PARALLEL AND SCALABLE COMBINATORIAL STRING ALGORITHMS ON DISTRIBUTED MEMORY SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Patrick Flick

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

May 2019

PARALLEL AND SCALABLE COMBINATORIAL STRING ALGORITHMS ON DISTRIBUTED MEMORY SYSTEMS

Approved by:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard W. Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Bora Uçar
Département d'Informatique
CNRS and ENS Lyon

Date Approved: March 11, 2019

ACKNOWLEDGEMENTS

First and foremost, I want to to express my gratitude towards my advisor Dr. Srinivas Aluru: thank you for your many years of constant support, advice, encouragement, and the freedom given to pursue my research interests.

To my thesis committee members, Richard Vuduc, Ümit Çatalyürek, Sudhakar Yalamanchili, and Bora Uçar, thank you for your advice, comments, and help in making this dissertation possible.

To my friends, fellow Georgia Tech Ph.D. students, and labmates: thank you for countless wonderful discussions, fun distractions, and our amazing time together. You truly made my time at GT enjoyable. I want to thank all my past and current labmates for all the amazing time we spend together - may it be due to research, slacking off, long winded discussions, or grabbing beers: Indranil Roy, Tony Pan, Rahul Nihalani, Chirag Jain, Ankit Srivastava, Don Kushan Saminda ”Sammy” Wijeratne, Harsh Shrivastava, Nagakishore Jammula, Neda Tavakoli, Shruti Shivakumar, and Haowen Zhang. Furthermore, I want to thank former and current students GT from other groups who became great friends over the years: Srinivas Eswar, Elias Khalil, Robert Chen, Lluis Munguia, Adam McLaughlin, Mikhail ”Mike” Isaev, Lanssie Ma, and Marat Dukhan: thank you for all the fun times we shared together! I also want to thank my former room mates and other Atlanta friends: Arturo Santa Ruiz, Jason Fernando, Daniel Henderson, Nolan Wagener, Ashley Kunkle, and Alireza Nazari: thank you for a fun many years and for being great friends!

Most of all, to Eileen Shi: thank you for your love and support, thank you for always being there for me when I needed you and for supporting me through these many years, thank you for sharing all your happiness and joy, thank you for all the fun adventures and trips we took together - I love you.

To my early mentors, teachers, and partners in crime: Thank you so much for inspiring curiosity, sharing a passion for learning, and many thanks for your guidance and useful advice. A special thank you to my early mentor Peter Karcher: I am so grateful that you took me under your wing when I was still attending high school. Thank you for sharing your knowledge and wisdom, and guiding me towards a path that lead to where I am today. I would not be where I am today without you.

Last but not least, to my loving parents: thank you for your unconditional love and support. Thank you for always supporting and encouraging my decisions, even if it meant I would live far far away.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Methods for processing and analyzing DNA and genomic data are built upon combinatorial graph and string algorithms. The advent of high-throughput DNA sequencing is enabling the generation of billions of reads per experiment. Classical and sequential algorithms can no longer deal with these growing data sizes - which for the last 10 years have greatly out-paced advances in processor speeds. To process and analyze state-of-the-art genomic data sets require the design of scalable and efficient parallel algorithms and the use of large computing clusters.

Suffix arrays and trees are fundamental string data structures which lie at the foundation of many string algorithms, with important applications in text processing, information retrieval, and computational biology. Consequently, the parallel construction of these indices is an actively studied problem. However, prior approaches lacked good worst-case run-time guarantees and exhibit poor scaling and overall performance.

In this dissertation, we first introduce our distributed memory parallel algorithms for the construction of suffix arrays and longest common prefix (LCP) arrays that simultaneously achieve good worst-case run-time bounds and superior practical performance.

We then present a work-optimal distributed memory parallel algorithm for the construction of suffix trees. We formulate a generalized version of the All-Nearest-Smaller-Values problem and provide an optimal distributed solution. We then show how this algorithm can be used to construct the suffix tree of a string given its suffix and LCP arrays. In contrast to the linear work performed by our algorithm, previous distributed memory algorithms exhibit quadratic worst-case complexity. Our algorithm is a clear improvement over the prior state-of-the-art, as it improves theoretical complexity and exhibits far better practical performance.

Next, we introduce a novel distributed string index, the Distributed Enhanced Suffix Array (DESA), which is based on the suffix and LCP arrays and adds additional data structures. The DESA is designed to allow efficient pattern search queries in distributed memory while requiring at most $O(n/p)$ memory per process. We present efficient distributed-memory parallel algorithms for querying, as well as for the efficient construction of this distributed index.

Finally, we present our work on distributed-memory algorithms for clustering de Bruijn graphs and its application to solving a grand challenge metagenomic dataset.

CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Strings, Pattern Matching, and String Indexes

A *String* or *Text* $S = s_0 s_1 \cdots s_{n-1}$ 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 symbols. The string S doubles as an array, so that we can access any character s_i by using array notation $S[i]$ in constant time. For example, a genome can be represented as a string with alphabet $\Sigma = \{\texttt{a, t, c, g}\}$.

Given a *pattern* string $P = p_0 p_1 \ldots p_{m-1}$ of length m, we are interested in finding if and where the string P occurs within a larger string S, where commonly $n >> m$. This problem is referred to by multiple names, such as the *String Matching*, *Pattern Matching*, or *Substring Search* problem. Well known algorithms such as the *Rabin-Karp* algorithm [1], the *Knuth-Morris-Pratt* algorithm [2], or the *Boyer-Moore* algorithm [3] pre-process the pattern string P and then linearly scan the larger string S for occurrences.

1.1.1 String Indexes

(Sub)string Indexes are data-structures created by pre-processing the string S, subsequently allowing much faster pattern search.

Here, we distinguish between *structured* and *unstructured texts*. Natural language documents (such as written English) or source code are examples of *structured texts*, i.e., texts that can be split into tokens or words. Such texts can be indexed using *dictionaries*, which map each word to a location in a document. *Unstructured texts* such as DNA sequences allow no division into disjoint words.

A string index for an unstructured texts needs to index all substrings of the given string

S. Examples of such string indexes include the k-mer index, suffix tree, suffix array, and FM-index.

A k-mer is defined as a length k substring. The k-mers of a string S refer to all length k substrings of S, i.e., the k prefixes of all suffixes of S. Often, k-mers are represented as integers in the range 0 to $|\Sigma|^k - 1$, and can thus be used as an index into a lookup table.

A *k-mer Index* is a string index which allows finding patterns of fixed size k in constant time. k-mer indexes are most commonly implemented as hash tables, indexed by each k-mer's corresponding integer value. A clear disadvantage of k-mer indexes is that k is fixed, and only patterns of length k can be queried. Depending on the representation of the k-mer index, it may be possible to find shorter queries as well.

Suffix Trees and *Suffix Arrays* instead 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

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

i.e., the suffix of S starting at position i. For simplifying presentation and to break ambiguity, an additional terminating character \$ is often appended to the input string $S = s_0 s_1 \cdots s_{n-1}\$$ (i.e., $s_n = \$$). By defining $\$ < c$ for all $c \in \Sigma$, the lexicographical ordering of all suffixes becomes unique.

1.1.1 Suffix Trees

The *Suffix Tree (ST)* of a string S is a (compacted) trie of all the suffixes of S. The leafs of the tree are the lexicographically sorted suffixes $Suf(i)$ 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. Figure 1.1 shows the Suffix Tree for the string $S = \texttt{mississippi\$}$ in blue.

For a given string S of length n, its ST can be constructed in $O(n)$ time [4]. 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 their invention in 1973, suffix trees have become the de facto data structure for the design of numerous clever string algorithms, such as approximate pattern matching, identification of longest common substrings, finding maximal suffix-prefix overlaps, data compression, and many more [5]. 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.

1.1.1 Suffix arrays and LCP arrays

The *Suffix Array (SA)* of a string S is an array of length n containing the lexicographically sorted order of all suffixes of S, and as such, compactly represents the leafs of a *Suffix Tree*. Each suffix S_i is represented by its starting position i, so that $SA[j] = i$, if and only if the suffix $Suf(i)$ 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.

$$LCP[i] = lcp(Suf(SA[i-1]), Suf(SA[i]))$$

The definition above is undefined for $i = 0$, hence we set the value for this position in the LCP array to $LCP[0] = 0$. Figure 1.1 illustrates the concepts of the *suffix array* (in red) and the *LCP Array* (in yellow) for the input string $S =$ `mississippi$`.

Manber and Myers first introduced suffix arrays as a space-efficient alternative to suffix trees [6], and showed how to use suffix arrays for exact pattern matching in $O(|P| \log n)$ time, and when used in conjunction with the *Longest Common Prefix (LCP)* array in $O(|P| + \log n)$ time, i.e., at the additional cost of $O(\log n)$ compared to suffix trees. Man-

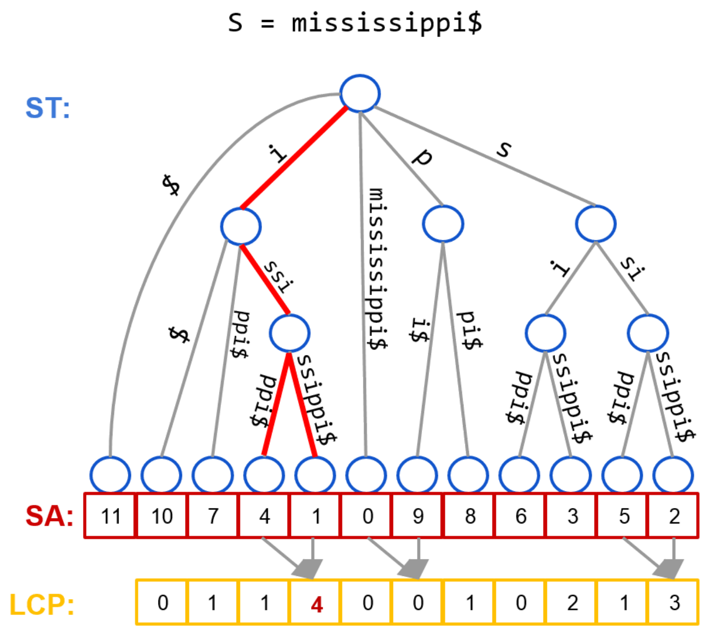


Figure 1.1: Suffix tree, Suffix Array, and LCP Array for the example input string $S =$ `mississippi$`.

ber and Myers introduced an algorithm to construct suffix arrays along with LCP arrays in $O(n \log n)$ time. Suffix arrays can also be constructed in linear time. Interestingly, direct linear time algorithms that avoid suffix trees as an intermediary step were invented only in 2003 [7, 8]. Puglisi *et al.* give a good survey of various approaches to construct suffix arrays [9].

Suffix arrays are often used in conjunction with *Longest Common Prefix (LCP)* arrays. The LCP array simply records the length of the longest common prefix between every pair of consecutive suffixes in the suffix array. Abouelhoda *et al.* showed that suffix arrays combined with *LCP* arrays, as well as a child table can support a majority of operations supported by suffix trees [10]. The *LCP* array can be constructed either during the construction of the suffix array [6, 7], or from a given suffix array in linear $O(n)$ time [11].

Figure 1.2 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 $SA[l], \ldots, SA[r]$. 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

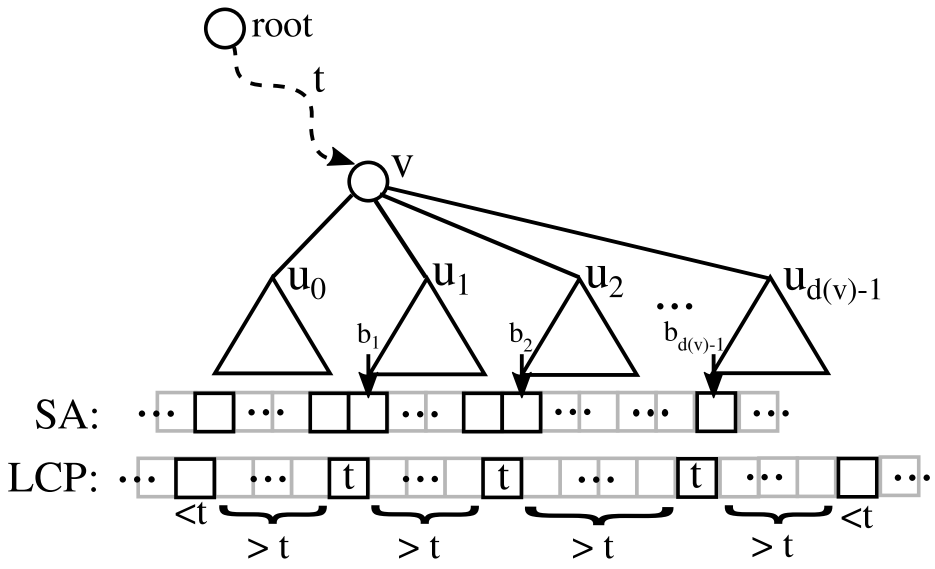


Figure 1.2: Relationship between suffix trees, suffix arrays, and LCP arrays: a subtree starting at an internal node v with string-depth t and child nodes $u_0, u_1, \ldots, u_{d(v)-1}$.

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 $LCP[i] = t$. 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 $[l, r]$.

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. 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.

1.1.1 Generalized Suffix Arrays and Trees

Suffix arrays and trees can also be constructed for a collection of strings, and referred to as *generalized suffix arrays (GSA)* and *generalized suffix trees (GST)*. Further, we denote the LCP array in this case as the generalized LCP (GLCP). In this case, the input is given as

m strings S_i with $i \in \{0, \ldots, m-1\}$, each of length n_i and with total combined length $n = \sum_{i=0}^{m-1} n_i$. We denote the j^{th} suffix of string S_i by $Suf_i(j)$.

Having multiple strings, suffixes may no longer be unique, i.e., two or more strings may end in the same suffix. A common approach to define a unique deterministic ordering of all suffixes is to append a unique terminating character $\$_i$ to each string S_i with $\$_0 < \$_1 < \cdots < \$_{m-1}$ and $\$_i < c$ for all $c \in \Sigma$. Then, defining S as the concatenation of all strings $S = S_0\$_0S_1\$_1 \cdots S_{m-1}\$_{m-s}$, the suffix array and LCP array of S yields the GSA and the corresponding GLCP array for the collection of strings.

However, introducing m new characters to the alphabet is not practical, because multiple optimizations depend on constant alphabet size $|\Sigma| = O(1)$, and the string representation uses at most one byte per character. Concatenating all strings and using the same separating character $\$$, which in practice is often the null character `'\0'`, does not yield the correct results. Consider two identical suffixes which are followed by strings sharing a common prefix, then the LCP computed for these two will falsely include the number of characters shared in the prefix of the strings following the two suffixes.

1.1.2 De Bruijn Graphs

A *de-Bruijn graph* of a string or a set of strings is defined as a graph where the set of vertices are all unique k-mers of the strings. An edge connects two vertices u_i and u_j, iff the string or string set contain a $k+1$ length substring Q for which $Q[0 \ldots k-1]$ is the k-mer corresponding to u_i and $Q[1 \ldots k]$ is the k-mer corresponding to u_j. De-Bruijn graphs find heavy use in computational biology, for example in error correction, genome assembly and scaffolding.

The assembly problem becomes highly compute and memory intensive as the size and complexity of the data set increases. In the absence of errors, the size of the de Bruijn graph is equivalent to the number of unique k-mers in the data set, thus is bounded by the number of base pairs in the genome(s). For metagenomic datasets, however, the presence of large number of species substantially increases this bound. For these datasets, the de-Bruijn graph has as much as 100s of billions of nodes.

1.2 Parallel and Distributed Computing

1.2.1 Shared-Memory and PRAM

A shared-memory parallel machine has $p > 1$ parallel processors that all share a common main memory. Parallel algorithms for shared-memory machines may be expressed as concurrently running threads or processes of execution, which can access the same memory locations for sharing information required for parallel computing.

A *Parallel Random Access Machine (PRAM)* is an abstraction for a shared-memory parallel machine and a parallel extension for the RAM computing model used for complexity analysis. In the PRAM setting, one considers to have p parallel processors, each of which can access any location in a common shared memory in constant time.

1.2.2 Distributed Memory

In a distributed memory parallel machine processes generally do not share a common main memory. Concurrently running processes may exchange information via the passing of messages. Sharing information in this model is no longer possible in constant time. Analyzing distributed parallel algorithms thus considers the cost associated with the sending of messages.

For analysis, a message of size m (often in bytes) is assumed to take time

$$\tau + \mu m$$

where τ models the *latency* and μ models the bandwidth as $\mu \sim \frac{1}{\text{Bandwidth}}$. The values for τ and μ heavily depend on the given systems and network, and are thus left as variables during the runtime analysis. The analysis is performed in O-notation. For example, a parallel reduction (sum) of n numbers with p processors has complexity $O(\frac{n}{p} + (\tau + \mu)\log p)$ in this model assuming a Hypercube or fully connected network.

1.2.3 Parallel Programming Model

The distributed memory algorithms introduced in this thesis are implemented using the *Message Passing Interface (MPI)* - a C/Fortran API for sending and receiving messages on distributed computing systems. MPI implements many useful collective operations, e.g., for broadcasting, gathering, scattering, and all-to-all communication between all or a subset of processors. We assume the reader is familiar with the definition of these operations.

1.2.4 Data Distribution

For most algorithms discussed in this thesis, all data including input, output, and working data is distributed equally among p processors. Let A be a sequence (e.g., an array) $A = (a_0, a_1, \dots, a_{n-1})$ of size n. We call a sequence A *(equally) block distributed* across the p processors, if the following distribution of elements onto processors holds. If n is divisible by p, then each processor contains exactly $\frac{n}{p}$ consecutive elements of the sequence, such that processor i contains the subsequence $A_i = (a_{i\frac{n}{p}}, \dots, a_{(i+1)\frac{n}{p}-1})$. Otherwise, let the remainder of the division be $r = n \mod p \neq 0$. In this case, the first r processors contain $\lceil \frac{n}{p} \rceil$ consecutive elements each, and the remaining processors contain $\lfloor \frac{n}{p} \rfloor$ elements each.

