a single parallel scan of the input. Each processor generates all q -mers with a sliding window approach for its $O(n/p)$ section of the input string and creates a frequency histogram of q -mers. A parallel reduction (allreduce), followed by a prefix sum over the histogram, then creates the required lookup table on each processor.

This q -mer lookup table based approach is commonly utilized in computational biology for DNA and protein sequences. Modeled as strings, these exhibit a relatively small alphabet size and relatively even q -mer distribution. However, for datasets with large alphabet sizes and very skewed q -mer distributions, this approach does not work well. Table 1 illustrates the high load imbalance resulting from the use of the TLLT even for very large q -mer tables.

To achieve a better load balance without further increasing the space complexity, a more dynamic partitioning and distribution scheme is required.

4.2 Top-Level Dynamic Trie

The Top-Level Dynamic Trie (TLDT) is a top level index in the form of a trie, where each leaf of the trie points to a subtree of the (virtual) suffix tree. We construct the TLDT such that each subtree corresponding to a leaf in the TLDT has size $\leq \frac{n}{cp}$, while the size of the subtree under its parent is $> \frac{n}{cp}$. Here, c is a parameter controlling for load imbalance.

We construct the TLDT given the equally block distributed LCP and L_c arrays. The main challenge in creating the TLDT is to efficiently identify the largest subtrees not exceeding the threshold size. Every LCP position corresponds to a subtree (see Figure 1). For every position i in the distributed LCP array, we find the position of the closest smaller element to its left and the closest to its right. This yields the subtree interval (and thus also its subtree size) of the parent of the subtree corresponding to $LCP[i]$. Figure 1 visualizes this property. The distributed *All-Nearest-Smaller-Values* (ANSV) algorithm from [15] can be used to find these positions for all nodes simultaneously in parallel in time $O(\frac{n}{p} + p)$ (for details, please see that paper). Once we have the parent range $(L[i], R[i])$ for every LCP position $LCP[i]$, we then mark all positions i for which the parent interval size exceeds the given threshold size. Additionally, we also mark all locations i for which $LCP[i] = 0$, i.e., where no true parent range exists. This in-effect, marks all nodes of the tree for which the corresponding subtree sizes are larger than the threshold, as well as all nodes which are the immediate children of such nodes.

input	σ	q	TLLT		TLDT	
			size	L	size	L
chr1	4	8	131	71%	5.8	0.7%
dna	16	5	262	582%	5.8	0.5%
proteins	27	5	262	4%	19	0.5%
english	239	3	131	1639%	29	0.5%
sources	230	3	131	3060%	39	1.9%
dblp.xml	97	3	16	2577%	22	0.4%

Table 1: Comparisons of load balance achieved for different data sets (the Pizza & Chili corpus [11] and chromosome 1 of the human genome) for a partitioning onto $p = 1024$ processors using a large TLLT or the TLDT. The size is the total memory usage of the top level index in MB, and L denotes the relative load balance achieved, where $L = 0\%$ denotes a perfect load balance.

The *marked* locations are kept track of by storing their indices into a new offset array *OFF* and each processor creates new smaller LCP and L_c arrays consisting of all locally marked positions. Then all three arrays are collected on all processors using an all-gather operation, where they are used to create the top level trie using the same strategy as the local subtree index introduced in Section 3.

4.3 Partitioning

Once the Top-Level Index is created using the equally block distributed arrays (SA, LCP , and L_c), the arrays need to be re-partitioned such that all subtrees are contained fully within a processor. We use a simple, order preserving 1D partitioning via recursive bisection on the subtree sizes of the top level index in order to repartition the subtrees.

Table 1 compares the TLLT and TLDT approaches with regard to their total memory size and load balance achieved for partitioning onto $p = 1024$ processors. The results clearly show that the TLDT is a much superior strategy for achieving good load balance for all data sets. The TLDT is smaller in size in almost all cases and achieves almost perfect load balancing across all input data sets. For structured input texts (language or sources), the load imbalance using the TLLT is dramatic, where some processors end up having up to $30\times$ more elements than the optimal $\frac{n}{p}$.

5 DISTRIBUTED QUERYING

Taking all parts together, it is now relatively straightforward to formulate the distributed query algorithms. We will first describe a distributed query for a single pattern P , assuming the pattern arrives at one of the processors. Then, we will describe a bulk parallel algorithm, where each processor has a set of query patterns, and all queries are processed simultaneously.

5.1 Distributed Locate

Algorithm 5 shows at a high level the process of querying the Distributed ESA for a single pattern P . Given a pattern $P = p_0, \dots, p_{m-1}$ of length m , the pattern search proceeds in three phases: First, the Top-Level Index (TLLT or TLDT) is queried to determine the subtree/subrange in which the pattern might occur, and which processor $\rho \in \{0, \dots, p-1\}$ the subtree is located on. The pattern is then sent to processor ρ , where the local index is queried. The

ALGORITHM 5: Distributed Locate

```

1 Function distributed_locate( $P$ )
2   // Phase I: send  $P$  to correct processor
3    $\rho \leftarrow \text{TLI.lookup\_proc}(P)$ 
4   send( $P$ , to= $\rho$ )
5   // Phase II: locally query for possible match
6   if  $\rho = \text{comm.rank}()$  then
7      $[l, r] \leftarrow \text{idx.locate\_possible}(P)$ 
8     send( $(P, l, r, \text{SA}[l])$ , to= $\text{rank\_of}(\text{SA}[l])$ )
9   end
10  // Phase III: rule out false positive
11  if  $P, pos \leftarrow \text{recv}()$  then
12    match  $\leftarrow \text{strcmp}(P, S[pos \dots])$ 
13    if no match then
14      return  $\emptyset$ 
15    end
16    return  $P, [l, r]$ 
17  end
18 end

```

query of the local index returns a *possible* result range $[l, r]$. The pattern P is guaranteed to appear in this result range, or not appear in S at all. The false positive result in the latter case has to be ruled out in the third phase of the query algorithm. To do so, we compare the pattern with one of the suffixes in the result range $[l, r]$, specifically, we perform a string comparison between P and $S[\text{SA}[l] \dots]$. Since the string S is still *equally block distributed*, we send the pattern to the processor which contains the string segment starting at $\text{SA}[l]$. There, a single string comparison yields the required result. In some cases $\text{SA}[l] + m$ might not be located on the same processor as $\text{SA}[l]$. In this case, the pattern can be split up and send to the corresponding processors containing the segments. These are then also compared locally and the results combined. The final query result can then either be output to a file or sent back to the originating processor if further processing is required.

Complexity. The cost for Sending a pattern is modeled as $O(\tau + m\mu)$, where τ models the latency, μ the inverse bandwidth, and $m = |P|$. The local query is executed in $O(m)$ time. Sending the pattern and range information for ruling out false positives again costs $O(\tau + m\mu)$. Finally, checking the pattern against the string requires linear $O(m)$ time.