We assume that n and p is known by each processor, along with its rank i among processors with $0 \leq i < p$. Then, each processor i can independently calculate its *global index* range as:

$$\left[i\lfloor \frac{n}{p} \rfloor + \min(i, r),\ (i+1)\lfloor \frac{n}{p} \rfloor + \min(i+1, r) \right)$$

Conversely, for any element with global index $0 \leq j < n$, any processor can independently and in constant time determine the rank of the processor on which this element resides:

$$rankof(j) = \begin{cases} \frac{j}{\lfloor n/p \rfloor + 1} & \text{if } j < (\lfloor n/p \rfloor + 1)r \\ r + \frac{j - (\lfloor n/p \rfloor + 1)r}{\lfloor n/p \rfloor} & \text{if } j \geq (\lfloor n/p \rfloor + 1)r \end{cases}$$

For the sake of simplifying the presentation and without loss of generality, we assume from here on that p equally divides n.

1.2.5 Notions of Scalability

Speedup and *Efficiency* are common metrics used to evaluate parallel algorithms, both theoretically and practically.

The *(Absolute) Speedup* of an algorithm with runtime $T(n,p)$ for input size n and using p processors is defined as

$$\text{Speedup}(n,p) = \frac{T_{seq}(n)}{T(n,p)}$$

where $T_{\text{seq}(n)}$ is the runtime of the best sequential algorithm or implementation. Speedup can be analyzed theoretically in O-notation, as well as numerically computed from experimental data. *Relative Speedup* is the speedup measured or derived by comparing against the runtime with $p = 1$ processors for the same algorithm instead of comparing against the best known sequential algorithm:

$$\text{Relative-Speedup}(n,p) = \frac{T(n,1)}{T(n,p)}$$

The *Efficiency* of a parallel algorithm is in turn defined as:

$$\text{Efficiency}(n,p) = \frac{\text{Speedup}(n,p)}{p}$$

When analyzing these metrics of scalability, we distinguish between *strong scaling*, which fixes the input size n and varies p, and *weak scaling*, which fixes the input size per processor (fixed $\frac{n}{p}$).

We consider an algorithm or distributed data structure as *memory scalable*, if each processor uses at most $O(\frac{n}{p})$ memory. This allows scaling to arbitrarily large inputs by increasing p, i.e., by adding more processors to the job or system.

1.3 Contributions and Overview

In this dissertation, we present distributed memory parallel algorithms for solving string and graph problems stemming from applications in computational biology. Our algorithms are designed to be able to scale to much larger problems compared to prior approaches, a necessity motivated by the rapid increase in the sizes of genomic data sets.

The advent of high-throughput DNA sequencing is enabling the generation of billions of reads per experiment. Classical and sequential algorithms can no longer deal with these growing data sizes - which for the last 10 years have greatly out-paced advances in processor speeds. To process and analyze state-of-the-art genomic data sets require the design of scalable and efficient parallel algorithms and the use of large computing clusters.

Suffix arrays and trees are fundamental string data structures which lie at the foundation of many string algorithms, with important applications in text processing, information retrieval, and computational biology. Conversely, the parallel construction of these indices is an actively studied problem. However, prior approaches lack good worst-case run-time guarantees and exhibit poor scaling and overall performance.

In order to be able to scale to very large inputs, our algorithms are designed so that all data and data structures are fully distributed, requiring no more than $O(n/p)$ memory per processor - a key constraint that allows our algorithms to scale to arbitrarily large inputs given enough compute nodes. Surprisingly, most prior approaches do not follow this constraint and require up to $O(n)$ memory per node - a drastic limitation for scalability.

In Chapter 2, we present our work on distributed memory parallel construction of suffix arrays and LCP arrays. Whereas prior approaches with best performance lack efficient worst-case guarantees and vice versa, our algorithms advance the state-of-the-art by simultaneously achieving both: good worst-case run-time bounds and superior practical performance.

Next in Chapter 3, we present our generalized formulation and distributed algorithmic solution for the *All-Nearest-Smaller-Values (ANSV)* problem, and its application to constructing suffix trees in parallel distributed memory. The resulting distributed suffix tree construction algorithm is work optimal and a clear improvement of the state-of-the-art, both in terms of its theoretical complexity and practical performance.

Then in Chapter 4, we introduce a novel distributed string index, the Distributed Enhanced Suffix Array (DESA). This distributed data-structure allows efficient construction and querying, all while requiring at most $O(n/p)$ memory per process. We derive required properties of the data structure, and provide efficient algorithms for querying, as well as for the efficient construction of this distributed index.

Finally, in Chapter 5, we present our work on distributed-memory algorithms for finding connected components in large graphs, and its application to clustering the de Bruijn graph of a grand challenge metagenomic dataset.

The original work covered in this dissertation is published in the following papers:

- P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 16

- P. Flick and S. Aluru, "Parallel construction of suffix trees and the all-nearest-smaller-values problem," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, IEEE, 2017, pp. 12–21

- P. Flick and S. Aluru, "Distributed enhanced suffix arrays: Efficient algorithms for construction and querying," in *SPAA'19 (under review)*

- P. Flick *et al.*, "A Parallel Connectivity Algorithm for de Bruijn Graphs in Metagenomic Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 15

- C. Jain *et al.*, "An adaptive parallel algorithm for computing connected components," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, 2017
- P. Flick *et al.*, "Reprint of a parallel connectivity algorithm for de bruijn graphs in metagenomic applicationsi," *Parallel Computing*, vol. 70, pp. 54–65, 2017

CHAPTER 2
DISTRIBUTED PARALLEL CONSTRUCTION OF SUFFIX ARRAYS

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. Prior work 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 chapter, 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. We published this work at Supercomputing 2015 [12].

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 present several algorithm engineering techniques that improve performance in practice. 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 $110\times$ compared to *divsufsort* [18], the fastest sequential suffix array construction implementation, commonly used as comparison [19] [20].

The rest of this chapter is organized as follows: In Section 2.1, we review the prior state of the art. In Section 2.2, we introduce further notation and relevant concepts that will be used throughout the chapter. We then describe our suffix array and LCP array construction algorithms in Section 2.3 and Section 2.4, respectively. Section 2.5 contains detailed experimental results, followed by conclusions in Section 2.6.

2.1 Related Work

There are multiple prior approaches for parallelizing suffix array construction. The *Futamura-Aluru-Kurtz (FAK)* [21] 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)* [22]. 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. [23] provides a MPI based implementation of the *FAK* algorithm, adapted to cloud computing on *AWS (Amazon Web Services)*.

Another parallel distributed memory algorithm *pDC3* was introduced by Kulla and Sanders [24]. This algorithm is based on the *skew/DC3* linear time, recursive construction algorithm of Kärkkäinen and Sanders [7]. 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.* [25] introduced the *mkESA* algorithm, which is a parallelized variant of the *Deep-Shallow* algorithm of Manzini and Ferragina [26]. The authors of *mkESA* report a speedup of less than 2 when using 16 threads. Mohamed and Abouelhoda [27] introduced a parallel bucket pointer refinement *(pBPR)* algorithm based on the algorithm of Schürmann and Stoye [28]. 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 [20] 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 [29], and Osipov [19] 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 [29]. Osipov follows a prefix-doubling approach similar to Larsson and Sadakane [30] 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* [19]. Another GPU implementation utilizing prefix-doubling is available in the *Nvidia NVBIO* library.

2.2 Preliminaries

For the definitions and notations for strings, string sets, Suffix Arrays (SA), and LCP arrays, please refer to Section 1.1 of the Introduction Chapter.

In this Chapter, we describe the algorithms for the generalized case of multiple input strings, i.e., the construction of the generalized suffix array. Note that the suffix array of a single string is a special case here $m = 1$. Some steps of the algorithms simplify for the case of a single string ($m = 1$), in which case we will note those changes in the algorithm descriptions below.

2.2.1 Distributed Strings

In the case of a single string, we assume that the input string S of length n is *block distributed* across the p processors according to the characters $S[j]$.

For the generalized case of indexing multiple strings, we define an additional data structure. In general, a collection of strings can be represented in a number of different ways. For example, lines in a file, DNA reads in fasta/fastq format, strings in a database or table format, in-memory as an array of pointers, and many more.

For our distributed memory representation of a collection of strings $S_0, S_1, \ldots, S_{m-1}$, we index single characters by assuming all strings are concatenated without any separating characters, and block distribute the working and output data according to this total size $n = \sum_j n_j$. Note, this representation is internal to our algorithm used for indexing, it is not a hard requirement for the input file(s).

Additionally, we introduce an array L as the exclusive prefix sum over the string sizes: $L[i] = \sum_{j=0}^{i-1} n_j$, $L[0] = 0$. The array L is distributed across processors such that $L[i]$ is locally accessible to all processors which contain at least one character of string S_{i-1} or S_i according to the block distribution of concatenated string. This distribution of L allows local lookup of the global starting and ending position, as well as length of a string S_i. The j^{th} character of the i^{th} string is then located at global index $L[i] + j$.

We define $Suf_S(i)$ as the suffix which starts at global position i within the concatenated string S, i.e., it is the suffix $Suf_k(j)$ of string S_k where $i = L[k] + j$, $j > 0$ and k is the maximum for which this holds. Note, if there is a single input string $m = 1$, then $Suf_S(i) = Suf_0(i)$.

2.3 Parallel Suffix Array Construction

2.3.1 Prefix Doubling

Our approach is motivated by *Prefix Doubling*, which was first introduced by Karp *et al.* [31], and then first utilized by Manber and Myers [6] 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 $Suf(i)$ and $Suf(j)$. The ordering of these two suffixes according to their $2h$-prefix can be deduced by using the current relative ordering of suffixes $Suf(i + h)$ and $Suf(j + h)$, 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)* [6] 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 *Inverse Suffix Array (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) [30] 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 $Suf(i)$ 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 (Algorithm PSAC-GS) 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 (Algorithm PSAC-NS) improves upon the first by avoiding global sorts, and sorting only non-singleton h-groups, leading to large improvements in run-time.

2.3.2 Prefix doubling using global sorting

Our algorithm for suffix array construction using global sorting PSAC-GS follows the high level steps as shown in Algorithm 1. In this section, we explain what each step accomplishes and how it is parallelized. We parallelize our algorithms using *MPI*. Most steps of the algorithm map to common parallel primitives such as parallel sorting, `all-to-all`, `all-reduce`, `scan`, and `exscan`.

We keep the following invariants at the beginning of each iteration. The suffix array

ALGORITHM 1: PSAC-GS (construct SA via global sorting)

```
Input: strings $S_0, S_1, \ldots, S_{m-1}$
Output: (Generalized) Suffix Array $SA$ and inverse suffix array $ISA$
1  $B = k$-mers of strings $S_j$
2  $\left[\ldots, \binom{B[i]}{SA[i]}, \ldots\right] = \texttt{sort}([\binom{B[i]}{i} | i = 0 \ldots n-1])$
3  $B = \texttt{rebucket}(B)$ // assign $h$-group ranks
4  for $h = k, 2k, 4k, 8k, \ldots$ do
5      // reorder to string order
6      for $i = 0, \ldots, n-1$ do
7          $B'[SA[i]] = B[i]$
8      end
9      $B = B'$
10     if $done$ then
11         $ISA = B$
12         break
13     end
14     // shift $B$ by $h$ for each string $S_i$
15     $B_2 = array(n, 0)$ // array initialized to 0
16     for $j = 0, \ldots, m-1$ do
17         for $i = L[j] + h, \ldots, L[j+1] - 1$ do
18             $B_2[i-h] = B[i]$
19         end
20     end
21     // sort tuples lexicographically
22     $\left[\ldots, \begin{pmatrix} B[i] \\ B_2[i] \\ SA[i] \end{pmatrix}, \ldots\right] = \texttt{sort}([\begin{pmatrix} B[i] \\ B_2[i] \\ i \end{pmatrix} | i = 0 \ldots n-1])$
23     $B = \texttt{rebucket}(B, B_2)$ // assign $2h$-group rank
24     $done = \texttt{check-all-singleton}(B)$
25 end
```

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 $Suf()SSA[i]$. We represent each h-group by its leftmost element. We set $B[i] = i + 1$ for the first position i of the h-group and $B[j] = i + 1$ 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. Note that the index into B is 0-based, whereas the h-group ranks start with 1. We reserve rank 0 for use during the doubling stage to represent the rank of string positions past the last character, essentially implementing the \$ terminating character.

Example We show an example of this algorithm in Figure 2.1 for a single input string $S =$`mississippi`. The example illustrates the data dependencies and movements for the different steps of the algorithm using gray arrows.

k-mer sorting In the first step of the algorithm we generate the k-mers of each string S_i. These are the length k prefixes of all suffixes of S_i. We map the alphabet Σ to integers $1, \ldots, \sigma$, and use the integer 0 as a fill character for the last $k-1$ suffixes of each string S_i, i.e., those which do not have k characters. We choose k according to the alphabet size, such that all k characters of a k-mer fit into a single machine word. Let W be the bit-length of the machine word (e.g. 32 or 64). We then chose k as:

$$k = \left\lfloor \frac{W}{\lceil \log_2(\sigma + 1) \rceil} \right\rfloor$$

The k-mers of a string can be generated in linear time by a simple scan through the characters of the string and by using shift and bit operations. Note, this is the only time during the suffix array construction during which we read the input string(s). The input can be read in parallel by equally splitting the input file(s). Then, each processor i sends the first $k-1$ characters to its left neighboring processor (rank $i-1$,) and can then independently generate all k-mers for it's allocated input.

We store the k-mers of all strings S_i into an array B, and then sort them lexicographically using parallel sorting. 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 elements of the tuples yields the SA according to a k-ordering.

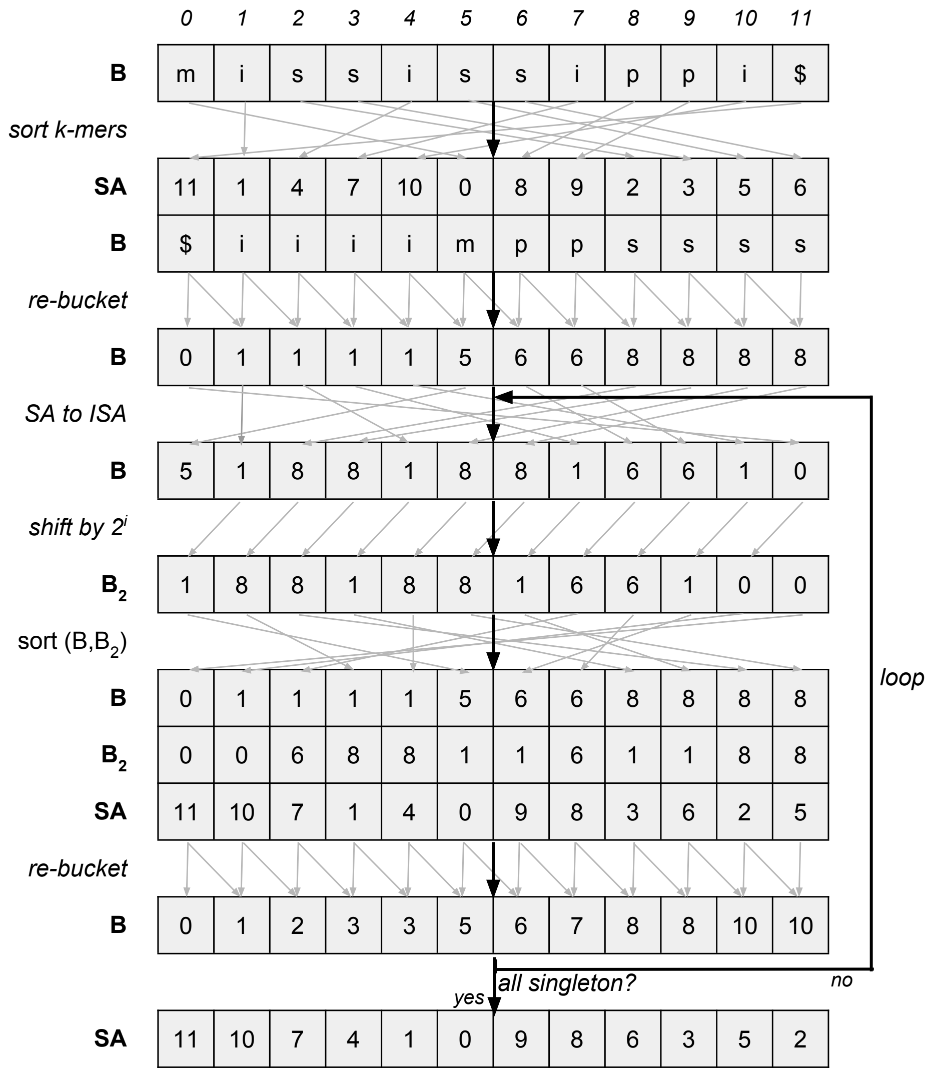
Re-bucketing After sorting the k-mers, we need to assign the k-group ranks to each position in B. Since each k-group consists of identical k-mers, it appears in a contiguous block in B. Thus, we can assign the new k-group ranks using a simple linear scan through all positions i of B. If two k-mers are identical and end in a 0 character, then they represent identical suffixes coming from different strings. Since we sorted the $\langle k\text{-mer}, i\rangle$ tuples lexicographically, they are already in the correct ordering with respect to each other and we assign distinct k-group ranks. Note, in the single string case suffixes are unique, and it suffices to sort by the k-mer value.

The re-bucketing within the main loop follows the same procedure, but here we check for new $2h$-groups, which begin whenever either (1) $B[i-1] = B[i] \wedge B_2[i-1] \neq B_2[i]$, or (2) $B[i-1] = B[i] \wedge B_2[i-1] = B_2[i] = 0$. In the second case, the two suffixes are identical but from different strings.

When solving this in parallel, an h-group might span across multiple processors and finding the beginning of an h-group requires communication. To solve this efficiently, we perform two passes over the data. In the first pass, we identify each position i which is the first of its h-group, i.e., those for which either (1) $B[i-1] \neq B[i] \vee B_2[i-1] \neq B_2[i]$ or (2) $B[i-1] = B[i] \wedge B_2[i-1] = B_2[i] = 0$. We set these elements to $B[i] \leftarrow i+1$, while setting all other elements of the same h-group to 0. We then perform a prefix-scan with the $\max$ operation as the combination operator. The result of this operation is that $B[i] = i+1$ only for the first element of each h-group and $B[j] = i+1$ for all $j \geq i$ within the same h-group. Hence, this operation re-establishes the invariant for B.