5.2 Distributed Bulk-Locate

The algorithm for bulk-parallel querying of many patterns simultaneously is shown in Algorithm 6. Each processor $\rho \in \{0, \dots, p-1\}$ receives a separate list of patterns $P_1, \dots, P_z$, not necessarily the same number on each processor. The TLI is used to determine for each pattern P_i , which processor can process the query. The target processor $T[i]$ for each pattern P_i is saved, the patterns bucketed/sorted by target processor index and then an all-to-all communication is used to exchange all patterns between all processors such that each processor will receive those patterns which it can process locally.

In the second phase, each processor locally queries its received patterns Q and records for each pattern $Q[i]$ the potential solution range $[L[i], R[i]]$.

ALGORITHM 6: Distributed Bulk-Locate

```

1 Function distributed_bulk_locate( $\{P_1, \dots, P_z\}$ )
2   // Phase I: lookup where each pattern will be located
3    $T \leftarrow \text{new array}(\text{size} = z)$ 
4   for  $i = 1, \dots, z$  do
5      $T[i] \leftarrow \text{TLI.lookup\_proc}(P_i)$ 
6   end
7   // all-to-all exchange of patterns for querying
8    $[Q_1, \dots, Q_y] \leftarrow \text{alltoall}([P_1, \dots, P_z], \text{to}=[T_1, \dots, T_z])$ 
9
10  // Phase II: local queries
11   $L, R \leftarrow \text{new arrays}(\text{size} = z)$ 
12   $T \leftarrow \text{new array}(\text{size} = y)$ 
13  for  $i = 1, \dots, y$  do
14    // locally query for the pattern
15     $L[i], R[i] \leftarrow \text{idx.locate\_possible}(Q_i)$ 
16    // determine which processor contains the string data for
17    // suffix  $\text{SA}[L[i]]$ 
18     $T[i] \leftarrow \text{rank\_of}(S, \text{SA}[L[i]])$ 
19  end
20  // all-to-all exchange
21   $Q, L, R, pos \leftarrow \text{alltoall}([\dots, (Q_i, L[i], R[i], \text{SA}[L[i]]), \dots], \text{to}=T)$ 
22  send( $P, \text{SA}[l]$ , to= $\text{rank\_of}(\text{SA}[l])$ )
23
24  // Phase III: rule out false positives
25  for  $i = 1, \dots, |Q|$  do
26    match  $\leftarrow \text{strcmp}(Q_i, S[pos[i] \dots])$ 
27    if no match then
28       $L[i], R[i] \leftarrow \emptyset$ 
29    end
30  end
31  // return the result SA range for every query
32  return  $Q, L, R$ 
33 end

```

Next, false positive matches need to be excluded. For this, the patterns are exchanged via an all-to-all communication. Each pattern $Q[i]$ with solution range $[L[i], R[i]]$ is sent to the processor owning the section of the string S which contains $\text{SA}[L[i]]$. Once received, the pattern can be compared with the string S at the suffix location. If the comparison is not successful, the solution range is replaced by the empty range $\emptyset$. Finally, the result can be saved or returned to the originating processor.

6 DISTRIBUTED CONSTRUCTION

Given a distributed input string S , the construction of the DESA has multiple steps. First, we need to construct the SA and LCP arrays for the string S . We do this by using our distributed construction algorithm [14]. This yields the SA and LCP arrays in (equally) block distributed form. We modify our algorithm to also construct the L_c array simultaneously to the LCP array with no increase in runtime complexity. We describe this approach in section 6.1 below. We note that any parallel or distributed algorithm for SA and LCP construction can be used to construct these arrays, however, this may require a different approach for efficiently constructing the L_c array. Next, we construct the top-level index and then partition

and redistribute the SA , LCP , and L_c arrays accordingly. Finally, we construct the RMQ locally over the redistributed local parts of the LCP array, completing the construction of the DESA.

6.1 Efficient Construction of L_c

To efficiently construct the L_c array, we modify our LCP construction algorithm of [14], which is a parallel, distributed memory adaptation of a similar LCP construction algorithm by Manber and Myers [31].

In [14], the LCP array is constructed during the prefix-doubling algorithm for suffix array construction. Assuming all suffixes are sorted with respect to their h -prefix (by the first h characters of each suffixes), one iteration of prefix-doubling will sort all suffixes with respect their $2h$ -prefix. In this step, a group of suffixes sharing a common h -prefix, called an h -group, gets split into multiple adjacent $2h$ -groups. Two suffixes from the same h -group, but in adjacent $2h$ -groups share a common prefix of at least h but less than $2h$ characters.

After the prefix-doubling, the LCP array can be set at these boundaries between $2h$ groups, to a value $h \leq LCP[i] < 2h$.

Let $Suf(SA[i-1])$ and $Suf(SA[i])$ be two adjacent suffixes of the same h -group but different $2h$ -groups. The LCP value at i is determined at this location using a Range-Minimum-Query as: $LCP[i] = h + LCP[RMQLCP(a, b)]$, where the query range given by a and b is determined based on the current ranks (positions in the current sorted order) for the suffixes $Suf(SA[i-1] + h)$ and $Suf(SA[i] + h)$, i.e., by a and b for which $Suf(SA[a]) = Suf(SA[i-1] + h)$ and $Suf(SA[b]) = Suf(SA[i] + h)$. We assume that RMQ returns the leftmost minimum position in the range. For a detailed description of this algorithm, we refer to [14].

The L_c character array is defined as:

$$L_c[i] = S[SA[i-1] + LCP[i]]$$

Two adjacent suffixes $Suf(SA[i-1])$ and $Suf(SA[i])$ share $\ell = LCP[i]$ characters. The ℓ^{th} character differs, and $L_c[i]$ contains this ℓ^{th} character of the left ($SA[i-1]$) suffix.

We propose a construction algorithm for L_c as a modification of the SA and LCP construction algorithm of [14]: For the new $2h$ -group boundaries, where the LCP is set to

$$LCP[i] = h + LCP[RMQLCP(a, b)],$$

we also set

$$L_c[i] = L_c[RMQLCP(a, b)]$$

CLAIM 6.1. *When $LCP[i]$ is set as described above, it holds that:*

$$L_c[RMQLCP(a, b)] = S[SA[i-1] + LCP[i]]$$

where the left-hand side has been set in a previous iteration.

PROOF. Consider the adjacent suffixes $Suf(SA[i-1])$ and $Suf(SA[i])$. Let $q = RMQLCP(a, b)$ be the position of the leftmost minimum inside the LCP array between positions a and b . Since $LCP[q] < h$, $LCP[q]$ has been set in a previous iteration, and thus $L_c[q]$ has been set as well.

The value of $LCP[i]$ is determined based on the current ranks for the suffixes $Suf(SA[i-1] + h) = Suf(SA[a])$ and $Suf(SA[i] + h) = Suf(SA[b])$, which share exactly $LCP[i] - h = LCP[q]$ characters.