Reorder to string order For prefix doubling of each string position i, we need to pair up the current h-group rank for each suffix $Suf_j(i)$ with the h-group rank of suffix $Suf_j(i+h)$. To efficiently pair these ranks, we reorder the h-group ranks available in B from the current suffix array ordering into the ordering according to the string indexes, yielding the ISA with h-group ranks.

For each position $B[i]$ in the h-ordered sorted order, we know it's corresponding original string position due to the array SA. Thus, the reordering of B to string order can be performed by permuting B according to the indexes in SA. Sequentially, this can be done by a simple linear scan. If all h-groups are singletons, the reordered B is equal to the final



0 1 2 3 4 5 6 7 8 9 10 11
B: m i s s i s s i p p i $
sort k-mers
SA: 11 1 4 7 10 0 8 9 2 3 5 6
B: $ i i i i m p p s s s s
re-bucket
B: 0 1 1 1 1 5 6 6 8 8 8 8
SA to ISA
B: 5 1 8 8 1 8 8 1 6 6 1 0
shift by 2^i
B_2: 1 8 8 1 8 8 1 6 6 1 0 0
sort (B,B_2)
B: 0 1 1 1 1 5 6 6 8 8 8 8
B_2: 0 0 6 8 8 1 1 6 1 1 8 8
SA: 11 10 7 1 4 0 9 8 3 6 2 5
re-bucket
B: 0 1 2 3 3 5 6 7 8 8 10 10
loop
all singleton?
yes
no
SA: 11 10 7 4 1 0 9 8 6 3 5 2

Figure 2.1: Example of suffix array construction by global sorting for the input string $S =$mississippi\$ and $k = 1$.

ISA.

In the parallel setting, we perform this reordering 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.

Shifting by h In order to perform prefix doubling of each suffix $Suf_j(i)$, we need to pair the h-group rank of $Suf_j(i)$ with $Suf_j(i+h)$. In the reordered B array, these are now simply accessible at the respective positions $B[L[j]+i]$ and $B[L[j]+i+h]$. To combine these two ranks locally, we define an array B_2 and fill it as shown in Algorithm 1. This accomplishes that $B_2[i]$ will be equal to $B[i+h]$ if the suffix at $i+h$ is from the same string as the one at i, and otherwise $B_2[i]=0$.

For each string, this operation can be seen as a left-shift of values in B by an offset h. To perform this in the parallel, distributed memory setting, each processor identifies its at most two string which span across other processors, and sends the corresponding values in B to the left, and receives values from the right. Since the B array is evenly distributed, each processor sends data to at most two other processors, and receives from at most two processors.

In the case of a single input string, this reduces to a simple global left shift of the distributed array B.

Doubling via sorting To induce the $2h$-ordering, we interpret each position i of the distributed arrays B and B_2 as a tuples of size three $\langle B[i], B_2[i], i\rangle$ and sort these tuples lexicographically. After sorting, the third tuple position represents the SA in a $2h$-ordering. We can now re-assign $2h$-group ranks using the re-bucketing procedure explained above.

There are many different approaches for sorting on parallel distributed memory architectures. Blelloch *et al.* [32] give a good review. We implement parallel sample sort with regular sampling in our implementation.

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 after the re-bucketing. 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.

2.3.2 Complexity

The overall complexity of Algorithm PSAC-GS 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 with regular sampling, the sorting complexity is $O(\frac{n}{p} \log \frac{n}{p} + \mu\frac{n}{p} + \tau \log(p))$.

2.3.3 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. After termination, this is equivalent to the rank of the suffix and thus the complete ISA.

Our second approach, PSAC-NS (see Algorithm 2), improves upon PSAC-GS. Algorithm PSAC-GS is 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 PSAC-NS, which takes partially solved SA, B and ISA arrays as input. We assume that all three arrays, ISA, SA, and B are already

ALGORITHM 2: PSAC-NS

```
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: suffix array $SA$ and inverse suffix array $ISA$
1  // let $r$ be the processors rank
2  // we use $i$ as a global index into the distributed arrays
3  // keep track of all non-singleton $h$-groups
4  $W = \left[i \in \{r\frac{n}{p} \dots r(\frac{n}{p}+1)-1\} \mid B[i] \neq i+1 \text{ or } B[i+1] = i+1\right]$
5  while $h \leq n$ do
6      // request the group rank needed for doubling
7      $M \leftarrow \left[\binom{SA[i]+h}{i} | i \in W\right]$
8      $M \leftarrow$ all-to-all($M$, dest = rankof($SA[i]+h$))
9      // send responses (read from $ISA$)
10     $M \leftarrow \left[\binom{ISA[i]}{j} | \binom{i}{j} \in M\right]$
11     $M \leftarrow$ all-to-all($M$, dest = rankof($j$))
12     // sort each unresolved $h$-group by using the keys returned in $M$
13     $W_{new} = \emptyset$
14     for each non-singleton $h$-group $G$ in $W$ do
15         // the $h$-group $G$ is a contiguous section of $SA$ and $B$
16         sort($G$ by the $ISA$ returned in $M$)
17         rebucket(elements of $G$)
18         // keep track of non-singletons $2h$-groups
19         $W_{new} = W_{new} \cup [i \in G \mid B[i] = i+1 \text{ and } B[i+1] \neq i+1]$
20     end
21     // update the $ISA$
22     $M \leftarrow \left[\binom{SA[i]}{B[i]} | i \in W\right]$
23     $M \leftarrow$ all-to-all($M$, dest = rankof($SA[i]$))
24     for $\binom{i}{b} \in M$ do $ISA[i] = b$
25     // update $W$ and check for convergence
26     $W = W_{new}$
27     if $W = \emptyset$ then break
28 end
```

equally distributed among all processors using a block distribution.

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 $B[i] \neq i{+}1$ (this element is not the representative of its h-group) or $B[i{+}1] = B[i]$ (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 compare $B[i]$ with $B[i+1]$ for its last local element i. We keep track of the indexes of all non-singleton elements in an additional distributed 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 $\texttt{rankof}(g)$. 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 $Suf_S(SA[i])$, we need the corresponding h-group rank for the suffix starting at $SA[i]+h$ if this suffix is part of the same string. If the position $SA[i]+h$ is a suffix for another string, then the doubling happens with rank value 0. The required information is available on processor $\texttt{rankof}(SA[i]+h)$ via $ISA[SA[i]+h]$ and the L array.

We execute these doubling queries in a parallel bulk-query fashion: we generate an array M of query 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 $\texttt{rankof}(SA[i]+h)$ and then using an `all-to-all` communication.

On the target processor, for each query $j = SA[i]{+}h$ we check whether the suffix $j-h$ belongs to the same string via L. If it does, we replace the first tuple element by the ISA value: $ISA[j]$, otherwise, we set the first tuple element to 0. This corresponds to the $B_2[i]$ value of PSAC-GS. In order to efficiently query the L array, we first sort the tuples by their first element $SA[i]{+}h$. Then we can process the array of query tuples M by a simple linear scan of both M and L. Note, in the simple case of $m = 1$, this step simplifies to simply accessing $ISA[SA[i]+h]$, in which case there is no need for sorting queries.

Next 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 B_2 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 PSAC-NS has a higher memory consumption than PSAC-GS due to the additional W and M arrays needed to keep track of the non-singleton positions and the messages. We switch from PSAC-GS to PSAC-NS only when the number of remaining non-singleton elements falls 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.

2.4 LCP construction

2.4.1 Calculating the LCP

Manber and Myers [6] first introduced the construction of the *Longest-Common-Prefix (LCP)* array alongside the suffix array. Our approach follows a similar strategy and makes use of observations from [6] 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 $lcp(Suf_S(SA[i]), Suf_S(SA[j])) < h$. Furthermore, the lcp is given by the minimum value of the LCP array within the range between the two h-groups [6]. 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:

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

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 $B[j]$ ($i < j$) do not have a common prefix of length $2h$. Thus these suffixes must have a lcp with: $h \leq lcp < 2h$.

Let the two suffixes be $Suf_S(SA[i]) = Suf_u(i')$ with $SA[i] = L[u]+i'$ and $Suf_S(SA[j]) = Suf_v(j')$ with $SA[j] = L[v]+j'$. If either $i'+h \geq n_u$ or $j'+h \geq n_v$, then the lcp of the two suffixes has to be $\leq h$, however, since the two suffixes were also part of the same h-group, their lcp is equal to h. Note, that in this case at least one of $B_2[i] = 0$ or $B_2[j] = 0$. If otherwise both $i' + h < n_u$ and $j' + h < n_v$, we can determine the lcp between the two $2h$-groups as:

$$\begin{aligned} & lcp(Suf_S(SA[i]), Suf_S(SA[j])) \\ & = h + lcp(Suf_S(SA[i] + h), Suf_S(SA[j] + h)) \end{aligned}$$

Since $lcp(Suf(SA[i] + h), Suf(SA[j] + h)) < h$, we can use the above invariant and property to calculate the LCP directly. Furthermore, we have the h-group ranks for the suffixes $Suf_S(SA[i]+h)$ and $Suf_S(SA[j]+h)$ available locally during the prefix doubling step as $B_2[i]$ and $B_2[j]$ respectively.

We can thus calculate the LCP value for new $2h$-group boundaries via:

$$LCP[B[j]] \leftarrow \begin{cases} h & \text{if } B_2[i] = 0 \text{ or } B_2[j] = 0 \\ h + \min\limits_{q \in R_{ij}} LCP[q] & \text{otherwise} \end{cases}$$

where the minimum is over the range R_{ij} of indexes between $min(B_2[i], B_2[j])$ and $max(B_2[i], B_2[j])$.

2.4.2 LCP construction during SA construction

2.4.2 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. A new k-group begins when either two consecutive k-mers are different, or if they are identical but end in the 0 character. 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. We do so efficiently, by using bitwise `xor` and `or` operations, as well as intrinsics to determine the highest/lowest set bit. To perform this operation in parallel, each processor i requires the last k-mer of the previous processor $i-1$. Thus, each processor simply sends its last k-mer to the next processor, and all other processing can then be performed locally.

2.4.2 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 [6] use a size n search tree. We implement the succinct *Range-Minimum-Query (RMQ)* from Fisher and Heun [33] 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 $O(1)$.

For this, 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*.

2.4.2 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 *RMQ*s 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.

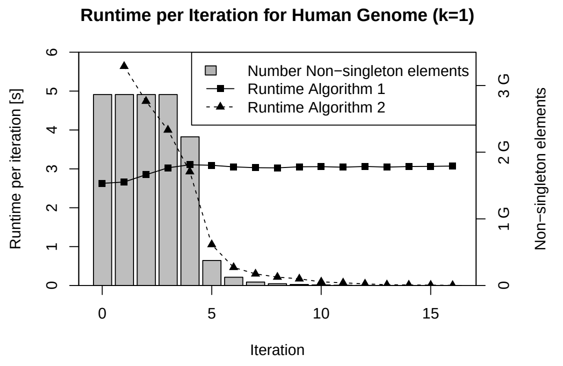
Memory Consumption If the LCP array is computed along the suffix array, an additional $4n$ bytes are required for the LCP array itself. During the LCP construction, additionally $2n+o(n)$ bits are needed for the succinct range-minimum-query, in addition to the memory for the packed message exchange. 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.

2.5 Experiments and Results

2.5.1 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



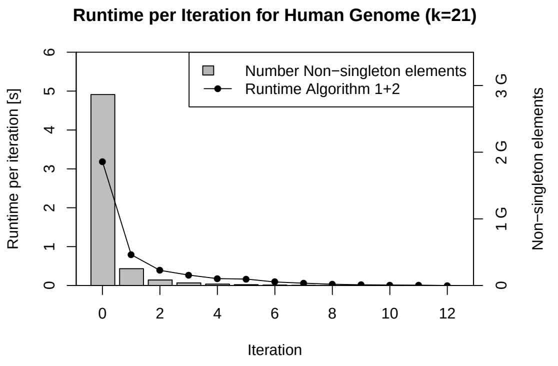


Figure 2.2: Run-time of each iteration of Algorithm 1 and Algorithm 2 and the number of non-singleton elements in each iteration. The *left* graph shows the run-time for both algorithms, given that the first sorting is performed on k-mers with $k = 1$. The *right* graph shows the time per iteration for 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).

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 [34] 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.* [35].

2.5.2 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 2.2 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

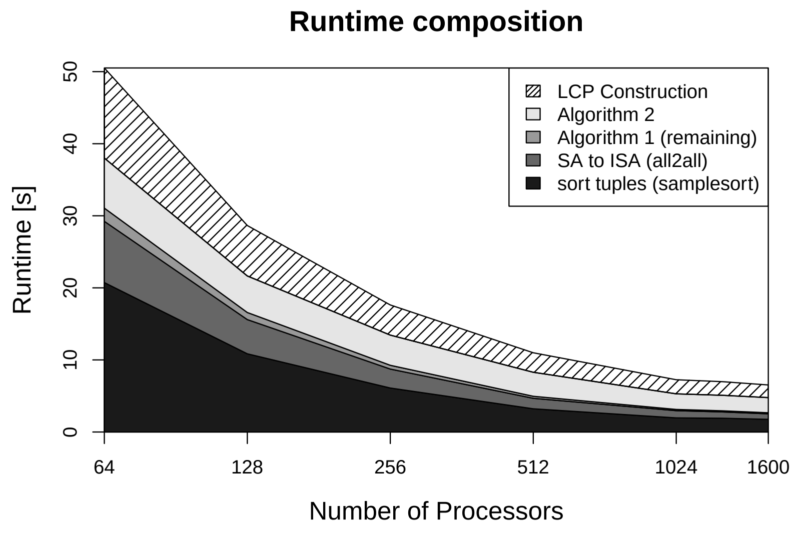


Figure 2.3: Run-time composition for suffix array and LCP construction of the human genome using up to 1600 cores (100 nodes).

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 2.2 shows the run-time per iteration for the combined approach. The first iteration (≈ 3.2 seconds) is the most expensive step of the full suffix array construction, which completes after a total of 5.3 seconds.

Figure 2.3 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 run-time 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.

2.5.3 Comparison with Prior State-of-the-art

We compare our algorithm and implementation against multiple other methods. The *divsufsort* suffix array implementation [18] 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* [25]. Additionally, we directly compare our implementation with the *FAK* implementation *cloudSACA* by Abdelhadi *et al.* [23]. 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 [24, 36, 37, 38], 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 2.1: 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 2.1, 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 processes using an MPI_Scatter operation. In order to compare the performance of our algorithm

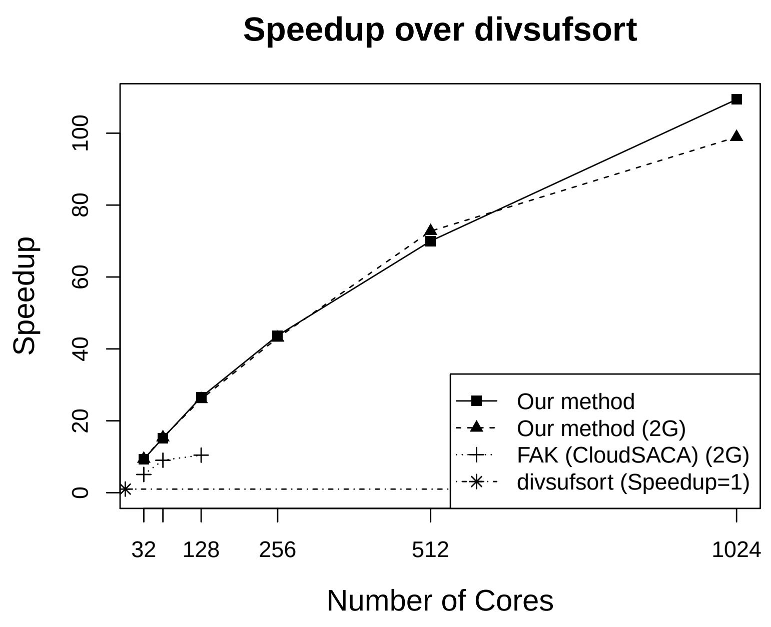


Figure 2.4: Speedup achieved by our suffix array construction algorithm and the *FAK* implementation of Abdelhadi [23]. Speedup is calculated with respect to the sequential algorithm and implementation *divsufsort*. Results are for the human genome. The *(2G)* denotes that the input is limited to only the first 2 billion nucleotides of the human genome.

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 $10\times$ when using 128 processes (see also Figure 2.4). 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 2.4 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

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 2.2: 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.

of our method for the full human genome. Our method reaches speedups of over $110\times$ when using 1024 cores. As such, our algorithm outperforms *cloudSACA* by a large margin.

2.5.4 Scalability and Large Problems

Our approach scales to well above 1024 cores and can handle large genomes. Table 2.2 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.

2.6 Conclusions

In this chapter, we introduced new parallel distributed memory suffix array and LCP array construction algorithms that scale to large inputs and a large number of nodes and cores. Our implementation reaches speedups of above $110\times$ 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.

CHAPTER 3
DISTRIBUTED PARALLEL CONSTRUCTION OF SUFFIX TREES

A Suffix tree is a fundamental and versatile string data structure that is frequently used in important application areas such as text processing, information retrieval, and computational biology. Sequentially, the construction of suffix trees takes linear time, and optimal parallel algorithms exist only for the PRAM model. Recent works mostly target low core-count shared-memory implementations but achieve suboptimal complexity, and prior distributed-memory parallel algorithms have quadratic worst-case complexity.

In this chapter, we present a novel, efficient distributed memory algorithm for constructing the suffix tree for a string given its suffix array and LCP array, an approach that has been used for PRAM and shared memory, but not for distributed memory.

To do so, we introduce a novel generalization of the All-Nearest-Smaller-Values (ANSV) problem and give an optimal algorithm to solve this problem in distributed memory, minimizing overall communication volume. Combining this with the work of the previous Chapter on constructing suffix and LCP arrays from a string [12] results in a parallel algorithm for constructing suffix tree from a given input string. Compared to previous distributed memory algorithms for suffix tree construction, this yields superior theoretical complexity as well as practical performance. We demonstrate the construction of the suffix tree for the human genome given its suffix and LCP arrays in under 2 seconds on 1024 Intel Xeon cores. We published this work at IPDPS 2017 [13].