Since we know that $LCP[i] = h + LCP[q]$ and $SA[i-1] + h = SA[a]$, we have:

$$SA[i-1] + LCP[i] = SA[i-1] + h + LCP[q] = SA[a] + LCP[q]$$

and therefore:

$$L_c[i] = S[SA[i-1] + LCP[i]] = S[SA[a] + LCP[q]] \quad (1)$$

To finish the proof, we then have to show that $L_c[i] = L_c[q]$. This is the case iff $S[SA[q-1] + LCP[q]] = S[SA[a] + LCP[q]]$, i.e., if suffixes $SA[a]$ and $SA[q]$ share a common prefix of size larger than $LCP[q]$. By our assumption on the RMQ, q is the leftmost minimum and we have $LCP[j] > LCP[q]$ for all $a \leq j < q$. Therefore, all suffixes between a and $q-1$ must share at least $LCP[q] + 1$ common characters in their prefix. $\square$

6.2 Complexity

The run-time complexity construction of the DESA is dominated by the SA and LCP construction, which has a worst-case complexity of $O(\log(n)T_{\text{sort}}(n, p))$ [14], where $T_{\text{sort}}(n, p)$ is the parallel complexity to sort n elements on p processors.

7 EXPERIMENTS AND RESULTS

We implemented the algorithms described above using MPI and C++11, and incorporated them into our distributed string algorithms framework [14, 15]. The code is available as open source on GitHub¹.

To ensure our parallelization efficiency does not stem from needless injection of redundancy, we demonstrate our performance on a single core against state-of-the-art sequential algorithms and codes, at the scale of problems which they can solve. We ran such sequential experiments on *TheMachine*, a single socket Intel Core i7-4770 system with 16 GB main memory. For these, we compiled our code with gcc version 7.3.0 and with optimization flags `-O3 -march=native`.

We ran scaling experiments on *Stampede2*, using up to 32 *SKX Compute Nodes*. Each node has two sockets of *Intel Xeon Platinum 8160* ("Skylake") processors, for a total of 48 cores per node. Each node is equipped with 192GB RAM main memory and nodes are interconnected by a *100Gb/sec Intel Omni-Path (OPA)* network. For the experimental evaluation we used up to 32 nodes, for a total of up to 1536 cores. All reported runtimes are averaged over multiple executions.

Since our work is motivated by applications in computational biology, we use the human genome as an input in order to evaluate performance and scalability. The human genome (*H.sapiens*) is commonly used as a benchmark dataset for index construction [2, 7, 14, 16–18, 35] and consists of approximately 3 billion nucleotides (A, C, T, and G) with an alphabet size of 4. Specifically, we use the reference genome from the 1000 Genomes Project [1] version GRCh37.

Unfortunately, it is not feasible to compare our distributed querying implementation against the few previous approaches at scale.

¹github.com/patfflick/psac

	dna	proteins	english	sources	dblp.xml
sa_index	2.8	3.0	3.2	2.9	3.3
esa_index	7.6	11.2	28.0	30.4	26.6
desa_index	6.4	9.7	19.1	20.0	15.7
desa_tl_index	6.0	5.8	14.5	14.7	10.1
sdsl::csa_wt	6.3	13.7	15.1	19.9	18.8
sdsl::csa_sada	74.9	72.2	65.9	94.1	97.9

Table 2: Average runtime per query in μs for sequential locate/count of 1000 randomly drawn, 20 character long patterns for different inputs of the 200MB Pizza & Chili corpus.

No implementation of the approach by Arroyuelo et al. [5] is available, and after many attempts, we could not get *DPT*, the implementation by Fischer et al. [13], to work on large inputs. Therefore, we compare our implementation against *DPT* [13] only on a smaller scale using chromosome 1 (*chr1*) of the human genome (225 million nucleotides), and independently study the scalability of our approach using the entire human genome.

7.1 Sequential querying

First, we compare a sequential DESA implementation against other sequential approaches. We implement the following data-structures and query algorithms. Our implementations for the ESA, DESA, and DESA with lookup table share most of their underlying construction and querying code - allowing for a fair comparison between these approaches.

- *sa_index* (*SA*, *S*): implements binary search using the suffix array and string *S*, having query time complexity $O(m \log(n))$.
- *esa_index* (*SA*, *LCP*, *RMQ(LCP)*, *S*): ESA index using the suffix array, LCP array, string *S*, and a succinct RMQ implementation. The query complexity is $O(\sigma m)$.
- *desa_index* (*SA*, *LCP*, *RMQ(LCP)*, *L_c*): (sequential) DESA index. Uses *L_c* instead of accesses to *S*, and performs string comparison at the end to rule out false-positives. Otherwise, same implementation as ESA above.
- *desa_tl_index* (*SA*, *LCP*, *RMQ(LCP)*, *L_c*, *TL*): (sequential) DESA index with a *q*-mer Top-Level Lookup table.

Furthermore, we compare the query time of our implementations against two implementations of compressed suffix arrays of the *Succinct Data Structure Library 2.0* (*sdsl*) [19]. We use its current version 2.1.1 from GitHub².

We ran query benchmarks using a set of different inputs from the *Pizza & Chili* Corpus provided by Ferragina et al. [9] and available online³. Same as in the *sdsl* benchmarks by Gog and Petri [19], we use the 200MB prefix of the input types *dna*, *proteins*, *english*, *sources*, and *dblp.xml*.

For our benchmark queries, we generate 1000 random substrings of length 20 from the input file. We then construct and query the different indexes by running the generated queries 10,000 times. We report the achieved average time per query in Table 2. The times reported are given in microseconds (μs), averaged per query. For *sdsl* we report the time required for the count operation - which reports the number of matches. The *locate* function in *sdsl* reports the occurrences, but takes significantly longer to run. The results

Figure 3: Total construction time of DESA and DPT [13] for *chr1* (including the time required to construct the underlying SA and LCP arrays [14]), scaling from 1 to 384 cores (8 nodes) of *Stampede2*.

show that querying the suffix array alone is multiple times faster than any of the other indexes - however due to the dependence on *n* and the many random accesses required, this method is not applicable to distributed memory querying. Among the enhanced indexes, our sequential DESA index shows the best performance for all categories of input. This demonstrates that our scaling results are measured against the best possible sequential performance.

7.2 Comparison with DPT

7.2.1 Construction. First, we compare the construction time of our distributed *DESA* implementation with *DPT*⁴ [13]. Both construction algorithms use our earlier suffix array and LCP array construction implementation [14], and build the respective distributed index given the SA and LCP arrays. To fairly compare the total construction time, we report the time to construct the entire index from the string, i.e., including the time required to construct the SA and LCP arrays. As the code of *DPT* fails to work on the entire human genome, we report the runtime for constructing the distributed indices while scaling from one to 384 cores (8 nodes) of *Stampede2*.

The results illustrate that the index construction algorithms of both approaches scale well. However, the *DPT* construction requires significant additional time on-top of the SA and LCP array construction, whereas the *DESA* construction uses very little extra time. Given the SA and LCP arrays, the *DESA* construction is between 14.1× to 25.7× faster compared to the *DPT* construction. When comparing the total construction time (including the time for the SA and LCP construction), the *DESA* construction is still between 2.0× to 2.6× faster than *DPT* (see Figure 3).

7.2.2 Querying. Next, we compare the query times using *DESA* and *DPT* for *chr1*. A key limitation of the approach of *DPT* [13] is the constant and fixed maximum query length. Their implementation uses 30 as the maximum length. Therefore, we compare the query performance using as queries 10 million randomly sampled substrings of *chr1* each of length 30. Figure 4 shows the query rate by number of cores, given a fixed total number of 10 million queries. The figure on the left is a *zoomed* version, showing the

²sdsl-lite version 2.1.1: github.com/simongog/sdsl-lite

³<http://pizzachili.dcc.uchile.cl/texts.html>

⁴github.com/kurpicz/dpt

Figure 4: Query rate of DESA and DPT, executing 10 million queries total on *chr1*, scaling from 1 to 384 cores (8 nodes). The left figure shows a zoomed view of the data, up to 48 cores, i.e., all cores of a single node.

Figure 5: Weak scaling for the entire human genome, 10 million queries per core, each of length 100. Run on 96-1536 cores (2-32 nodes) of Stampede2.

performance on a single node, whereas the figure to the right shows the performance for up to 384 cores (8 nodes) of Stampede2.

The results show that our approach has slight edge over DPT in terms of query speed. The DESA query algorithm is between 13 to 29% faster while scaling from one to 384 cores. This is despite our algorithm using a more sophisticated top-level indexing scheme, and returning query results to their original processors - whereas DPT terminates as soon as query answers are found locally without returning query results to their origin.

DESA achieves a speedup of 36.4 $\times$ within a single 48-core node, an additional factor of 7.4 $\times$ speedup when scaling from 1 to 8 nodes (using 48 cores each), for a total speedup of 270.9 $\times$ when going from 1 to 384 cores on this relatively small dataset.

7.3 Scaling

Next, we study the scalability of DESA on the entire human genome using up to 1536 cores of Stampede2. Figure 5 visualizes the weak scaling for a fixed number of 10 million queries per compute core, and further shows the breakdown of the total time into local compute and communication. The total time required slightly increases with the number of cores used: from 96 cores (2 nodes) to 1536 cores (32 nodes), the total time required increases by 19.9%. However, most of this increase is due to an increase in communication time by 1.9 $\times$: entirely due to the all-to-all communications required to

Figure 6: Runtime contributions of the different components of DESA construction algorithm.

exchange queries. The local query time slightly increases by 8% when increasing the number of nodes by 16 $\times$. This local work increase is due to the increased size of the dynamic top level trie, which is dynamically sized based on the number of cores used - in order to maintain a good load-balance.

7.4 Construction

Our earlier paper experimentally compares the the SA and LCP construction against other approaches and demonstrates its superior performance [14]. Therefore, here, we will not further compare against other SA and LCP construction algorithms. The construction of the SA and LCP arrays make up close to the entire runtime of the DESA construction (see also Figure 3). Figure 6 shows where the remaining time of the parallel DESA construction is spent. The runtime contributions are given as a percent increase compared to the time required to construct the SA and LCP arrays using [14]. These results show that the extra time required is less than 8% on 96 cores (2 nodes) and that this percentage shrinks with increasing core-count. The parallel construction of the Top-Level Dynamic Trie (TLDT) and the subsequent communication for redistributing elements require approximately 2% extra time each.

Furthermore, we compare the runtime of our L_c construction algorithm against the time required to construct the SA and LCP Array without L_c (the algorithm and implementation of [14]), as well as a naive construction of the L_c array, by implementing a bulk-query using all-to-all communication to directly read the underlying string for each $L_c[i]$ at $S[SA[i-1] + LCP[i]]$.

The results are shown in Figure 6. The L_c construction is at the bottom of the stacked plot; it is not even visible because it does not exceed 0.007% total contribution. Naive L_c construction, on the other hand, adds as much as 11% to the total runtime - more than the entire DESA construction.

The work in [14] previously illustrated fast distributed construction of the suffix array and how constructing the LCP array alongside adds just a small fraction to the total runtime. Here, we have shown that the additional data structures required to create an efficiently queryable distributed enhanced suffix array can be accomplished in little extra time. Constructing the DESA index for the full human genome on 1536 cores (64 nodes) of Stampede2 takes just 4.3 seconds.

8 CONCLUSION

In this paper, we introduced a novel distributed string index, the Distributed Enhanced Suffix Array (DESA). This distributed data structure allows efficient construction and querying, all while requiring only $O(n/p)$ memory per process and supporting arbitrarily long query patterns. We introduce a novel top level index and its distributed construction algorithm, and showed how it can lead to improved load balancing compared to a naive approach. Further, we presented an efficient distributed-memory parallel algorithm for the construction of the L_c array. We demonstrated the performance of the DESA query algorithms by comparing against other sequential approaches, and demonstrated scalability to over 1500 cores. Our results constitute an advancement over the state-of-the-art method DPT [13] – providing improved absolute performance and scalability in both construction and querying, while removing limitations of fixed size query lengths and enabling scaling to much larger datasets. Taken together, these results constitute both theoretical and practical improvements of the state-of-the-art in distributed memory algorithms for string indexing and searching. Lastly, the implementation of all these algorithms is integrated into our open-source library for distributed string indexing⁵, which include the implementations for SA and LCP construction [14, 15].

ACKNOWLEDGMENTS

This research is supported in part by the National Science Foundation under IIS-1416259 and CCF-1704552.