Given a sequence of values, solving the *All-Nearest-Smaller-Values (ANSV)* problem requires finding for each element the first smaller element to the left (or right). A number of problems can be reduced to the *ANSV* problem, including merging of two sorted sequences, monotone polygon triangulation, Cartesian tree construction, and parenthesis matching [39, 40]. Thus, while we present our parallel algorithm for the ANSV problem to facilitate construction of suffix trees, it is a problem worth studying in its own right and our algorithm can be used in these and many other applications.

3.1 Related Work and Our Contributions

3.1.1 Suffix Tree Construction

Previous methods for constructing suffix trees in parallel are abundant, but focus mostly on the PRAM and shared memory architectures. Apostolico *et al.* [41] showed how to construct suffix trees in $O(\log n)$ time and using $O(n^{1+\epsilon})$ space on a n processor CRCW PRAM. Hariharan [42] introduced the first work-optimal CRCW PRAM algorithm for constant sized alphabets. While these PRAM methods are of theoretical importance and unravel novel techniques for reasoning on strings, to our knowledge no practical implementations exist nor seam feasible.

Constructing the suffix tree of a string from its suffix array and LCP array became viable with the advent of fast, linear work suffix array construction algorithms [8, 7]. Kärkkäinen and Sanders [7] first showed how to construct suffix arrays with $O(n \log n)$ work and in $O(\log^2 n)$ time on a n processor EREW PRAM.

Iliopoulos and Rytter [43] introduced the first parallel CREW PRAM algorithm for constructing the suffix tree for a string from its suffix and LCP arrays. Their algorithm works in linear $O(n)$ space and $O(\log n)$ time using n processors, resulting in $O(n \log n)$ work. Recent work by Shun and Blelloch [44] improved upon this result and provided a linear work, linear space, and $O(\log^2 n)$ time EREW PRAM algorithm for constructing cartesian trees utilizing a previous result for binary tree merging from Hagerup and Rüb [45]. The authors however implement a simpler, non-optimal algorithm for their experiments which requires at least $O(n \log n)$ work, but show good performance on the human genome, for which they construct the suffix tree in 168 seconds on 40 cores of a shared memory machine.

To date, algorithms for constructing suffix trees for distributed memory have failed to provide any reasonable complexity guarantees, while also leaving much room for improvement in both practical performance and scalability.

Ghoting and Makarychev introduced the (*WaveFront*) algorithm for constructing suffix trees in parallel on distributed memory systems, as well as an algorithm for the *External-Memory (EM)* model [36]. Their algorithm scans the input string S of length n up to $O(n/k)$ times and extends the suffix tree by $O(k)$ characters in each of these iterations, where k is a fixed block size dependent on the size of the memory available on each node. Each processor works on an assigned subtree of the suffix tree, but still requires reading the whole input up to $O(n/k)$ times. Hence, the parallel algorithm has a quadratic worst-cast time complexity. In their experiments, the authors illustrate the performance of their algorithm by constructing the suffix tree of the human genome in 15 minutes on a 1024 processor *IBM Blue Gene/L.*

Following a similar principle to *WaveFront*, the *Elastic-Range (ERA)* algorithm, introduced by Mansour *et. al*, improves upon its predecessor [37]. *ERA* uses a dynamic block size, rather than a fixed k, but retains the quadratic worst-case run-time. The authors report a large improvement in runtime: 13.7 minutes on only 32 cores for the whole human genome.

Comin and Farreras [38] claim to improve upon *ERA* and *WaveFront* with their algorithm *Parallel Continuous Flow (PCF)*. Their algorithm is closely based on its predecessors and shares its quadratic time complexity. For building the suffix tree of the human genome, the authors report a runtime of 7 minutes on 172 cores, an improvement of at most 2 times, while using more than 5 times the number of cores compared to its predecessor.

3.1.2 All-Nearest-Smaller-Values

Berkman *et al.* [39] showed how to solve the *ANSV* problem optimally in $O(\log\log n)$ time using $O(n/\log\log n)$ processors on CRCW PRAM, and in $O(\log n)$ time using $O(n/\log n)$ processors on the weaker CREW PRAM model. Based on Berkman's algorithm, Jájá and Ryu (JR) [46] demonstrated how to solve the *ANSV* problem on a *pipelined hypercube* architecture in $O(\frac{n}{p})$ time using up to $p \leq n/(\log^3 n \log\log n)$ processors.

Later, Kravets and Plaxton (KP) [47] introduced an $O(\log n)$ time algorithm for an n input, n processor normal hypercube, and proved its optimality by showing that any hypercube algorithm requires $\Omega(n)$ processors to solve the problem in $O(\log n)$ time. Their algorithm, however, is only of theoretical importance, as it uses a large number of calls to sub-routines, each of which involves routing data across the hypercube.

He and Huang (HH) [40] presented a BSP (Bulk Synchronous Parallel) adaptation of Berkman's algorithm and give experimental results for their MPI based implementation. They show that the communication phase of their algorithm is a $(\frac{n}{p} + p)$-relation. As such, this algorithm is closest to the algorithm we introduce in this work. However, our algorithm has multiple advantages over HH, which will be discussed in detail later.

Both He and Huang's, and Jájá and Ryu's algorithms are closely based on the communication structure first introduced and proven to be correct by Berkman *et al.* These algorithms all assume that all elements in the input are distinct. When used on sequences where some values appear more than once, they do not return the correct results. A simple transformation of the sequence can eliminate duplicates by replacing each element a_i by a pair containing its value and index (a_i, i), and then comparing elements lexicographically. However, this does not yield the expected results to the original problem instance.

3.1.3 Our Contributions

Previous parallel ANSV algorithms [39, 40, 46], including HH, all follow the same communication structure and use the same set of proofs from Berkman et al. [39]. We introduce a generalized formulation of the *ANSV* problem, which handles duplicate values in user specified ways, and generalizes the communication structure, provides novel proofs, and allows optimizing and minimizing total communication volume. Furthermore, our formulation identifies left and right matches simultaneously, whereas the HH algorithm is formulated to find right matches only. Our algorithm has a worst case of $O(\frac{n}{p} + p)$ both in terms of computation complexity and communication volume.

Our parallel suffix tree algorithm improves significantly upon previous distributed memory algorithms both in theory and practice. It runs in $O(\frac{n}{p}+p)$ time, which is work-optimal for $p = O(\sqrt{n})$. All prior algorithms suffer from $O(n^2/p)$ worst-case complexity, although their practical performance is better than quadratic. We illustrate the performance of our algorithm on the human genome, for which we construct the suffix tree in less than 2 seconds on 1024 cores given its suffix and LCP arrays. Combined with the results from the previous chapter [12], this yields a suffix tree construction in overall time of less than 10 seconds.

3.2 All Nearest Smaller Values

3.2.1 Definition and Notation

The *All Nearest Smaller Values (ANSV)* problem is defined as follows: Let $A = (a_0, a_1, \ldots, a_{n-1})$ be a sequence of n elements from a totally ordered set. For each element a_i, find the index of the closest elements to the left and right which are smaller than a_i:

$$l(a_i) = \max_{j<i}\{j \mid a_j < a_i\} \tag{3.1}$$

$$r(a_i) = \min_{j>i}\{j \mid a_j < a_i\} \tag{3.2}$$

We call $l(a_i)$ the *left match*, and $r(a_i)$ the *right match* for element a_i. For convenience, we will simply denote these indexes by $l(i)$ and $r(i)$ instead of $l(a_i)$ and $r(a_i)$, respectively.

If an element a_i does not have a left match we define $l(i) = \perp$, where $\perp$ can be any special value that is not a valid index in $\{0, \ldots, n-1\}$. Similarly, we define $r(i) = \perp$ if the element a_i does not have a right match in A.

3.2.2 Generalized ANSV

We are interested in solving the ANSV problem directly, even if the input values are not all unique. We specify three different ways of handling duplicate values by redefining the meaning of a left (respectively, right) match. In the following, we give three variants for finding left matches. The definitions apply similarly for right matches.

1. Nearest-Smaller (NS): $l_{NS}(a_i) = \max_{j<i}\{j \mid a_j < a_i\}$
2. Nearest-Nonlarger (NN): $l_{NN}(a_i) = \max_{j<i}\{j \mid a_j \leq a_i\}$
3. Nearest-Nonlarger-Furthest-Equal (FE): $l_{FE}(a_i) =$
   $\min_{j<i}\{j \mid (a_j = a_k) \wedge (k = l_{NN}(a_i)) \wedge (j > l_{NS}(a_k))\}$

The definition of *Nearest-Smaller (NS)* is equivalent to the original ANSV formulation, and the *Nearest-Nonlarger (NN)* is a slight variant thereof, where we are interested in finding the nearest non-larger (rather than strictly smaller) element to the left (or right).

The *Nearest-Nonlarger-Furthest-Equal (FE)* variant may seem unintuitive at first, however, we will show how this version allows us to construct multiway Cartesian trees, as well as suffix trees. To illustrate the usefulness of *FE*, we first define the *equal-range* for an element: two elements a_i and a_j with equal value belong to the same *equal-range (er)*, iff no element in between a_i and a_j is smaller:

$$er(a_i) = \{j \mid (a_j = a_i) \wedge \forall i \leq k \leq j : a_k \geq a_i\}$$

When used for multiway Cartesian tree or Suffix tree construction, each element in an equal-range corresponds to the same node in the tree. The *FE* variant, contrary to the other two, defines a unique representative element for each equal-range. For left matches, this is the leftmost element in the equal-range, whereas it is the rightmost element of the equal-range for right matches. Hence, we call this element the *furthest* of its equal-range.

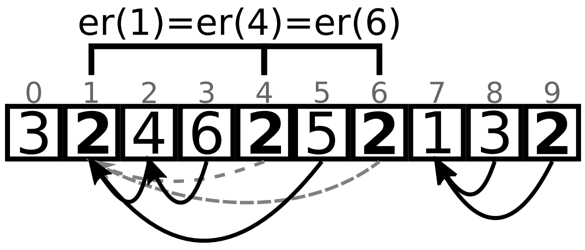


Figure 3.1: Example for *equal-range* and *FE* left-matches l_{FE}.

Whereas the *NS* and *NN* formulations always return the nearest element, the *FE* element returns the unique representative for the matching equal-range. Furthermore, if an element is not the representative for its equal range, the *FE* variant returns the representative as its match.

Example Figure 3.1 illustrates the concepts of *FE* and *equal-range*. Here, we show the equal range for elements of value 2, which contains indices 1, 4, and 6. Note that value 2 at index 9 does not belong to the same equal range, because it is separated by a smaller value 1 at index 7. The arrows visualize the *FE* left-matches for those items which have matches in this sequence. The match for value 5 at index 5 is the representative item of the *equal-range*, in this case the 2 at position 1.

For suffix tree construction, the *FE* variant has the advantage that it allows us to skip an otherwise required merging step. We show how we can solve this generalized ANSV problem with respect to any combination of the variants for left and right matches, and solve the matches for both sides simultaneously.

3.2.3 Our Distributed Memory ANSV Algorithm

Given an input sequence $(a_0, a_1, \dots, a_{n-1})$ of size n, our algorithm assumes that the input sequence is block distributed across the p processors with $\frac{n}{p}$ elements per processor. For simplicity and without loss of generality, assume n is divisible by p. Otherwise, if the remainder $r = n \mod p \neq 0$, the first r processors contain $\lceil \frac{n}{p} \rceil$ elements each, and the remaining processors contain $\lfloor \frac{n}{p} \rfloor$ elements each. Assuming from here on that n is divisible by p, processor i contains the sub-sequence $A_i = (a_{i\frac{n}{p}}, \ldots, a_{(i+1)\frac{n}{p}-1})$.

3.2.3 Overview of the algorithm

For many elements in A_i, their nearest smaller values can be found within A_i, i.e., on the processor itself. For the remaining elements, we define a representative sequence T_i, and solve the distributed problem on this much smaller sequence. To do so, each processor determines for each element in T_i, which processor contains the element's missing left or right match. We then communicate segments of the sequences T_i between processors, in such a way that all missing matches can be solved in parallel. In this step, a missing match for an element in T_i may be solved either locally using values received from other processors, or solved on another processor P_j. In the latter case, the result will be returned to P_i in an additional communication step.

3.2.3 Representative sequences

The sequence T_i on processor i consists of tuples $(val, lidx, ridx)$, each representing the extent of an equal-range for a value val, extending from global indexes $lidx$ to $ridx$, where both $A[lidx] = A[ridx] = val$ are the outermost occurrences of elements in its equal-range. For a tuple t in T_i, we write t^{val} to represent the value, as well as t^{lidx} and t^{ridx} to represent the left and right indexes.

To construct the representative sequence T_i, each processor P_i solves the ANSV problem on its local sequence A_i and identifies those elements which do not have a left or right match. The equal-range for each such element will be represented as a tuple in T_i. This representative sequence contains tuples in the same order as their respective elements appear in A_i and each equal range is represented only once.

When searching for the *nearest-smaller (NS)* or *nearest-nonlarger (NN)* value, matches found in the local sequence A_i are also the correct matches with respect to the global sequence A, since the matches found in A_i are necessarily *nearer* than any elements on

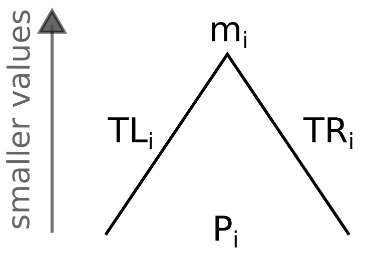


Figure 3.2: *Peak* for T_i.

other processors. Solving the distributed problem with regards to elements represented in T_i is thus sufficient. However, for the *FE* variant a match found in A_i might not be the furthest of its corresponding *equal-range*. The truly furthest element in this *equal-range* might be on another processor. In this case, the solution to the representative problem will be used to solve the original problem during a post-processing step.

Let $m_i = \min\{a_j \in A_i\}$ be the minimum value in A_i, and let TM_i be the tuple representing its equal-range. Any element $a_k > m_i$ without a left match in A_i must come before the leftmost occurrence of m_i, since otherwise this leftmost m_i would be a valid left match for a_k. Similarly, any element without a right match must come after the rightmost occurrence of m_i. We can thus represent the sequence T_i as a concatenation of three sequences $T_i = TL_i \parallel TM_i \parallel TR_i$, where TL_i contains the tuples for elements without left matches and TR_i those for elements without right-matches. For the example from Figure 3.1,

$$T_i = [\underbrace{\begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ 6 \end{pmatrix}}_{TL_i}, \underbrace{\begin{pmatrix} 1 \\ 7 \\ 7 \end{pmatrix}}_{TM_i}, \underbrace{\begin{pmatrix} 2 \\ 9 \\ 9 \end{pmatrix}}_{TR_i}]$$

Lemma 3.2.1. *The sequence T_i is bitonic, where TL_i is strictly decreasing and TR_i is strictly increasing with respect to their values.*

Proof. Assume this is not true for TL_i. Then there exists a tuple t_j in TL_i such that $t_{j-1}^{val} < t_j^{val}$. If so, t_{j-1}^{val} is a valid left match for t_j^{val}, and t_j cannot be in TL_i. A similar argument proves this property for TR_i. □

Due to this bitonic nature, T_i can be represented as illustrated in Figure 3.2. We represent the minimum m_i as the topmost element. Larger elements are shown towards the bottom. Hence, the decreasing sequence TL_i is represented as a rising line starting from the largest element without a left match and ending in the minimum m_i. The sequences T_i across all processors i can thus be represented as a series of such peaks as shown in Figure 3.3.

3.2.3 Remote matches

The following properties are stated with respect to right matches, but hold true symmetrically for left matches as well.

Lemma 3.2.2. *Let $a_k \in A_i$ be any element without a right match in A_i, but which does have a right match in the global sequence A, i.e., $r(k) \neq \perp$. Then a_k's right match is represented in T_j for some processor $j > i$.*

Proof. As a_k does not have a right match in A_i but has a match in A, its right match $a_{r(k)}$ must be in A_j for processor $j > i$. By the definition of the right match, there cannot be any element smaller than $a_{r(k)}$ between a_k and $a_{r(k)}$. Therefore, there cannot be any element in A_j both to the left of and smaller than $a_{r(k)}$. Consider the *equal-range* for $a_{r(k)}$. The leftmost element in this *equal-range* does not have a left match in A_j for any of the variants *NS*, *NN*, and *FE*, and thus is included in TL_j in all three cases. The match $r(k)$ for a_k is then either the leftmost (*NS*, *NN*) or rightmost (FE) element in $a_{r(k)}$'s equal-range. Due to how we defined the tuples inside TL_j, both these indexes are saved inside TL_j. □

3.2.3 Communication Ranges

For each processor P_j to the right of P_i ($i < j$), we define multiple ranges in T_i, namely I_{ij}, M_{ij} and L_{ij}. Each of these ranges can be represented by a starting and ending index with respect to the sequence T_i. Symmetrically, P_j will compute ranges I_{ji}, M_{ji}, and L_{ji} for every processor to its left. We assume that every processor knows the minimum m_j of every other processor.

The ranges are computed in such a way that matches for I_{ij} and I_{ji} can be fully solved on either P_i or P_j. This allows us to choose freely whether to send I_{ij} from P_i to P_j or I_{ji} from P_j to P_i, and thus allows to optimize communication volume or other metrics such as local workload.

The ranges M_{ij} and L_{ij} contain at most 2 tuples each. Unlike I_{ij}, these are not solved remotely, but instead are always communicated both ways, i.e., M_{ji} and L_{ji} will always be available on P_i. We show how these ranges are computed in Algorithm 3. Next we formally define these ranges and prove some required properties.

In-range I_{ij} We define the *in-range* I_{ij} for i with respect to j as the range of tuples in TR_i for which the right match must be in T_j on processor P_j. We distinguish between two cases:

Case $j = i + 1$**:** The *in-range* I_{ij} contains all tuples t in TR_i for which $\max(m_i, m_j) < t^{val}$.

Case $j > i + 1$**:** The *in-range* I_{ij} contains all tuples t in TR_i for which: $\max(m_i, m_j) < t^{val} < \min\{m_l \mid i < l < j\}$

Since TR_i and TL_i are monotonic sequences with respect to their value, the ranges defined above are contiguous within T_i.

We similarly define the *in-range* for left matches, such that for each pair of processors i and j, I_{ij} and I_{ji} are defined symmetrically with respect to T_i and T_j. Figure 3.3 illustrates the concepts of *in-ranges* I_{ij} and boundary ranges M_{ij}, which are defined next.

M_{ij} The range M_{ij} contains possible left matches for tuples in I_{ji} which have smaller value that those in I_{ij}. Specifically, we define M_{ij} as the range of at most two tuples from

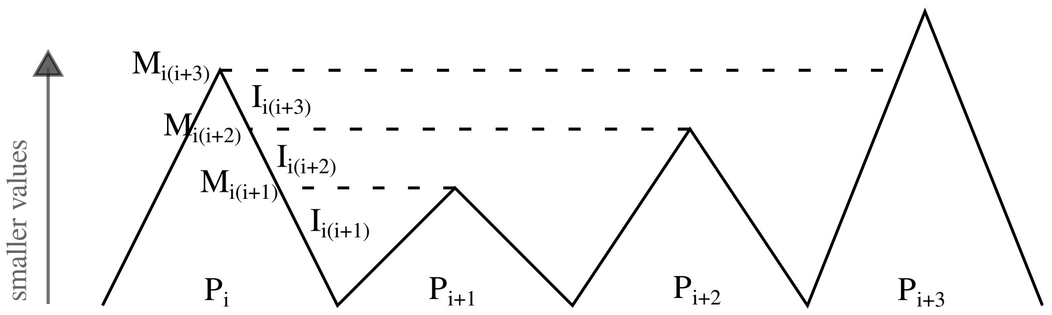


Figure 3.3: Peak representation for right matches for processor P_i. Ranges I_{ij} and M_{ij} are represented.

$TM_i \cup TR_i$, starting with the tuple for which $t^{val} = \max(m_i, m_j)$, and ending with the next smaller one if m_i is the smaller of the two.

ALGORITHM 3: Matching ranges

```
Input: Sequence $T_i$ on each processor $P_i$, and array of all processor minimums
        $[m_0, m_1, \ldots, m_{p-1}]$
Output: Ranges $I_{ij}$, $M_{ij}$ and $L_{ij}$ for all right processors $j > i$.
1  $upper \leftarrow \infty$, $LB \leftarrow \emptyset$
2  for $j = i+1, \ldots, p-1$ do
3      if $m_j < upper$ then
4          // determine in-range
5          $I_{ij} \leftarrow$ range of tuples $t$ with $\max(m_i, m_j) < t^{val} < upper$
6          $L_{ij} \leftarrow LB$
7          $M_{ij} \leftarrow$ range of tuples $t^{val} = \max(m_i, m_j)$ and next smaller
8          $LB \leftarrow M_{ij}$
9          $upper \leftarrow m_j$
10     else
11         $I_{ij} \leftarrow \emptyset$, $M_{ij} \leftarrow \emptyset$
12         $L_{ij} \leftarrow LB$
13     end
14 end
```

Algorithm 3 illustrates how to compute the ranges I_{ij}, M_{ij}, and L_{ij} for all right processes $j > i$. To compute these ranges for processes left of i, we follow a symmetric protocol by starting the for loop at $j = i - 1$ and iterating backwards till $j = 0$. The range L_{ij} computed inside the algorithm is equal to a range M_{ik}, where k is a processor between i and j, namely the one for which m_k is the minimum for all processors between i and j.

Matches for tuples in I_{ij} Each tuple in the range I_{ij} has its right match on processor j in either I_{ji} or in M_{ji}, and symmetrically each tuple in I_{ji} has its left match on processor i in either I_{ij} or M_{ij}. We prove this property for *NS* and *NN* matches. For *FE* the match is no smaller than the values in M_{ji} (M_{ij}), however, the furthest element of that value might be on a later processor $q > j$. In this case the match is in L_{qi}.

Proof. Take any tuple t in the *in-range* I_{ij}. By definition, its value is smaller than the minimums of all processors between i and j, thus its match cannot be on these processors. Due to the symmetric upper bound on the values in I_{ij} and I_{ji}, the right-match for t cannot be to the left of the first, leftmost tuple in I_{ji}. Furthermore, t^{val} is larger than both m_i and m_j, and thus larger than any value in M_{ij}. Therefore, a valid right match for t when looking for the *nearest-smaller (NS)* or *nearest-nonlarger (NN)* must be in either I_{ji} or M_{ji}. There is one notable exception to this when looking for the *nearest-nonlarger-furthest-equal (FE)* right match for t: when $min(M_{ij}) = m_j$ and the right match for t has value m_j. In this case the global equal-range for m_j might end on another processor P_q with $q > j$ and $m_q \leq m_j$. The valid match will then be in M_{qj}. There cannot be any processors between P_i and P_q with a minimum $< m_q$, since otherwise t could not have a match on P_q. Hence, $L_{qi} = M_{qj}$, and t's match will be available on P_i as well as P_j. $\square$

Now that we know how to find matches for the *in-ranges*, we are left with finding matches to those tuples not contained in any *in-ranges*, i.e., those elements equal to the processor minimums. For any processor $j > i$, this is the tuple in TR_i either with value m_j or the overall minimum m_i. In the first case, the match will be either m_j (NN), or in L_{qi} for a processor $q > j$, whereas in the second case, the right match will be in L_{qi} for some $q \geq j$.

3.2.3 The Algorithm

Algorithm 4 shows the high level steps of the resulting algorithm. First, we solve the ANSV problem locally for the sub-sequence A_i, and create the representative sequence T_i. This

ALGORITHM 4: Generalized parallel ANSV

```
Input: Sequence $A_i$ on each processor $P_i$, $left\_type, right\_type \in \{NS, NN, FE\}$
Output: Left and Right matches $L_{left\_type}$, $R_{right\_type}$
1  // solve ANSV locally for $A_i$ and return sequence $T_i$
2  $T_i = (TL_i, m_i, TR_i) \leftarrow \texttt{local\_ansv\_unmatched}(A_i)$
3  $[m_0, m_1, \ldots, m_{p-1}] \leftarrow \texttt{allgather}(m_i)$
4  // get the ranges for all $j$ with Algorithm 3
5  $[\ldots, I_{ij}, M_{ij}, L_{ij}, \ldots] \leftarrow \texttt{get\_ranges}(T_i, [m_0, m_1, \ldots, m_{p-1}])$
6  // all-to-all exchange tuples in ranges $L_{ij}$ and $M_{ij}$
7  $L_{recv} = [L_{0i}, \ldots, L_{(p-1)i}] \leftarrow \texttt{alltoall}([L_{i0}, \ldots, L_{i(p-1)}])$
8  $M_{recv} = [M_{0i}, \ldots, M_{(p-1)i}] \leftarrow \texttt{alltoall}([M_{i0}, \ldots, M_{i(p-1)}])$
9  // communicate in-ranges
10 $I_{recv}[I_{0i}, \ldots, I_{(p-1)i}] \leftarrow \texttt{send\_in\_ranges}([I_{i0}, \ldots, I_{i(p-1)}])$
11 // serially solve local sequences and received in-ranges
12 $T_i, I_{recv} \leftarrow \texttt{ansv\_merge}(T_i, I_{recv}, M_{recv}, L_{recv})$
13 // return solutions to received in-ranges
14 $\texttt{return\_in\_ranges}(I_{recv})$
```

step can easily be done in linear $O(\frac{n}{p})$ time using a stack based approach. Next, we use *allgather* to collect the minimums m_i from all processors. Then we can determine the begin and end indexes for all ranges in T_i: I_{ij}, M_{ij}, L_{ij} by using Algorithm 3 both for left and right matches. This is done with a simple linear scan of TR_i and TL_i, and thus possible in $O(\frac{n}{p} + p)$ time. Once the ranges are determined, we send all tuples which fall into M or L ranges to their target processor using `all-to-all` collective communications. This takes $O(p)$ time, since each processor sends and receives at most $O(p)$ tuples in this step. Next, we send and receive the *in-ranges* in a collective step the procedure for which is described in more detail below. The received elements are utilized for solving the matches for T_i, as well as the matches in received *in-ranges* I_{ji}. Solving these matches can be done with a linear time merge-like operations, since both T_i and the received *in-ranges* are either strictly increasing or decreasing. As a final step, we return the solutions for received *in-ranges* to their original processors.

Communication of *in-ranges* The matches for elements in the ranges I_{ij} on P_i and I_{ji} on P_j can be solved on either P_i or P_j. If we naively send all elements in I_{ij} to P_j and

I_{ji} to P_i, a single processor could be required to receive $O(n)$ elements in the worst-case. Berkman *et al.* [39], as well as all parallel algorithms based on it [40, 46], formulate the communication so that each processor receives data from at most two other processors, one from the left and one from the right. The number of received elements can thus be bound by $O(\frac{n}{p})$.

We propose to further minimize the communication volume. If every processor P_i knows the size of I_{ji} for all processors j, we can dynamically decide how best to communicate these elements. To minimize the per process and global communication volume, we always send the smaller of the two ranges I_{ij} and I_{ji}. This minimizes the communication cost for both sending the local elements, as well as receiving the solutions.

Complexity The total complexity for the algorithm is dominated by the communication of *in-ranges*. Since each processor sends and receives at most $O(\frac{n}{p})$ elements, the time complexity for the generalized ANSV algorithm is $O(\frac{n}{p} + p)$.

3.3 Parallel Suffix Tree Construction

In this section, we show how to construct suffix trees from suffix and LCP arrays on distributed memory parallel computers by utilizing our algorithm for the generalized ANSV problem.

3.3.1 Prerequisites

We briefly discuss how suffix arrays and LCP arrays are related to suffix trees, and the properties used for construction.

Each leaf of a suffix tree represents a specific suffix of the string, and leaves appear in their lexicographical order. A suffix array therefore represents the leaves of the suffix tree. Each entry in the LCP array corresponds to a specific internal node in the suffix tree, as explained below. However, multiple entries in the LCP array might correspond to the same internal node.

Each internal node v of a suffix tree is associated with its *string-depth* t, which is the length of the longest common prefix (lcp) shared by all suffixes in its subtree. As such, the minimum lcp over all pairs of consecutive suffixes in v's subtree is also t. The node v splits the set of its suffixes into at least two subsets, i.e., its child nodes $u_0, u_1, \ldots, u_{d(v)-1}$, where the suffixes in each subtree under u_j share a common prefix $> t$ (see Figure 3.4). Since the LCP array contains the length of the lcp for each pair of consecutive suffixes, the subtree for v corresponds directly to a range in the LCP array with values all $\geq t$.

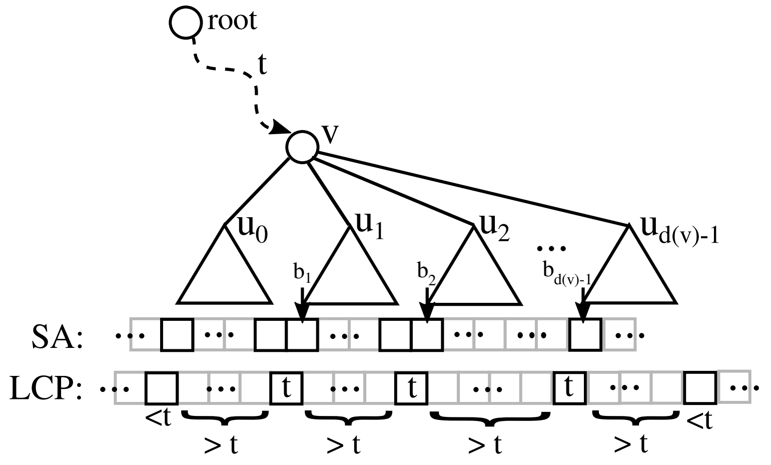


Figure 3.4: Suffix Tree structure and LCP for a subtree starting at an internal node v with child nodes $u_0, u_1, \ldots, u_{d(v)-1}$.

More specifically, for any internal node v with $d(v)$ children, there are exactly $d(v) - 1$ pairs of consecutive suffixes $SA[b_j - 1]$ and $SA[b_j]$ in this range for which $LCP[b_j] = lcp(SA[b_j - 1], SA[b_j]) = t$. These are all the positions for which b_j is the first suffix in its subtree u_j with the exception of the first subtree u_0 (see Figure 3.4). We can thus identify each internal node v of a suffix tree uniquely by b_1, the index of the first suffix in the second child u_1 of v. We denote this leftmost element of value t, the *representative* for v.

Every position in the `LCP` array corresponds to an internal node, unless that position is not the representative for the node of its value. A position in the LCP array is not a representative if its nearest non-larger value to the left is of equal value.

From the position of the representative in the LCP array b_1, let $l(b_1)$ be the position

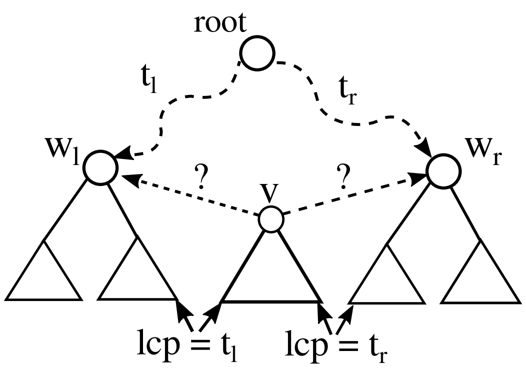


Figure 3.5: The two possible choices for the parent node for v.

of the nearest element to the left which is smaller than t, and similarly let $r(b_1)$ be the index of the nearest smaller element to the right. Let the values of the `LCP` array at these positions be $t_l = \texttt{LCP}[l(b_1)]$ and $t_r = \texttt{LCP}[r(b_1)]$, then $t_l < t$ and $t_r < t$. These two positions correspond to the beginning and end of the subtree for v, and each corresponds to an internal node, say w_l and w_r respectively (see Figure 3.5). Out of these two nodes, the parent for v is the node with which it shares a longer common prefix, which leads to the following property:

Case $t_l < t_r$:

The parent for v is the node w_r. The index $r(b_1)$ and the LCP entry at that position is the representative for this node, since there can not be any entry further left equals to t_r.

Case $t_l > t_r$:

In this case, the parent for v is the node w_l. The LCP position $l(b_1)$ might not be the representative for this node, if the node has more than two children. The representative is the leftmost element of value t_l, given that no value in between is smaller.

Case $t_l = t_r$:

In this case, $w_l = w_r$ and the representative of value t_l is towards the left.

For the last two cases, the index for the parent node is the representative for the nearest smaller element to the left. Iliopoulos and Rytter [43] calculate a table `Leftmost` for determining the representative for each element of the LCP array. They give an algorithm for calculating this table in $O(n \log n)$ work and $O(\log n)$ time on n processors.

We instead propose to directly solve this problem by using our generalized *ANSV* solution. Specifically, we calculate the *nearest-nonlarger-furthest-equal (FE)* element to the left while simultaneously determining the *nearest-smaller (NS)* element to the right. Note that our ANSV algorithm was formulated to calculate the matches to both sides simultaneously while supporting different matching variants for both sides.

3.3.2 The Algorithm

The high-level steps of our parallel distributed algorithm for the construction of suffix trees are shown in Algorithm 5. The algorithm assumes that all inputs are distributed equally across processors with $\frac{n}{p}$ elements per process. This includes the string S, suffix array SA, as well as the LCP array.

3.3.2 Parents of internal nodes

We can now formulate how to determine the parent for each internal node by looking only at its LCP value and the left and right matches returned by the generalized *ANSV* solution. We create an array E of edges $(j, parent(j))$, where each edge defines the index for the parent of the node corresponding the $LCP[j]$.

For each position j in the LCP array, we determine the values of the left and right matches $LCP[l_{FE}(j)]$ and $LCP[r_{NS}(j)]$. We distinguish between multiple cases:

If $LCP[l_{FE}(j)] = LCP[j]$**:**

In this case $LCP[j]$ is not the representative for its node, because its not the leftmost element of its value within the range to the nearest smaller. Thus, we can simply skip this element, since it is not a valid node.

Else If $LCP[l_{FE}(j)] < LCP[r_{NS}(j)]$**:**

This corresponds to the case above where $t_l < t_r$, and the parent for the node at j is

ALGORITHM 5: Parallel Distributed Suffix Tree Construction

```
Input: String $S$, Suffix and LCP arrays: $SA$ and $LCP$ for $S$, all equally distributed with $\frac{n}{p}$
       elements per process.
Output: Suffix tree for $S$.
// get the ANSV matches for the LCP in $O(\frac{n}{p} + p)$
$L_{FE}, R_{NS} \leftarrow$ gANSV($LCP$, left=FE, right=NS)
// iterate through all local elements in $O(\frac{n}{p})$
$E$ : array of edges $(j, parent(j), SA[j] + LCP[parent(j)] + 1)$
// get parents for internal nodes
for $j = \frac{n}{p}i, \ldots, \frac{n}{p}(i+1) - 1$ do
    if $LCP[L_{FE}[j]] = LCP[j]$ then
        // this is not a representative
        continue
    else if $LCP[L_{FE}[j]] < LCP[R_{NS}[j]]$ then
        $parent \leftarrow R_{NS}[j]$
    else
        $parent \leftarrow L_{FE}[j]$
    end
    $E$.insert($j, parent, SA[j] + LCP[parent] + 1$)
end
// get parents for leaf nodes
for $j = \frac{n}{p}i, \ldots, \frac{n}{p}(i+1) - 1$ do
    if $LCP[j] < LCP[j+1]$ then
        $parent \leftarrow j + 1$
    else if $LCP[L_{FE}[j]] = LCP[j]$ then
        $parent \leftarrow L_{FE}[j]$
    else
        $parent \leftarrow j$
    end
    $E$.insert($j + n, parent, SA[j] + LCP[parent] + 1$)
end
// send tuples in $E$ to the processor which contains the parent
$E \leftarrow$ alltoall($E$, target=$P[j]$)    // $O(\frac{n}{p} + p)$
// get character $S[SA[j] + LCP[parent] + 1]$ for each edge
$C \leftarrow$ bulk-query($E.cpos$)
// create internal nodes (alternatively use hash-table)
$nodes \leftarrow$ array of size $(\sigma + 1)\frac{n}{p}$, init to $-1$
for each $(idx, parent, char)$ in $E$ and $C$ do
    $nodes[(\sigma + 1)parent + char] \leftarrow idx$
end
```

the nearest-smaller to the right: hence $parent(j) = r_{NS}(j)$. Since this is the leftmost element of value t_r in the equal-range for t_r, this is also the representative.