REFERENCES

- [1] 1000 Genomes Project Consortium and others. 2012. An integrated map of genetic variation from 1,092 human genomes. *Nature* 491, 7422 (2012), 56–65.
- [2] Ahmed Abdelhadi, AH Kandil, and Mohamed Abouelhoda. 2014. Cloud-based parallel suffix array construction based on MPI. In *Biomedical Engineering (MECBME), 2014 Middle East Conference on*. IEEE, 334–337.
- [3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* 2, 1 (2004), 53–86.
- [4] Alberto Apostolico, Costas Iliopoulos, Gad M. Landau, Baruch Schieber, and Uzi Vishkin. 1988. Parallel construction of a suffix tree with applications. *Algorithmica* 3, 1-4 (1988), 347–365.
- [5] Diego Arroyuelo, Carolina Bonacic, Veronica Gil-Costa, Mauricio Marin, and Gonzalo Navarro. 2014. Distributed text search using suffix arrays. *Parallel Comput.* 40, 9 (2014), 471–495.
- [6] Robert S Boyer and J Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10 (1977), 762–772.
- [7] Matteo Comin and Montse Farreras. 2013. Efficient parallel construction of suffix trees for genomes larger than main memory. In *Proceedings of the 20th European MPI Users' Group Meeting*. ACM, 211–216.
- [8] Mrinal Deo and Sean Keely. 2013. Parallel suffix array and least common prefix for the GPU. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 197–206.
- [9] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2009. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)* 13 (2009), 12.
- [10] Paolo Ferragina and Fabrizio Luccio. 1999. String search in coarse-grained parallel computers. *Algorithmica* 24, 3-4 (1999), 177–194.
- [11] Paolo Ferragina and Gonzalo Navarro. 2005. Pizza&Chili Corpus. (2005). <http://pizzachili.dcc.uchile.cl/index.html>
- [12] Johannes Fischer and Volker Heun. 2007. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 459–470.
- [13] Johannes Fischer, Florian Kurpicz, and Peter Sanders. 2017. Engineering a Distributed Full-Text Index. In *2017 Proceedings of the Nineteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 120–134.
- [14] Patrick Flick and Srinivas Aluru. 2015. 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*. ACM, 16.
- [15] Patrick Flick and Srinivas Aluru. 2017. Parallel Construction of Suffix Trees and the All-Nearest-Smaller-Values Problem. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 12–21.
- [16] Natsuhiko Futamura, Srinivas Aluru, and Stefan Kurtz. 2001. Parallel suffix sorting. (2001), 76–81.
- [17] Amol Ghoting and Konstantin Makarychev. 2009. Indexing genomic sequences on the IBM Blue Gene. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 61.
- [18] Amol Ghoting and Konstantin Makarychev. 2009. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 827–840.
- [19] Simon Gog and Matthias Petri. 2014. Optimized succinct data structures for massive data. *Software: Practice and Experience* 44, 11 (2014), 1287–1314.
- [20] Dan Gusfield. 1997. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press.
- [21] Ramesh Hariharan. 1994. Optimal parallel suffix tree construction. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. ACM, 290–299.
- [22] CS Iliopoulos and Wojciech Rytter. 2004. On parallel transformations of suffix arrays into suffix trees. In *15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*. Citeseer.
- [23] Juha Kärkkäinen and Peter Sanders. 2003. Simple linear work suffix array construction. In *Automata, Languages and Programming*. Springer, 943–955.
- [24] Richard M Karp and Michael O Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31, 2 (1987), 249–260.
- [25] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial pattern matching*. Springer, 181–192.
- [26] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing* 6, 2 (1977), 323–350.
- [27] Pang Ko and Srinivas Aluru. 2003. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*. Springer, 200–210.
- [28] Fabian Kulla and Peter Sanders. 2007. Scalable parallel suffix array construction. *Parallel Comput.* 33, 9 (2007), 605–612.
- [29] Julian Labeit, Julian Shun, and Guy E Blelloch. 2017. Parallel lightweight wavelet tree, suffix array and FM-index construction. *Journal of Discrete Algorithms* 43 (2017), 2–17.
- [30] Yunhao Li, Jiahui Jin, Runqun Xiong, and Junzhou Luo. 2017. A Distributed Approach for Constructing Generalized Suffix Tree on Spark by Using Optimized Elastic Range Algorithm. In *Advanced Cloud and Big Data (CBD), 2017 Fifth International Conference on*. IEEE, 117–122.
- [31] Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing* 22, 5 (1993), 935–948.
- [32] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. 2011. ERA: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment* 5, 1 (2011), 49–60.
- [33] Vitaly Osipov. 2012. Parallel suffix array construction for shared memory architectures. In *String Processing and Information Retrieval*. Springer, 379–384.
- [34] Simon J Puglisi, William F Smyth, and Andrew H Turpin. 2007. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)* 39, 2 (2007), 4.
- [35] Julian Shun and Guy E Blelloch. 2014. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing* 1, 1 (2014), 8.
- [36] Esko Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica* 14, 3 (1995), 249–260.
- [37] Leyuan Wang, Sean Baxter, and John D Owens. 2016. Fast parallel skew and prefix-doubling suffix array construction on the GPU. *Concurrency and Computation: Practice and Experience* 28, 12 (2016), 3466–3484.

⁵github.com/patflick/psac

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

For sequential experiments, we ran on {TheMachine}, a single socket Intel Core i7-4770 system with 16 GB main memory. For these, we compiled our code with gcc version 7.3.0 and with optimization flags {-O3 -march=native}.

We ran scaling experiments on {Stampede2}, using up to 32 {SKX Compute Nodes}. Each node has two sockets of {Intel Xeon Platinum 8160 ("Skylake")} processors, for a total of 48 cores per node. Each node is equipped with 192GB RAM main memory and nodes are interconnected by a {100Gb/sec Intel Omni-Path (OPA)} network. For the experimental evaluation we used up to 32 nodes, for a total of up to 1536 cores. All reported runtimes are averaged over multiple executions.

Date of use: we used Stampede2 throughout the months of March - May 2019. The collect_environment.sh script was run on 5/25/2019.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

All code for the introduced data structures and

→ algorithms is available on github:

→ <https://github.com/patflick/psac>

DOI: 10.5281/zenodo.3365719

We put all runtime logs (experiment results) and the

→ analysis script creating all plots for the paper

→ on github as well:

→ <https://github.com/patflick/desa-timing>

DOI: 10.5281/zenodo.3365721

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Stampede2 [SKX nodes: Intel Xeon Platinum 8160 ("Skylake") 2.1GHz nominal, 192GB (2.67GHz) DDR4 RAM]; TheMachine (for sequential benchmarks only)[Intel Core i7-4770 @ 3.4 GHz, 16 GB RAM]

Operating systems and versions: Stampede2 [CentOS Linux release 7.6.1810, kernel 3.10.0-957.5.1.el7.x86_64], TheMachine [Ubuntu 18.04, kernel 4.15.0-47]

Compilers and versions: Stampede2 [icpc (ICC) 18.0.2 20180210], TheMachine [gcc 7.3.0]

Libraries and versions: Stampede2 [Intel(R) MPI Library for Linux* OS, Version 2018 Update 2 Build 20180125 (id: 18157)], TheMachine [sequential only, no MPI]

Input datasets and versions: Data sets used: Pizza & Chili (<http://pizzachili.dcc.uchile.cl/texts.html>), Human Genome g1k v37, chr1 of Human Genome g1k v37

Paper Modifications: In order to be able to compare against the competitor code DPT (<https://github.com/kurpicz/dpt>), we had to fix a few bugs that initially made the code unusable.

My fork of DPT containing the fixes we had to implement: <https://github.com/patflick/dpt> Most of my fixes were already integrated into the upstream branch 'fix-dpt' by the original author: https://github.com/kurpicz/dpt/tree/fix_dpt

The commit history of that branch shows all my changes: https://github.com/kurpicz/dpt/commits/fix_dpt