Else If $LCP[l_{FE}(j)] > LCP[r_{NS}(j)]$**:**

This corresponds to the case above where $t_l > t_r$. Since the *FE* variant returns the leftmost element as the representative for its equal-range, the representative for w_l as well as the parent for j is given by: $parent(j) = l_{FE}(j)$.

Else If $LCP[l_{FE}(j)] = LCP[r_{NS}(j)]$**:**

In this case, $parent(j) = l_{FE}(j)$, since both matches correspond to the same node, the representative of which is to the left given by $l_{FE}(j)$.

For each position j in the LCP, we insert the edge $(j, parent(j))$ into E.

3.3.2 Parents of leaf nodes

For the leaf nodes, represented by the suffix array, we can determine the parent in a similar fashion. In order to distinguish between internal and leaf nodes, we offset the indexes of leaf nodes by n, such that the leaf node $SA[j]$ is represented by index $n + j$. For each position in the suffix array $SA[j]$, we can determine its parent independently. The suffix $SA[j]$ shares a parent node with either its left ($SA[j - 1]$) or right ($SA[j + 1]$) neighbor, specifically the one with which it shares a longer prefix. Since the length of the prefix matches between these is given by $LCP[j]$ and $LCP[j + 1]$ respectively, the parent node is the representative for the larger of the two. Here we distinguish only between two cases:

Case $LCP[j] \geq LCP[j + 1]$**:**

The parent for $SA[j]$ is the the left LCP. Since this may not be the representative of its equal-range, we check whether $LCP[l_{FE}(j)] = LCP[j]$. If this is the case, the parent for the leaf $SA[j]$ is $l_{FE}(j)$, otherwise it is $parent = j$. We insert the edge $(n + j, parent)$ to E.

Case $LCP[j] < LCP[j + 1]$**:**

The parent for leaf j in this case has the index $j+1$, since the right LCP is necessarily the leftmost of its value and thus its own representative. Thus, we insert $(n+j, j+1)$ into E.

3.3.2 Creating internal nodes

So far we have created an array E of edges pointing for each position to their parent index. For pattern matching applications, instead of having parent pointers, each internal node should point to its children.

Conceptually, an internal node is a small lookup table, which for each character contains the index of the child node, which either is the index of an internal node (index into the node array) or the index of a leaf node (index into the suffix array).

Additionally to each character in the alphabet Σ, the internal node needs an entry for the end of string character $\$$. For the case of a single input string S, all suffixes are unique. Thus, the pointer for $\$$ always points to a leaf node. In this case, the internal node is a lookup table of size $\sigma + 1$.

In the generalized case of $m \geq 2$ input strings, suffixes are no longer unique. Due to how we defined the LCP array in this case, a group of identical suffixes will appear as a contiguous segment in the LCP and suffix arrays. The LCP value for all of these will be the length of the suffixes. The first character at which they differ are the virtual $\$_i$ end of string characters. The internal node corresponding to this segment of the LCP array then can have more than $\sigma + 1$ children. In this generalized case, we use lookup tables of size $\sigma + 2$ instead, with one entry for each character $c \in \Sigma$ and two entries for the $\$$ character(s), giving the index range for the identical suffixes in the suffix array as a start and end index.

Conceptually, the internal nodes are represented as a size $(\sigma + 2)n$ block distributed array. This approach works well for very small alphabets, but becomes prohibitively expensive in terms of memory usage for larger alphabets. Using a hash-table on each process allows to store the the internal nodes in a more space-compact format, at the cost of lookups being only expected time $O(1)$. We still conceptually split the nodes across processors according to their index as done in the array format.

Creating the lookup table requires inverting the edges E, as well as reading the first character associated to each edge. For a node j with parent index $q = parent(j)$, all suffixes in the subtree of q share the common prefix $LCP[q]$, but each child of q differs at the next character $LCP[q] + 1$. For each suffix in the subtree for j this character is identical, thus we can chose any suffix from j's subtree, e.g., $SA[j]$. We label each edge $(j, parent(j))$ in E with the character at position $cpos(j) = SA[j] + LCP[parent(j)] + 1$ of the input string. Note that $LCP[parent(j)]$ is the value of the left or right match used to determine the parent index itself. Therefore, $cpos(j)$ can be calculated locally for each j just after determining the index of the parent.

Inverting the edges requires communication of the edges $(j, parent(j), cpos(j))$ to the processor which contains the index $parent(j)$. We exchange all edges for remote parents in one collective communication step.

Since each processor contains $\frac{n}{p}$ elements and internal nodes, it will send at most $O(\frac{n}{p})$ edges in this communication step. Each internal node has at most $\sigma + 2$ outgoing edges. Therefore, the communication complexity of this many-to-many communication is thus bound by $O(\sigma \frac{n}{p} + p)$ in the very worst case. In practice, the expected number of edges per process is $O(\frac{n}{p})$. Furthermore, the number of edges actually sent to another process is much smaller, since tuples are send to one of the two nearest smaller matches. The communication thus follows the same structure as for the ANSV problem, for which the expected number of elements communicated is logarithmic, the majority of which are exchanged between direct neighboring processors.

3.3.2 Edge labels

For each internal node, we now have all its outgoing edges, i.e. the index for each child node. To determine which position in the lookup table the child index should be written to, we need the character at the position $cpos(j)$ for each child index j.

In the case of a single input string S, the required character is given by $S[cpos(j)]$ if $cpos(j) < n$. Otherwise, the required character is \$. Since the input string S is distributed across processors, accessing these characters corresponds to a random access reads into the distributed string. We process these in a bulk fashion as described below.

In the generalized case of $m \geq 2$ input strings, we indexed the character positions assuming that all strings are concatenated together without any separators. Note, in this virtual representation we can access all characters at $cpos(j)$ directly at the global position $cpos(j)$, unless the suffix was equal to another. In the latter case, the $cpos(j) = SA[j] + LCP[parent(j)] + 1$ points one character past the corresponding string for suffix $SA[j]$, i.e., at the first character of the following string. Requests to these positions should return the \$ character instead. To do this, we simply change the first character of each string to \$ prior to processing the queries for $cpos(j)$. Note, the first character of each string is required only for the root node. We know that the root node has to have children for all possible characters that appear in the strings. Therefore, we can handle the root node as a special case, and don't require looking up the characters at $cpos(j)$ for edges from the root node.

In either case of a single or multiple strings, we need to perform what are essentially random access reads into the distributed string. We answer these reads by generating requests for all required positions which are not available locally.

In the worst-case, $O(n)$ requests could target the same processor. To mitigate this problem, we sort all requests based on the index requested using parallel distributed integer sorting. We then send only unique requests to their target processor, return the requested characters to the original position of the requests in the sorted range, set the returned character to all requests targeting the same index, and then return all requests to their origin process.

Note that this procedure is only needed in the worst-case. If the maximum number of requests targeting a single processor is less than $\leq c\frac{n}{p}$ for a small constant c, we can simply send each request to its target processor directly and skip the sorting of requests. This results in a much better runtime in the expected case. The overall complexity for handling the requests is thus dominated by the complexity for parallel integer sorting: $O(\frac{n}{p} + p)$.

3.4 Experiments and Results

3.4.1 Implementation

We implemented our algorithm for the generalized *ANSV* problem as well as suffix tree construction using *C++11* and *MPI*. Our code is thoroughly tested and available as fully Open Source on GitHub[1]. For experiments reported here, we compiled our code using `gcc 5.2` and `MVAPICH2 v 1.7` using the optimization flags `-O3 -march=native`.

3.4.2 Systems and Data sets

We experimentally evaluated our algorithm and its implementation on two different systems. The first system *Cyence* is an Intel Xeon Infiniband cluster, where each node has two 2.0 GHz 8-core Intel E5 2650 processors and 128 GB RAM. The nodes are connected via a QDR (40Gbit) Infiniband interconnect. The second system *FatNode*, is a single shared memory node with four 2.1 GHz Intel Xeon E7-8870 18-core processors and 1 TB of RAM. We use this system to compare against Shun's shared memory implementation [44].

The human genome, with an approximate length of 3 billion base pairs, is commonly used as a benchmark for comparing the performance of parallel suffix array and tree construction algorithms [36, 37, 38, 44, 12]. In addition to benchmarking against other methods using the human genome, we evaluate the scalability of our approach on the much larger Pine (*Picea abies*) genome, which has a length of over 12 billion base pairs [35]. The alphabet size of genomes is 4 (nucleotides A, C, G, and T).

[1] github.com/parbliss/psac

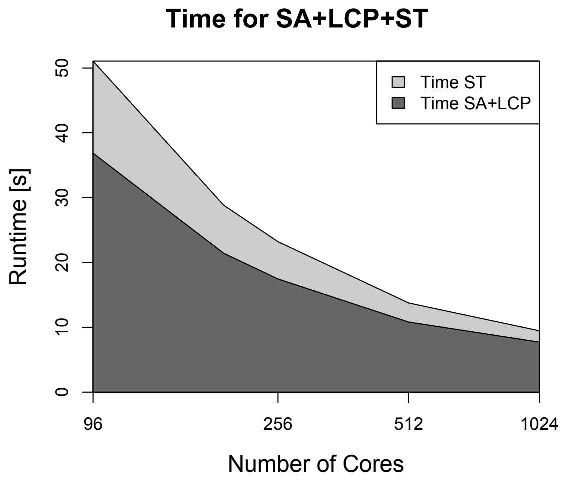


Figure 3.6: Cumulative runtime for constructing the Suffix (SA) and LCP Arrays via [12] and subsequently the Suffix Tree (ST) for the human genome on up to 1024 cores on *Cyence*.

3.4.3 Performance of ST Construction

On the *Cyence* system, we ran experiments on up to 1024 cores (64 nodes) for the human genome. In Figure 3.6, we show the run-time composition for constructing the suffix tree for the human genome from scratch, i.e., the time taken for first constructing the suffix and LCP arrays, as well as the time required to construct the suffix tree from these arrays including the time required for solving the All-Nearest-Smaller-Values problem. We observe that the latter part represents only a minority of the total runtime. On 1024 cores, it take less than 2 seconds within an overall run-time of 9.5 seconds.

3.4.3 Comparison with prior State-of-the-art

Due to unavailability of any source code, we could not experimentally compare the performance of our code against that of other distributed memory approaches using the same system. Below, we provide a table (Table 3.1) of previously reported results for suffix tree construction of the human genome, and compare them to our experimental results (bold). It appears the performance gains observed in the table exceed the advantage gained by hardware improvements.

Shun and Blelloch's [44] algorithm takes a similar approach to ours, but is developed for the PRAM model. The authors provide a competitive implementation for shared memory machines, against which we compare the performance of our MPI based implementation on the *FatNode* large shared memory machine. For this implementation we show both the reported time, as well as the time measured on *FatNode*. As can be seen, our algorithm outperforms the shared memory algorithm on a shared memory system.

Algorithm	**System**	**Cores**	**Time**
WaveFront [36]	IBM BG/L	1024	15 min
ERA [37]	16x Intel 2-core nodes	32	13.7 min
PCF [38]	MareNostrum	172	7 min
Shun [44]	4x 10 core Intel E7-8870	40	168 s
Shun [44]	***FatNode***	**72**	**146 s**
Ours	***FatNode***	**72**	**63 s**
Ours	***Cyence***	**1024**	**9.5 s**

Table 3.1: Prior results reported for the human genome and our experimental runtimes (bold).

3.4.3 Runtime Analysis and Scaling

Next, we analyze the runtime and scaling behavior of our suffix tree construction algorithm on the larger pine genome. To focus on contributions in this paper, this analysis considers suffix tree construction from suffix and LCP arrays. In Figure 3.7, we show the run-time of various components of the algorithm as the number of cores is varied from 256 (16 nodes) to 1600 (100 nodes), the maximum allocation on *Cyence*. We started the assessment at 16 nodes, because the peak memory consumption for this large genome exceeds the memory capacity of 8 nodes (reason why distributed memory algorithms are necessary). Determining the tree structure via solving the *ANSV*, and combining left and right ANSV to compute the parent edges *Calc.Parent*, together take less than 25% of the total runtime. The majority of the time ($\approx 60\%$) is spent in querying the characters for each edge (*EdgeChar*). We split this time into local work and the required all-to-all communications. Finally, creating the suffix tree nodes from the edges and their associated characters takes less than 15% of the overall time.

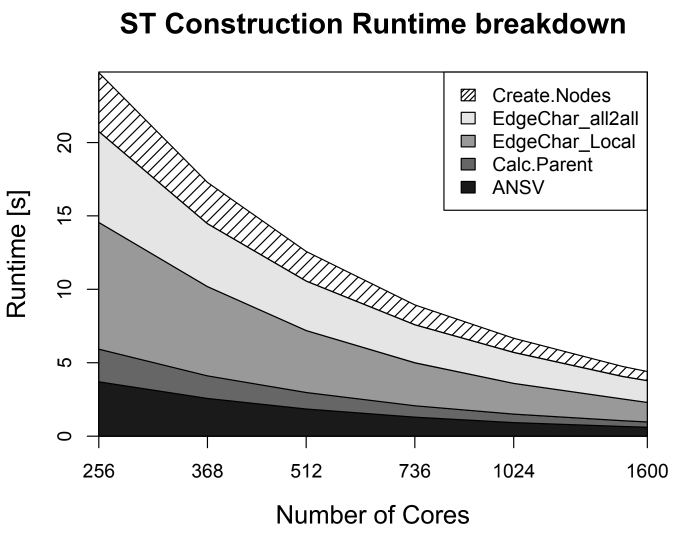


Figure 3.7: Breakdown of the runtime for different sections of the Suffix Tree construction of the Pine genome on up to 100 nodes of *Cyence*.

Results of strong scaling experiments for the pine genome are illustrated in Figure 3.8. The speedup is measured relative fixed to the time taken by 256 cores.

Our implementation scales with 89.9% efficiency, reaching a relative speedup of over 5.6 when increasing the number of cores from 256 to 1600 (a factor of 6.25). The main contributory factor limiting our scaling is the two all-to-all communications required for retrieving the edge characters. We plot the relative speedup for just these two operations (*EdgeChar_alltoall*), as well as the speedup for suffix tree construction excluding just these two communication steps. The latter scales with efficiency of 102%, possibly due to cache effects, showing that the limitation factor for scaling lies exclusively within the MPI_Alltoall communication. As it would be impossible to design an algorithm without one of these operations (even if the solution is known, transforming distributed representation of input to the output itself requires an all-to-all operation), the performance of our algorithm is not only good, but close what is obtainable.

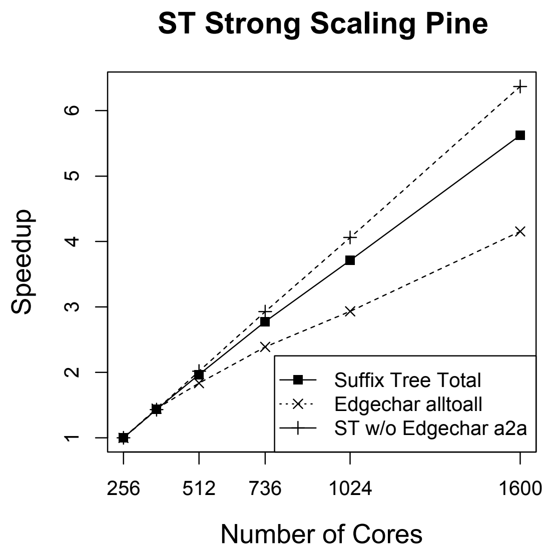


Figure 3.8: Strong scaling of the Suffix Tree construction of the Pine genome on up to 100 nodes of *Cyence*

3.5 Conclusions

In this Chapter, we presented a work-optimal distributed memory parallel algorithm for the construction of suffix trees. In contrast to the linear work performed by the algorithm, all previous distributed memory algorithms exhibit quadratic worst-case complexity. Our algorithm also improves prior state-of-the-art for distributed memory in terms of practical performance. We illustrate performance of the algorithm on the human genome, for which we construct the suffix and LCP arrays in 7.5 seconds from the genome, followed by construction of the suffix tree in less than 2 additional seconds, on 64-node dual 8-core Xeon CPU cluster. Furthermore, we demonstrate that our MPI based implementation performs better in shared memory than state-of-the-art shared memory algorithms, and can scale to a large number of cores in distributed memory.

CHAPTER 4
DISTRIBUTED ENHANCED SUFFIX ARRAYS

The previous chapters, we discussed the efficient distributed construction of suffix arrays and trees. However, we did not address how the indices can be efficiently queried in distributed memory.

It turns out that the classic sequential query algorithms for suffix arrays and LCP arrays do not generalize well to their distributed representation - as they would incur massive communication latency costs at almost every step.

Given a string S and its suffix array SA, sequentially we can find the range of occurrences of a pattern string P in S in $O(m \log n)$, where $n = |S|$ and $m = |P|$. Additionally using the LCP array, the time complexity for the query can be reduced to $O(m + \log n)$ [6].

Commonly the occurrences are returned as a range in the suffix array, however, if we wanted to also list all occurrences, then the time complexity of the query would also depend on the number of results. In this case we would write the complexity of the pattern search as $O(m \log n + \mathit{occ})$, or $O(m + \log n + \mathit{occ})$ respectively. In the following discussion, we assume this is implied, and we will state the complexity for finding the range of occurrences within the suffix array, and thus give the complexity without the dependence on *occ*.

Enhanced suffix arrays (ESA) were introduced by Abouelhoda *et al.* [10] and add additional data structures on top of the SA and LCP arrays, consequently supporting $O(\sigma m)$ pattern search - optimal for constant size alphabets. However, querying an SA, ESA, or compressed indexes such as the FM-index, all require random accesses to $O(n)$ sized data structures, which in distributed memory is highly inefficient and prohibitively expensive.

In this chapter, we first explain the design constraints for an efficient distributed solution and discuss the shortcomings of sequential query algorithms. Then, we derive and propose the *Distributed Enhanced Suffix Array (DESA)*, a novel distributed data structure allowing for efficient distributed querying. Further, we propose an efficient distributed algorithm to construct the DESA by modifying the distributed SA and LCP array construction algorithm from Chapter 2. Finally, we demonstrate the performance and scalability of our DESA implementation.

A majority of the content in this chapter is published in the following paper:

- P. Flick and S. Aluru, "Distributed enhanced suffix arrays: Efficient algorithms for construction and querying," in *SPAA'19 (under review)*

4.1 Motivation

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 arrays and suffix trees. Parallel suffix array construction has been studied for shared memory [29, 19, 48], as well as distributed memory [21, 23]. Many parallel algorithms for suffix tree construction exist: theoretical algorithms for PRAM [41, 42, 43], practical algorithms for shared-memory machines [44], and algorithms for external or distributed memory [36, 49, 37, 38].

State-of-the-art PRAM algorithms for suffix tree, suffix array, and FM-index construction were recently introduced by Shun and Blelloch [44] and Labeit *et al.* [48]. These algorithms perform extremely well on shared-memory systems, but do not generalize to distributed memory. Whereas the shared-memory parallel algorithms can randomly access the entire string, suffix tree, or array, in distributed memory each processor generally contains only a part of the whole input or index.

Our distributed construction algorithms for suffix arrays (Chapter 2) and suffix trees (Chapter 3) use 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 [21, 23, 36, 37, 38], 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.

However, no prior methods exists to efficiently query the distributed suffix arrays and trees in distributed memory while not exceeding $O(n/p)$ memory per processor. This is primarily because query algorithms require the whole input string S to be locally available on every processor. In this chapter, we present a data structure that allows efficient querying while not exceeding $O(n/p)$ memory.

4.2 Sequential pattern search

Given a *pattern* string $P = p_0 p_1 \dots p_{m-1}$ of length m, we are interested in finding if and where the string P occurs within a larger string S, where typically $n \gg m$.

In a suffix tree, this search is performed in a top-down traversal. 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 look-up of outgoing edges by character, the pattern search has complexity $O(m)$.

4.2.1 Suffix array search

Using a suffix array, the pattern search can be performed in $O(m \log(n))$ time [6]. Since the suffixes in a suffix array are sorted lexicographically, we can search for a given pattern using binary search where in each step we lexicographically compare the pattern P against the suffix at the given search location.

Algorithm 6 shows how the binary search works for finding the lower bound l_P of occurrences of P in the suffix array. The search compares the pattern lexicographically against sub-strings in S that start at positions given by the suffix array SA. Since the suffix

ALGORITHM 6: SA binary search algorithm [6]

```
Input: Pattern $P = p_0 p_1 \cdots p_{m-1}$, string $S$, and suffix array $SA$ of $S$
Output: smallest $l_P$ with $P \leq_{lex} S[SA[l_P]\ldots]$
1  if $P \leq_{lex} S[SA[0]\ldots]$ then
2      return $l_P \leftarrow 0$
3  else if $P >_{lex} S[SA[n-1]\ldots]$ then
4      return $l_P \leftarrow n$
5  else
6      $[l, r] \leftarrow [0, n-1]$
7      while $r - l > 1$ do
8          $mid \leftarrow \frac{l+r}{2}$
9          if $P \leq_{lex} S[SA[mid]\ldots]$ then
10             $r \leftarrow mid$
11         else
12             $l \leftarrow mid$
13         end
14     end
15     return $l_P \leftarrow r$
16 end
```

array is a permutation of $0, \ldots, n-1$, the access pattern $S[SA[mid]\ldots]$ (line 9) accesses the string S in random access fashion. There is no dependency or guarantee of the string position that will be accesses from one iteration to the next. This dependency on the entire underlying string is the reason this type of query algorithm does not work well when all data is distributed.

By using the LCP array in addition to the suffix array, the time complexity of the pattern search can be improved to $O(\log(n) + m)$ [6]. This is still dependent on n and thus worse than the complexity of pattern search in a suffix tree. This algorithm is also performing a binary search for the pattern, but additionally uses two arrays $LLCP$ and $RLCP$ which can be derived from the LCP array. These arrays allow to skip certain characters when comparing the pattern with suffix $SA[mid]$, such that at most $O(m)$ character comparisons are required during the entire binary search. The string is still accessed in the same location plus some offset c: $S[SA[mid] + c\ldots]$ (see Manber and Myers [6] for details), and thus suffers from the same problems for using this type of algorithm in distributed memory.

4.2.2 Enhanced suffix arrays

Enhanced Suffix Arrays (ESA) were introduced by Abouelhoda *et al.* as a space-saving replacement for suffix trees [10]. The ESA consists of the suffix array, LCP Array, an interval tree over the LCP array, and a *child-table*. Abouelhoda *et al.* showed how their data-structure supports the various processing modes of suffix trees. 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 [33], yielding a more space efficient ESA.

ALGORITHM 7: ESA query algorithm [33, 10]

```
Input: Pattern $P = p_0 p_1 \cdots p_{m-1}$
Output: interval $[l, r)$ for occurrences of $P$ in $SA$
1  $[l, r] \leftarrow [0, n]$
2  $c \leftarrow 0$,   found $\leftarrow$ true
3  while $found \wedge c < m \wedge l < r$ do
4      $[l, r] \leftarrow$ getChild$(l, r, P[c])$
5      if $[l, r] == \emptyset$ then
6          return'not found'
7      end
8      $\ell \leftarrow \min(\min_{i \in [l+1, r]} LCP[i], m)$
9      found $\leftarrow (S[SA[l] + c \ldots SA[l] + \ell - 1] == P[c \ldots \ell - 1])$
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 7 illustrates the forward search algorithms as performed by both Abouelhoda *et al.* [10], and Fischer and Heun [33]. 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

$SA[l], \ldots, SA[r-1]$ in the suffix array which have a c-length common prefix with P.

ALGORITHM 8: `getChild` function from Fischer and Heun [33]

```
Function getChild($(l, r, a)$)
    $i \leftarrow RMQ_{LCP}(l+1, r-1)$
    $\ell \leftarrow LCP[i]$
    repeat
        if $S[SA[l] + \ell] == a$ then
            return $[l, i-1]$
        end
        $l \leftarrow i$
        $i \leftarrow RMQ_{LCP}(l+1, r-1)$
    until $l = r \vee LCP[i] > \ell$;
    if $S[SA[l] + \ell] == a$ then
        return $[l, r]$
    else
        return $\emptyset$
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 8 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 $S[SA[l] + c...]$ in each iteration (Algorithm 7 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.

4.3 Distributed Enhanced Suffix Arrays

We propose the *Distributed Enhanced Suffix Array (DESA)* data structure to allow efficient querying in distributed-memory. The data structure consists of the distributed suffix array, LCP array, a *Range-Minimum-Query (RMQ)* data structure, and an additional character array L_c, which is formally defined in the next section. All these arrays and data structures are distributed with $O(n/p)$ elements per processor.

Our proposed DESA removes the requirement to read the suffix array SA and string S for every character in the pattern during querying, and as such no longer requires random accesses into the string S. This allows querying a subtree in the distributed representation while accessing purely local data.

4.3.1 Forward-Search

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.

A key idea is that 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 9), 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 7 of Fischer and Heun [33] as follows. First, we remove the $found$ check, which compares $\ell - c$ additional characters in each iteration (Algorithm 7, 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 9 illustrates the steps of the forward-search algorithm.

ALGORITHM 9: Global view of DESA query algorithm

```
Input: Pattern $P = p_0 p_1 \cdots p_{m-1}$
Output: interval of $P$ in $SA$
1 $c \leftarrow 0$
2 while $c < m \wedge l < r$ do
3     $[l, r] \leftarrow$ 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     found $\leftarrow (S[SA[l] \ldots (SA[l] + m - 1)] == P[0 \ldots (m-1)])$
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 8, Line 5: $S[SA[l] + \ell]$ is identical to $S[SA[j] + \ell]$ for all $j = l, l+1, \ldots, i-1$. 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 8 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.

ALGORITHM 10: DESA `getChild` function

```
Function $getChild(l, r, a)$
    $i \leftarrow RMQ_{LCP}(l + 1, r - 1)$
    $\ell \leftarrow LCP[i]$
    repeat
        if $L_c[i] == a$ then
            return $[l, i - 1]$
        end
        $l \leftarrow i$
        $i \leftarrow RMQ_{LCP}(l + 1, r - 1)$
    until $l = r \vee LCP[i] > \ell$;
    // return last child interval
    return $[l, r]$
end
```

Using the L_c character array, we arrive at the `getChild()` function as shown in Algorithm 10. 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 9) still works as intended, because we allow false-positives during the traversal itself, and check against the underlying string at the 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.

4.3.2 Distribution of Subtrees

The suffix array and LCP array construction algorithms from Flick and Aluru return these arrays block distributed across processors. This type of distribution is useful during con-

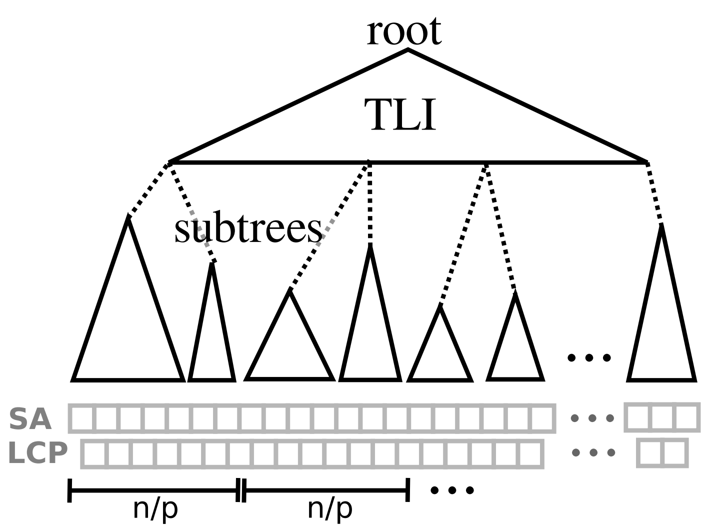


Figure 4.1: Distribution of subtrees and Top-Level Index (TLI).

struction, 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 queries 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 4.1 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).

4.3.2 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 [6].

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 large enough, 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. For example, for DNA alphabets and with $q = 10$, the lookup table will have a modest size of $8\texttt{MB}$, and split the suffix index into over 1 million subtrees - generally more than enough to partition the trees onto a very large number of processors while maintaining good load balance.

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.

The TLLT works well in practice for real world inputs, however, for some inputs this fixed top-level lookup table with constant q can fail to bring subtrees below the $O(n/p)$ size.

In that case, a more dynamic partitioning and distribution scheme is required.

4.3.2 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 is the largest subtree with size $\leq \frac{n}{cp}$, where c is a parameter controlling for load imbalance. Here, *largest* means that the parent of the subtree exceeds the threadhold size of $\frac{n}{cp}$.

The main challenge in creating the TLDT is to efficiently identify the largest subtrees with size $< \frac{n}{cp}$. We accomplish this by using the generalized *All-Nearest-Smaller-Values (ANSV)* algorithm (see Chapter 3). For every LCP position (corresponding to internal nodes of the ST), this algorithm can be used to find its subtree interval (and therefore its size) by identifying the left- and right-most elements of the associated subtree. The ANSV algorithm from Chapter 2 performs this for all nodes simultaneously in parallel in time $O(\frac{n}{p} + p)$. Having identified the size of each subtree, it is then straightforward to mark those subtrees with sizes below the threshold such that the subtree sizes of their respective parent nodes exceed the threshold.

The top-level trie is created in parallel as follows. Each processor creates new LCP and L_c arrays containing those elements which are not within the selected subtrees, i.e., those at higher levels. These arrays are then gathered together on all processors, where they are used to locally construct the top level trie. Finally, the top level tries are used for partitioning the subtrees onto processors and re-distributing the distributed SA and LCP accordingly.

4.3.3 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.

4.3.3 Distributed Locate

ALGORITHM 11: Distributed Locate

```
Function distributed_locate($P$)
    // Phase I: send $P$ to correct processor
    $\rho \leftarrow$ TLI.lookup_proc($P$)
    send($P$, to=$\rho$)
    // Phase II: locally query for possible match
    if $\rho$ = comm.rank() then
        $[l, r] \leftarrow$ idx.locate_possible($P$)
        send(($P, l, r, SA[l]$), to=rank_of($SA[l]$))
    end
    // Phase III: rule out false positive
    if $P, pos \leftarrow$ recv() then
        match $\leftarrow$ strcmp($P, S[pos \ldots]$)
        if no match then
            return $\emptyset$
        end
        return $P, [l, r]$
    end
end
```

Algorithm 11 shows at a high level the process of querying the Distributed ESA for a single pattern P. Given a pattern $P = p_0, \ldots, p_{m-1}$ 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 $\rho \in \{0, \ldots, p-1\}$ the subtree is located on. The pattern is then sent to processor ρ, where the local index is queried. The 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 $S[SA[l]...]$. Since the string

S is still *equally block distributed*, we send the pattern to the processor which contains the string segment starting at $SA[l]$. There, a single string comparison yields the required result. In some cases $SA[l] + m$ might not be located on the same processor as $SA[l]$. 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.

4.3.3 Distributed Bulk-Locate

The algorithm for bulk-parallel querying of many patterns simultaneously is shown in Algorithm 12. Each processor $\rho \in \{0, \ldots, p-1\}$ receives a separate list of patterns $P_1, \ldots, P_z$, 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 $[L[i], R[i]]$.

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 $SA[L[i]]$. Once received, the pattern can be compared with the string S at the suffix location. If the comparison is

ALGORITHM 12: Distributed Bulk-Locate

```
Function distributed_bulk_locate($[P_1, \dots, P_z]$)
    // Phase I: lookup where each pattern will be located
    $T \leftarrow$ new array(size $= z$)
    for $i = 1, \dots, z$ do
        $T[i] \leftarrow$ TLI.lookup_proc($P_i$)
    end
    // all-to-all exchange of patterns for querying
    $[Q_1, \dots, Q_y] \leftarrow$ alltoall($[P_1, \dots, P_z]$, to=$[T_1, \dots, T_z]$)

    // Phase II: local queries
    $L, R \leftarrow$ new arrays(size $= z$)
    $T \leftarrow$ new array(size $= y$)
    for $i = 1, \dots, y$ do
        // locally query for the pattern
        $L[i], R[i] \leftarrow$ idx.locate_possible($Q_i$)
        // determine which processor contains the string data for suffix SA[L[i]]
        $T[i] \leftarrow$ rank_of($S, SA[L[i]]$)
    end
    // all-to-all exchange
    $Q, L, R, pos \leftarrow$ alltoall($[\dots, (Q_i, L[i], R[i], SA[L[i]]), \dots]$,  to=$T$)
    send($P, SA[l]$, to=rank_of($SA[l]$))