Output from scripts that gathers execution environment information.

```
=====
Stampede 2
=====

I_MPI_STARTUP_MODE=pmi_shm_netmod
LMOD_FAMILY_COMPILER_VERSION=18.0.2
TACC_GIT_BIN=/opt/apps/git/2.9.0/bin
MKLR00T=/opt/intel/compilers_and_libraries_2018.2.19
↳ 9/linux/mkl
TACC_CMAKE_BIN=/opt/apps/cmake/3.10.2/bin
TACC_IMPI_DIR=/opt/intel/compilers_and_libraries_201
↳ 8.2.199/linux/mpi
SLURM_NTASKS_PER_NODE=48
TACC_INTEL_INC=/opt/intel/compilers_and_libraries_20
↳ 18.2.199/linux/compiler/include/intel64
IPPROOT=/opt/intel/compilers_and_libraries_2018.2.19
↳ 9/linux/ipp
MPICH_HOME=/opt/intel/compilers_and_libraries_2018.2
↳ .199/linux/mpi
TACC_INTEL_DIR=/opt/intel/compilers_and_libraries_20
↳ 18.2.199/linux
TACC_LIBFABRIC_BIN=/opt/apps/libfabric/1.7.0/bin
TACC_VTUNE_DIR=/opt/intel/vtune_amplifier_2018.4.0.5
↳ 74913
I_MPI_FABRICS=shm:ofi
SLURM_JOB_QOS=normal
TACC_IMPI_BIN=/opt/intel/compilers_and_libraries_201
↳ 8.2.199/linux/mpi/intel64/bin
TACC_LIBFABRIC_DIR=/opt/apps/libfabric/1.7.0
SLURM_TOPOLOGY_ADDR_PATTERN=node
XALT_DIR=/opt/apps/xalt/xalt/
```

```

LIBRARY_PATH=/opt/intel/compilers_and_libraries_2018.2.199/linux/daal/./tbb/lib/intel64_lin/gcc4.4:/opt/intel/compilers_and_libraries_2018.2.199/linux/daal/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/tbb/lib/intel64/gcc4.7:/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/compiler/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/ipp/lib/intel64
TACC_LIBFABRIC_INC=/opt/apps/libfabric/1.7.0/include
LD_PRELOAD=/opt/apps/xalt/xalt/lib64/libxalt_init.so
TACC_PYTHON2_INC=/opt/apps/intel18/python2/2.7.15/include
TACC_FAMILY_COMPILER_VERSION=18.0.2
QTDIR=/usr/lib64/qt-3.3
QTINC=/usr/lib64/qt-3.3/include
LMOD_VERSION=7.8.21
I_MPI_JOB_FAST_STARTUP=1
__LMOD_REF_COUNT_LOADEDMODULES=git/2.9.0:1;autotools/1.1:1;xalt/2.6.5:1;TACC:1;libfabric/1.7.0:1;intel/18.0.2:1;impi/18.0.2:1;python2/2.7.15:1;vtune/18.0.2:1;cmake/3.10.2:1
TACC_IMPI_INC=/opt/intel/compilers_and_libraries_2018.2.199/linux/mpi/intel64/include
TACC_VTUNE_BIN=/opt/intel/vtune_amplifier_2018.4.0.5/74913/bin64
TACC_INTEL_BIN=/opt/intel/compilers_and_libraries_2018.2.199/linux/bin/intel64
SLURM_NNODES=1
LD_LIBRARY_PATH=/opt/apps/intel18/python2/2.7.15/lib:/opt/intel/compilers_and_libraries_2018.2.199/linux/mpi/intel64/lib:/opt/intel/debugger_2018/lib:/opt/intel/debugger_2018/iga/lib:/opt/intel/compilers_and_libraries_2018.2.199/linux/daal/./tbb/lib/intel64_lin/gcc4.4:/opt/intel/compilers_and_libraries_2018.2.199/linux/daal/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/tbb/lib/intel64/gcc4.7:/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/compiler/lib/intel64_lin:/opt/intel/compilers_and_libraries_2018.2.199/linux/ipp/lib/intel64:/opt/intel/compilers_and_libraries_2018.2.199/linux/compiler/lib/intel64:/opt/apps/gcc/6.3.0/lib64:/opt/apps/gcc/6.3.0/lib:/opt/apps/libfabric/1.7.0/lib
PSTLROOT=/opt/intel/compilers_and_libraries_2018.2.199/linux/pstl
TACC_NODE_TYPE=skx
SLURM_TACC_NODES=1
TACC_XALT_BIN=/opt/apps/xalt/xalt/bin

```

```

CPATH=/opt/intel/compilers_and_libraries_2018.2.199/linux/pstl/include:/opt/intel/compilers_and_libraries_2018.2.199/linux/daal/include:/opt/intel/compilers_and_libraries_2018.2.199/linux/tbb/include:/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl/include:/opt/intel/compilers_and_libraries_2018.2.199/linux/ipp/include
IFC_BIN=/opt/intel/compilers_and_libraries_2018.2.199/9/linux/bin/intel64
TACC_MKL_LIB=/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl/lib/intel64
TACC_GIT_DIR=/opt/apps/git/2.9.0
TACC_PYTHON2_MAN=/opt/apps/intel18/python2/2.7.15/share/man:/opt/apps/intel18/python2/2.7.15/man
SLURM_COMMAND=sbatch
TACC_PYTHON2_BIN=/opt/apps/intel18/python2/2.7.15/bin
SLURM_NTASKS=48
LMOD_FAMILY_MPI_VERSION=18.0.2
ARCHIVER=ranch.tacc.utexas.edu
PATH=/opt/apps/xalt/xalt/bin:/opt/apps/cmake/3.10.2/bin:/opt/intel/vtune_amplifier_2018.4.0.574913/bin64:/opt/apps/intel18/python2/2.7.15/bin:/opt/apps/intel18/mpi/18.0.2/bin:/opt/intel/compilers_and_libraries_2018.2.199/linux/mpi/intel64/bin:/opt/intel/compilers_and_libraries_2018.2.199/linux/bin/intel64:/opt/apps/gcc/6.3.0/bin:/opt/apps/libfabric/1.7.0/bin:/opt/apps/autotools/1.1/bin:/opt/apps/git/2.9.0/bin:/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/opt/dell/srvadmin/bin:
TACC_SYSTEM=stampede2
I_MPI_OFI_LIBRARY=/opt/apps/libfabric/1.7.0/lib/libfabric.so
SLURM_TASKS_PER_NODE=48
ICC_BIN=/opt/intel/compilers_and_libraries_2018.2.199/9/linux/bin/intel64
TACC_GIT_LIB=/opt/apps/git/2.9.0/lib
TBBROOT=/opt/intel/compilers_and_libraries_2018.2.199/9/linux/tbb
LOADEDMODULES=git/2.9.0:autotools/1.1:xalt/2.6.5:TACC:1;libfabric/1.7.0:intel/18.0.2:impi/18.0.2:vtune/18.0.2:cmake/3.10.2
TACC_VEC_FLAGS=-xCORE-AVX2 -xCORE-AVX512,MIC-AVX512
I_MPI_F90=ifort
SLURM_NPROCS=48
I_MPI_CC=icc
SLURM_CPUS_ON_NODE=96
TACC_DOMAIN=stampede2
TACC_VTUNE_LIB=/opt/intel/vtune_amplifier_2018.4.0.5/74913/lib64
I_MPI_CXX=icpc
SLURM_CLUSTER_NAME=stampede2
SLURM_JOB_CPUS_PER_NODE=96
SLURM_JOB_PARTITION=skx-dev
ICC_LIB=/opt/intel/compilers_and_libraries_2018.2.199/9/linux/compiler/lib/intel64

```

Distributed Enhanced Suffix Arrays: Efficient Algorithms for Construction and Querying

```
TACC_FAMILY_MPI=impi
SLURM_JOB_NUM_NODES=1
COMPILER_PATH=/opt/apps/xalt/xalt/bin
PKG_CONFIG_PATH=/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl/bin/pkgconfig
OMP_NUM_THREADS=1
TACC_IMPI_LIB=/opt/intel/compilers_and_libraries_2018.2.199/linux/mpi/intel64/lib
TACC_FAMILY_MPI_VERSION=18.0.2
TACC_AUTOTOOLS_BIN=/opt/apps/autotools/1.1/bin
TACC_MPI_GETMODE=impi_hydra
SLURM_TACC_NNODES_SET=1
TACC_MKL_DIR=/opt/intel/compilers_and_libraries_2018.2.199/linux/mkl
LMOD_FAMILY_MPI=impi
SLURM_TACC_CORES=48
TACC_CMAKE_DIR=/opt/apps/cmake/3.10.2
I_MPI_ROOT=/opt/intel/compilers_and_libraries_2018.2.199/linux/mpi
```

```
LSB Version:           :core-4.1-amd64:core-4.1-noarch:desktop-4.1-amd64:desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing-4.1-noarch
Distributor ID:        CentOS
Description:            CentOS Linux release 7.6.1810
                        (Core)
Release:                7.6.1810
Codename:               Core
Linux c506-053.stamped2.tacc.utexas.edu
                        3.10.0-957.5.1.el7.x86_64 #1 SMP Fri Feb 1 14:54:57 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
Architecture:          x86_64
CPU op-mode(s):         32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 96
On-line CPU(s) list:    0-95
Thread(s) per core:     2
Core(s) per socket:     24
Socket(s):              2
NUMA node(s):           2
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  85
Model name:             Intel(R) Xeon(R) Platinum 8160
                        CPU @ 2.10GHz
Stepping:               4
CPU MHz:                2100.000
BogoMIPS:               4200.00
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               1024K
L3 cache:               33792K
```

```
NUMA node0 CPU(s):
↳ 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34
↳ 36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66
↳ 68,70,72,74,76,78,80,82,84,86,88,90,92,94
NUMA node1 CPU(s):
↳ 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35
↳ 37,39,41,43,45,47,49,51,53,55,57,59,61,63,65,67
↳ 69,71,73,75,77,79,81,83,85,87,89,91,93,95
Flags:                  fpu vme de pse tsc msr pae mce
                        cx8 apic sep mtrr pge mca cmov pat pse36 clflush
                        dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
                        pdpe1gb rdtscp lm constant_tsc art arch_perfmon
                        pebs bts rep_good nopl xtopology nonstop_tsc
                        aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
                        ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
                        pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
                        tsc_deadline_timer aes xsave avx f16c rdrand
                        lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3
                        intel_ppin intel_pt ssbd mba ibrs ibpb stibp
                        tpr_shadow vnmi flexpriority ept vpid fsgsbase
                        tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid
                        rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap
                        clflushopt clwb avx512cd avx512bw avx512vl
                        xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc
                        cqm_mbm_total cqm_mbm_local dtherm ida arat pln
                        pts pku ospke spec_ctrl intel_stibp flush_l1d
MemTotal:               196438176 kB
MemFree:                191187832 kB
MemAvailable:           190682336 kB
Buffers:                0 kB
Cached:                 65128 kB
SwapCached:             0 kB
Active:                 158468 kB
Inactive:               44244 kB
Active(anon):           137772 kB
Inactive(anon):         19196 kB
Active(file):           20696 kB
Inactive(file):         25048 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                  0 kB
Writeback:              0 kB
AnonPages:              137056 kB
Mapped:                 30020 kB
Shmem:                  19272 kB
Slab:                   2017988 kB
SReclaimable:           510720 kB
SUnreclaim:             1507268 kB
KernelStack:            26272 kB
PageTables:             15128 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:            182687500 kB
```

```

Committed_AS:      526676 kB
VmallocTotal:    34359738367 kB
VmallocUsed:      1826416 kB
VmallocChunk:    34257143908 kB
HardwareCorrupted: 0 kB
AnonHugePages:    12288 kB
CmaTotal:         0 kB
CmaFree:          0 kB
HugePages_Total:  0
HugePages_Free:   0
HugePages_Rsvd:   0
HugePages_Surp:   0
Hugepagesize:     2048 kB
DirectMap4k:      771904 kB
DirectMap2M:     17715200 kB
DirectMap1G:     183500800 kB

```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
sda	8:0	0	223.6G	0	disk	
├─sda1	8:1	0	1M	0	part	
├─sda2	8:2	0	1G	0	part	/boot
└─sda3	8:3	0	222.6G	0	part	
├─rootvg01-lv01	253:0	0	75G	0	lvm	/
├─rootvg01-tmp	253:1	0	143.6G	0	lvm	/tmp
└─rootvg01-var	253:2	0	4G	0	lvm	/var
sdb	8:16	0		0	disk	
loop0	7:0	0		1	loop	
loop1	7:1	0		1	loop	
loop2	7:2	0		1	loop	
loop3	7:3	0		1	loop	
loop4	7:4	0		1	loop	
loop5	7:5	0		1	loop	
loop6	7:6	0		1	loop	
loop7	7:7	0		1	loop	
loop8	7:8	0		1	loop	
loop9	7:9	0		1	loop	
loop10	7:10	0		1	loop	
loop11	7:11	0		1	loop	
loop12	7:12	0		1	loop	
loop13	7:13	0		1	loop	
loop14	7:14	0		1	loop	
loop15	7:15	0		1	loop	
loop16	7:16	0		1	loop	
loop17	7:17	0		1	loop	
loop18	7:18	0		1	loop	
loop19	7:19	0		1	loop	
loop20	7:20	0		1	loop	
loop21	7:21	0		1	loop	
loop22	7:22	0		1	loop	
loop23	7:23	0		1	loop	
loop24	7:24	0		1	loop	
loop25	7:25	0		1	loop	
loop26	7:26	0		1	loop	
loop27	7:27	0		1	loop	
loop28	7:28	0		1	loop	
loop29	7:29	0		1	loop	

```

loop30      7:30  0      1 loop
loop31      7:31  0      1 loop
loop32      7:32  0      1 loop
loop33      7:33  0      1 loop
loop34      7:34  0      1 loop
loop35      7:35  0      1 loop
[0:0:0:0]   disk   Generic  MassStorageClass WS01
↳ /dev/sdb   -
[3:0:0:0]   disk   ATA      MZ7KM240HMHQ0D3  GD53
↳ /dev/sda   240GB
Shell debugging temporarily silenced: export
↳ LMOD_SH_DBG_ON=1 for Lmod's output
Shell debugging restarted

```

```

=====
TheMachine (for sequential benchmarks only):
=====

```

```

Distributor ID:      Ubuntu
Description:          Ubuntu 18.04.2 LTS
Release:              18.04
Codename:             bionic
Linux thedevice 4.15.0-47-generic #50-Ubuntu SMP Wed
↳ Mar 13 10:44:52 UTC 2019 x86_64 x86_64 x86_64
↳ GNU/Linux
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               60
Model name:          Intel(R) Core(TM) i7-4770 CPU @
↳ 3.40GHz
Stepping:            3
CPU MHz:             3233.012
CPU max MHz:         3900.0000
CPU min MHz:         800.0000
BogoMIPS:            6784.44
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            8192K
NUMA node0 CPU(s):   0-7

```

Distributed Enhanced Suffix Arrays: Efficient Algorithms for Construction and Querying

```

Flags:          fpu vme de pse tsc msr pae mce
↳ cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↳ dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
↳ pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
↳ bts rep_good nopl xtopology nonstop_tsc cpuid
↳ aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
↳ vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
↳ sse4_1 sse4_2 x2apic movbe popcnt
↳ tsc_deadline_timer aes xsave avx f16c rdrand
↳ lahf_lm abm cpuid_fault epb invpcid_single pti
↳ ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
↳ ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
↳ erms invpcid xsaveopt dtherm ida arat pln pts
↳ flush_l1d
MemTotal:      16366520 kB
MemFree:       3437728 kB
MemAvailable:  8801808 kB
Buffers:       624856 kB
Cached:       4911152 kB
SwapCached:   232 kB
Active:       9584812 kB
Inactive:     2294820 kB
Active(anon): 6144312 kB
Inactive(anon): 389328 kB
Active(file): 3440500 kB
Inactive(file): 1905492 kB
Unevictable:  544 kB
Mlocked:      544 kB
SwapTotal:    7811068 kB
SwapFree:     7807984 kB
Dirty:        1044 kB
Writeback:    0 kB
AnonPages:    6343668 kB
Mapped:       784008 kB
Shmem:        190980 kB
Slab:         547036 kB
SReclaimable: 354068 kB
SUnreclaim:  192968 kB
KernelStack:  23572 kB
PageTables:   115556 kB
NFS_Unstable: 0 kB
Bounce:       0 kB
WritebackTmp: 0 kB
CommitLimit:  15994328 kB
Committed_AS: 21203564 kB
VmallocTotal: 34359738367 kB
VmallocUsed:   0 kB
VmallocChunk:  0 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
ShmemPmdMapped: 0 kB
CmaTotal:     0 kB
CmaFree:      0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0

```

```

HugePages_Surp: 0
Hugepagesize:   2048 kB
DirectMap4k:    993920 kB
DirectMap2M:   15724544 kB
DirectMap1G:   1048576 kB
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
loop0  7:0    0 140.7M  1 loop
↳ /snap/gnome-3-26-1604/82
loop1  7:1    0  34.6M  1 loop
↳ /snap/gtk-common-themes/818
loop2  7:2    0 140.7M  1 loop
↳ /snap/gnome-3-26-1604/78
loop3  7:3    0 174.4M  1 loop /snap/spotify/31
loop4  7:4    0  14.5M  1 loop /snap/gnome-logs/43
loop5  7:5    0      1 loop
loop6  7:6    0  34.8M  1 loop
↳ /snap/gtk-common-themes/1122
loop7  7:7    0  91.1M  1 loop /snap/core/6531
loop8  7:8    0  14.5M  1 loop /snap/gnome-logs/45
loop9  7:9    0  174M   1 loop /snap/spotify/34
loop10 7:10   0  14.8M  1 loop
↳ /snap/gnome-characters/206
loop11 7:11   0    4M   1 loop
↳ /snap/gnome-calculator/352
loop12 7:12   0 1008K   1 loop /snap/gnome-logs/57
loop13 7:13   0   3.7M  1 loop
↳ /snap/gnome-system-monitor/54
loop14 7:14   0 174.5M  1 loop /snap/spotify/32
loop15 7:15   0   13M   1 loop
↳ /snap/gnome-characters/139
loop16 7:16   0  35.3M  1 loop
↳ /snap/gtk-common-themes/1198
loop17 7:17   0  89.3M  1 loop /snap/core/6673
loop18 7:18   0 143.5M  1 loop
↳ /snap/gnome-3-28-1804/23
loop19 7:19   0   3.7M  1 loop
↳ /snap/gnome-system-monitor/57
loop20 7:20   0  53.7M  1 loop /snap/core18/782
loop21 7:21   0   91M   1 loop /snap/core/6405
loop22 7:22   0   3.7M  1 loop
↳ /snap/gnome-system-monitor/70
loop23 7:23   0      1 loop
loop24 7:24   0 140.7M  1 loop
↳ /snap/gnome-3-26-1604/74
loop25 7:25   0   2.3M   1 loop
↳ /snap/gnome-calculator/260
loop26 7:26   0  14.8M  1 loop
↳ /snap/gnome-characters/254
loop27 7:27   0    4M   1 loop
↳ /snap/gnome-calculator/406
sda    8:0    0 931.5G  0 disk
├─sda1  8:1    0  39.2M  0 part
├─sda2  8:2    0   750M  0 part
└─sda3  8:3    0  93.1G  0 part

```


—sda4	8:4	0	1K	0 part	
—sda5	8:5	0	93.1G	0 part	/
—sda6	8:6	0	7.5G	0 part	[SWAP]
—sda7	8:7	0	465.7G	0 part	/media/data
H/W path	Device	Class	Description		
=====					
		system	Computer		
/0		bus	Motherboard		
/0/0		memory	15GiB System		
↪ memory					
/0/1		processor	Intel(R)		
↪ Core(TM) i7-4770 CPU @ 3.40GHz					
/0/100		bridge	4th Gen Core		
↪ Processor DRAM Controller					
/0/100/1		bridge	Xeon E3-1200		
↪ v3/4th Gen Core Processor PCI Express x16					
↪ Controller					
/0/100/1/0		display	Caicos GTX		
↪ [Radeon HD 8490 / R5 235X OEM]					
/0/100/1/0.1		multimedia	Caicos HDMI		
↪ Audio [Radeon HD 6450 / 7450/8450/8490 OEM / R5					
↪ 230/235/235X OEM]					
/0/100/14		bus	8 Series/C220		
↪ Series Chipset Family USB xHCI					
/0/100/16		communication	8 Series/C220		
↪ Series Chipset Family MEI Controller #1					
/0/100/19	eth0	network	Ethernet		
↪ Connection I217-LM					
/0/100/1a		bus	8 Series/C220		
↪ Series Chipset Family USB EHCI #2					
/0/100/1b		multimedia	8 Series/C220		
↪ Series Chipset High Definition Audio Controller					
/0/100/1d		bus	8 Series/C220		
↪ Series Chipset Family USB EHCI #1					
/0/100/1f		bridge	Q87 Express		
↪ LPC Controller					
/0/100/1f.2		storage	SATA		
↪ Controller [RAID mode]					
/0/100/1f.3		bus	8 Series/C220		
↪ Series Chipset Family SMBus Controller					
/0/2	scsi1	storage			
/0/2/0.0.0	/dev/cdrom	disk	DVD+-RW		
↪ SN-208FB					
/0/2/0.0.0/0	/dev/cdrom	disk			
/1	docker0	network	Ethernet		
↪ interface					

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: N/A

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. Results reported are averaged over many runs.

ARTIFACT EVALUATION

Verification and validation studies: Correctness: Using extensive test suite and CI for making sure results are correct. Since the algorithm is combinatorial & deterministic, most results/aspects can very easily be verified for correctness.

Accuracy and precision of timings: Performance: multiple executions, each running multiple timing iterations.