    // Phase III: rule out false positives
    for $i = 1, \dots, |Q|$ do
        match $\leftarrow$ strcmp($Q_i, S[pos[i] \dots]$)
        if no match then
            $L[i], R[i] \leftarrow \emptyset$
        end
    end
    // return the result SA range for every query
    return $Q, L, R$
end
```

not successful, the solution range is replaced by the empty range $\emptyset$. Finally, the result can be saved or returned to the originating processor.

4.4 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 the construction algorithms from Chapter 2 [12]. This yields the SA and LCP arrays in (equally) block distributed form. We modify this 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 4.4.1 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.

4.4.1 Efficient Construction of L_c

To efficiently construct the L_c array, we modify the LCP construction algorithm of [12], which is a parallel, distributed memory adaptation of a similar LCP construction algorithm by Manber and Myers [6].

In [12], 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: $LCP[i] = h + LCP[RMQ_{LCP}(a, b)]$, 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 [12].

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 [12]: For the new $2h$-group boundaries, where the LCP is set to $LCP[i] = h + LCP[RMQ_{LCP}(a, b)]$, we also set

$$L_c[i] = L_c[RMQ_{LCP}(a, b)]$$

Claim 4.4.1. *When $LCP[i]$ is set as described above, it holds that:*

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

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 $q = RMQ_{LCP}(a, b)$

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:

$$L_c[i] = S[SA[i-1] + LCP[i]] = S[SA[a] + LCP[q]] \tag{4.1}$$

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.

□

4.4.2 Complexity

The run-time complexity construction of the DESA is dominated by the SA and LCP construction, which has a worst-case complexity of $O(\log(n)T_{sort}(n,p))$ [12], where $T_{sort}(n,p)$ is the parallel complexity to sort n elements on p processors.

4.5 Experiments and Results

We implemented the algorithms described above using MPI and C++11, and incorporated them into our distributed string algorithms framework [12, 13]. The code is available as open source on GitHub[1].

To ensure our parallelization efficiency does not stem from needless injection of redundancy, we demonstrate our performance on single core against state of the art sequential algorithms and codes, at the scale of problems which they can solve. For such 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 *Edison*, a Cray XC30 supercomputer. Each compute node has two sockets of Intel *Ivy Bridge* 12-core processors, for a total of 24 cores per node. Each node is equipped with 64 GB main memory and nodes are interconnected by a *Cray Aries* Dragonfly topology. For the experimental evaluation we used up to 64 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 the performance. The human genome (*H.sapiens*) contains 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 [34] version `GRCh37`.

4.5.1 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.

[1] github.com/patflick/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 4.1: 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.

- `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)* [50]. We use its current version 2.1.1 from GitHub [2].

We run query benchmarks using a set of different inputs from the *Pizza & Chili* Corpus provided by Ferragina *et al.* [51] and available online[3]. Same as in the `sdsl` benchmarks by Gog and Petri [50], we use the 200MB prefix of the input types `dna`, `proteins`, `english`, `sources`, and `dblp.xml`.

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

[3]http://pizzachili.dcc.uchile.cl/texts.html

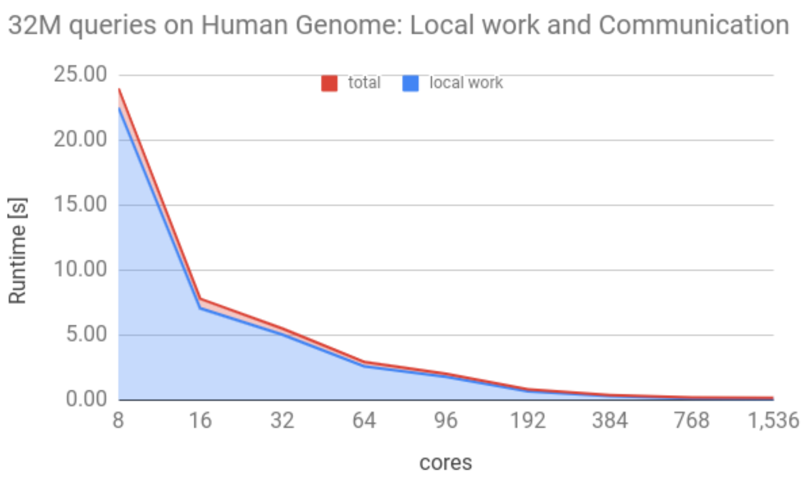


Figure 4.2: Local work and total runtime as a function of the number of cores for execution of `bulk_locate`, conducting 32 million pattern searches on the human genome.

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 4.1. 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 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.

4.5.2 Distributed Scaling

We implemented the distributed construction of the DESA data structure and the distributed bulk-query algorithm as presented in Algorithm 12.

In order to demonstrate the scalability of our data structure and algorithms, we use the human genome as an input and generate a pattern file with 32 million randomly chosen substrings of length 20. We ran scaling experiments on *Edison* on 8 to 64 nodes.

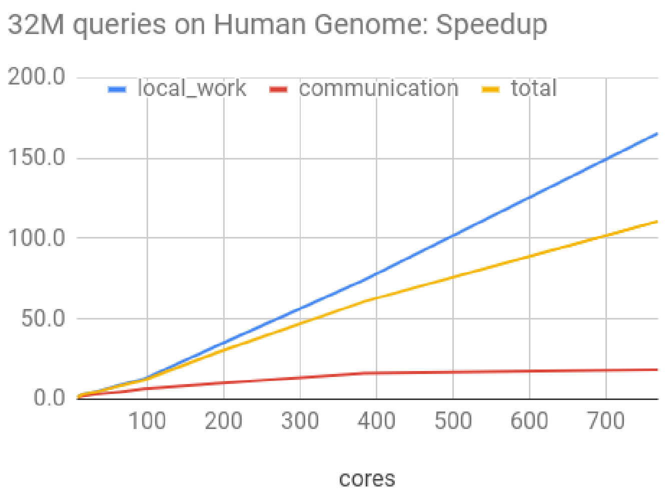


Figure 4.3: Relative speedup (strong scaling) of distributed querying on *Edison* 8-768 cores for 32 million queries on the human genome. The speedup is calculated with respect to the runtime using 8 cores on 8 nodes: $\frac{T(8)}{T(p)}$

Figure 4.2 shows the total time, and time spent in local work, plotted against the number of cores. As is evident, the time spent in communication (the incremental difference between the two graphs) remains a small fraction of the total runtime up until a large number of cores.

Figure 4.3 shows strong scaling results achieved by our implementation on *Edison*. The local work scales in a super-linear fashion, taking a total of 136ms on 768 cores (32 nodes) vs. 675 ms on 192 cores (8 nodes), corresponding to a relative speedup of $5\times$ while increasing the core count by only $4\times$. Similarly, the local work part of the algorithm takes 22.5 seconds on 8 cores (on 8 nodes), corresponding to a relative speedup of $136\times$ while increasing the core count by only $96\times$. The reason for distributing the 8 cores to 8 nodes instead of choosing them to be on the same node is because the memory required does not permit the usage of fewer than 8 nodes. The super-linear scaling is achieved because the distribution and splitting of the DESA onto more processors creates smaller local indexes of size $O(n/p)$, each of which can be queried faster while additionally benefiting from better cache reuse. The total runtime, including communication, goes from 826 ms on 192 cores, to 216 ms on 768 cores - a relative speedup of $3.8\times$. The runtime on 8 cores total is

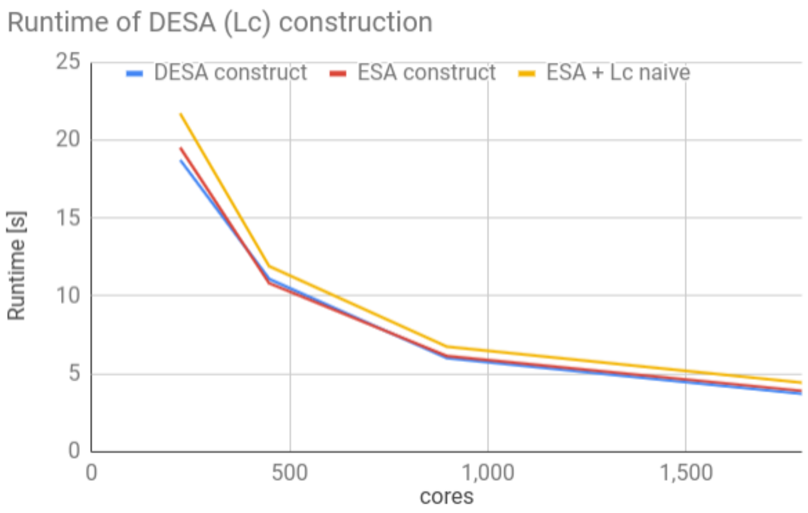


Figure 4.4: Runtime of distributed DESA construction compared to constructing the L_c array naively after construction of the suffix and LCP arrays.

24.0s - corresponding to a total speedup of over $110\times$ when increasing the processor count $96\times$.

4.5.3 Construction of L_c

In previous chapters, we experimentally compare the the SA and LCP construction against other approaches and demonstrates its superior performance. The construction of the SA and LCP arrays make up close to the entire runtime of the DESA construction. Therefore, here we show only the effect of the added L_c construction (as described in Section 4.4.1).

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 [12]), 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 4.4 and demonstrate that our L_c construction algorithm has a negligible contribution to the total runtime required to construct the other DESA components, whereas constructing the L_c array naively adds more than 10% in total runtime. Our work in [12] 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 further shown that the additional data structures required to create an efficiently query-able 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 *Edison* takes just under 4 seconds.

4.6 Conclusion

In this chapter, 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. We presented efficient distributed-memory parallel algorithms for querying, as well as for the efficient construction of this distributed index. We demonstrated the performance of our algorithms by comparing against other sequential approaches, and demonstrated strong scalability on over 1500 cores.

CHAPTER 5
DISTRIBUTED PARALLEL CONNECTED COMPONENTS LABELING OF DE-BRUIJN GRAPHS

5.1 Preface

The work on this project was initially motivated by a problem in Metagenomics assembly. The grand challenge Iowa corn soil metagenomic data set sequenced at the Joint Genome Institute contains 1.8 billion sequencing reads [52]. The corresponding de-Bruijn graph consists of approximately 135 billion vertices and edges, too large for any assembler to assembly directly. Howe *et al.* [53] discovered that the high species level heterogeneity in metagenomic data sets leads to a large number of disjoint connected components in the de Bruijn graph. This property can be exploited to partition the reads into disjoint sets and assemble each set independently.

Motivated by this application, we developed a distributed memory parallel connected components algorithm, making use of iterative sorting of an edge list graph format and merging of neighboring or overlapping components, as well as a neighbor doubling approach similar to the pointer jumping method used in list ranking. We demonstrated the scalability of this algorithm by partitioning the grand challenge Iowa corn soil metagenomic data set with 1.8 billion reads, a graph with $\approx$ 135 billion edges and $\approx$ 390 million components, in 22 minutes [15].

Our algorithm showed promising results also for other types for graphs. Chirag Jain as a lead author continued this work by creating a hybrid approach between our Connected Components (CC) algorithm and distributed memory Breadth-First-Search (BFS). We showed that using runtime algorithm selection between BFS and our CC algorithm, the hybrid method generalizes to diverse graph topologies and achieves superior performance [16]. The following Sections of this Chapter explain the algorithms involved and the experimental performance achieved.

5.2 Introduction

Computing connected components in undirected graphs is a fundamental problem in graph analytics. The sizes of graph data collections continue to grow in multiple scientific domains, motivating the need for high performance distributed memory parallel graph algorithms, especially for large networks that cannot fit into the memory of a single compute node. For a graph $G(V, E)$ with n vertices and m edges, two vertices belong to the same *connected component* if and only if there is a path between the two vertices in G. Sequentially, this problem can be solved in linear $O(m+n)$ time, e.g. by using one of the following two approaches. One approach is to use graph traversal algorithms, i.e., either Breadth First (BFS) or Depth First Search (DFS). A single traversal is necessary for each connected component in the graph. Another technique is to use a union-find based algorithm, where each vertex is initially assumed to be a different graph component and components connected by an edge are iteratively merged.

Parallel BFS traversal algorithms have been invented that are work-optimal and practical on distributed memory systems for small-world graphs [54, 55]. While parallel BFS algorithms have been optimized for traversing a short diameter big graph component, they can be utilized for finding connected components. However, connectivity can be determined for only one component at a time, as BFS cannot merge the multiple partial search trees in the same component that are likely to arise during concurrent runs. For an undirected graph with a large number of small components, parallel BFS thus has limited utility. On the other hand, BFS is an efficient technique for scale-free networks that are characterized by having one dominant short diameter component.

The classic Shiloach-Vishkin (SV) algorithm [56], a widely known PRAM algorithm for computing connectivity, simultaneously computes connectivity of all the vertices and promises convergence in logarithmic iterations, making it suitable for components with large diameter, as well as for graphs with a large number of small sized components. Note that compared to simple label propagation techniques, the SV algorithm bounds the number of iterations to $O(\log n)$ instead of $O(n)$, where each iteration requires $O(m + n)$ work. In this work, we provide a novel edge-based parallel algorithm for distributed memory systems based on the SV approach. We also propose optimizations to reduce data volume and balance load as the iterations progress.

To achieve the best performance for different graph topologies, we introduce a dynamic pre-processing phase to our algorithm that guides the algorithm selection at runtime. In this phase, we try to classify the graph as scale-free by estimating the goodness of fit of its degree distribution to a power-law curve. If and only if the graph is determined to be scale-free, we execute one BFS traversal iteration from a single root to find the largest connected component with high probability, before switching to the SV algorithm to process the remaining graph. While the pre-processing phase introduces some overhead, we are able to improve the overall performance by using a combination of parallel BFS and SV algorithms, with minimal parameter tuning.

Our primary application driver is metagenomic assembly, where de Bruijn graphs are used for reconstructing, from DNA sequencer outputs, constituent genomes in a metagenome[57]. A recent scientific study showed that high species-level heterogeneity in metagenomic data sets leads to a large number of weakly connected components, each of which can be processed as independent de Bruijn graphs [53]. This coarse grained data parallelism motivated our efforts in finding connected components in large metagenomic de Bruijn graphs. However, our work is applicable to graphs from domains beyond bioinformatics.

In this study, we cover a diverse set of graphs, both small world and large diameter, to highlight that our algorithm can serve as a general solution to computing connected components for undirected graphs. We experimentally evaluate our algorithm on de Bruijn graphs from publicly available metagenomic samples, road networks of the United States and European Union, scale-free networks from the internet, as well as Kronecker graphs from the Graph500 benchmark [58]. The graphs range in edge count from 82 million to 54 billion. Even though we focus on computing connected components in undirected graphs, ideas discussed in this work are applicable to finding strongly connected components in directed graphs as well. Our C++ and MPI-based implementation is available as open source at https://github.com/ParBLiSS/parconnect.

To summarize the contributions:

- We provide a new scalable strategy to adapt the Shiloach-Vishkin PRAM connectivity algorithm to distributed memory parallel systems.
- We discuss and evaluate a novel and efficient dynamic approach to compute weakly connected components on a variety of graphs, with small and large diameters.
- We demonstrate the scalability of our algorithm by computing the connectivity of the de Bruijn graph for a large metagenomic dataset with 1.8 billion DNA sequences and 54 billion edges in less than 4 minutes using 32K cores.
- Depending on the underlying graph topology, we see variable performance improvements up to 24x when compared against the state-of-the-art parallel connectivity algorithm.

The content of this Chapter was joined work with Chirag Jain and Tony Pan. The following list of papers are result of this work [15, 16, 17]:

- P. Flick *et al.*, "A Parallel Connectivity Algorithm for de Bruijn Graphs in Metagenomic Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 15
- C. Jain *et al.*, "An adaptive parallel algorithm for computing connected components," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, 2017

- P. Flick *et al.*, "Reprint of a parallel connectivity algorithm for de bruijn graphs in metagenomic applicationsi," *Parallel Computing*, vol. 70, pp. 54–65, 2017

5.3 Related Work

Due to its broad applicability, there have been numerous efforts to parallelize the connected component labeling problem. Hirschberg *et al.* [59] presented a CREW[1] PRAM algorithm that runs in $O(\log^2 n)$ time and does $O(n^2 \log n)$ work, while Shiloach and Vishkin [56] presented an improved version assuming a CRCW[2] PRAM that runs in $O(\log n)$ time using $O(m + n)$ processors. As our parallel SV algorithm is based on this approach, we summarize the SV algorithm in separate subsection. Krishnamurthy *et al.* [60] made the first attempt to adapt SV algorithm to distributed memory machines. However, their method is restricted to mesh graphs, which they could naturally partition among the processes [61]. Goddard *et al.* [62] discussed a practical implementation of SV algorithm for distributed machines with mesh network topology. Their method, however, was shown to exhibit poor scalability beyond 16 processors for sparse graphs [63].

Bader *el al.* [64] and Patwary *et al.* [65] discussed shared memory multi-threaded parallel implementations to compute spanning forest and connected components on sparse and irregular graphs. Recently, Shun *et al.* [66] reported a work optimal implementation for the same programming model. Note that these solutions are not applicable for distributed memory environments due to high frequency of remote memory accesses. Cong *et al.* [67] proposed a parallel technique for solving the connectivity problem on a single GPU.

There have been several recent parallel algorithms for computing the breadth-first search (BFS) traversal on distributed memory systems [54, 55, 68]. However, parallel BFS does not serve as an efficient, stand-alone method for computing connectivity. There are also several large-scale distributed graph analytics frameworks that can solve the connectivity problem in large graphs, including GraphX [69], PowerLyra [70], PowerGraph [71], and

[1]CREW = Concurrent Read Exclusive Write
[2]CRCW = Concurrent Read Concurrent Write

GraphLab [72]. Iverson *et al.* [73] proposed a distributed-memory connectivity algorithm using successive graph contraction operations, however, the strong scalability demonstrated for this method was limited to 32 cores.

Slota *et al.* [74] proposed a shared memory parallel *Multistep* method that combines parallel BFS and label propagation (LP) technique and was reported to perform better than using BFS or LP alone. In their *Multistep* method, BFS is first used to label the largest component before using the LP algorithm to label the remaining components. More recently, they proposed a distributed memory parallel implementation of this method and showed impressive speedups against the existing parallel graph processing frameworks [75]. However, their algorithm design and experimental datasets are restricted to graphs which contain a single massive connected component. While our algorithm likewise employs a combination of algorithms, in contrast to *MultiStep*, we use BFS and our novel SV implementation, and determine dynamically at runtime whether the BFS should be executed.

5.3.0 The Shiloach-Vishkin Algorithm

The Shiloach-Vishkin connectivity algorithm was designed assuming a PRAM model. It begins with singleton trees corresponding to each vertex in the graph and maintains this auxiliary structure of rooted directed trees to keep track of the connected components discovered so far during the execution. Within each iteration, there are two phases referred to as $shortcutting$ and $hooking$. $Shortcutting$ involves collapsing the trees using pointer doubling. On the other hand, $hooking$ connects two different connected components when they share an edge in the input graph. This algorithm requires $O(\log n)$ iterations each taking constant time. Since this approach uses $O(m+n)$ processors, the total work complexity is $O((m+n) \cdot \log n)$.

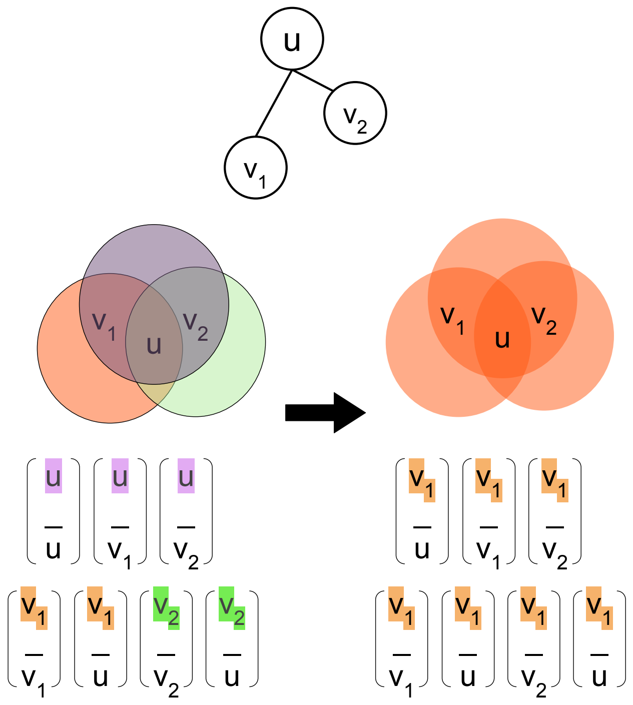


Figure 5.1: Initialization of array $\mathcal{A}$ for a small connected component with three vertices u, v_1, v_2 in our algorithm. Partitions are highlighted using different shades. Desired solution, assuming $v_1 = \min(u, v_1, v_2)$, shown on the right will be to have all three vertices in a single component v_1. Accordingly, all the tuples associated with this component should contain the equal partition id v_1.

Symbol	Description	Definition
V	Vertices in graph G	
E	Edges in graph G	
$\langle p, q, r \rangle$	Tuple	$p, q, r \in \mathbb{Z}$
$\mathcal{A}_i$	Array of tuples in iteration i	
$\mathcal{P}_i$	Unique partitions	$\{p \mid \langle p, q, r \rangle \in \mathcal{A}_i\}$
$\mathcal{PB}_i(p)$	Partition bucket for partition p	$\{\langle \hat{p}, q, r \rangle \in \mathcal{A}_i \mid \hat{p} = p\}$
$\mathcal{VB}_i(u)$	Vertex bucket for vertex u	$\{\langle p, q, r \rangle \in \mathcal{A}_i \mid r = u\}$
$\mathcal{V}_i(p)$	Vertex members in partition p	$\{r \mid \langle p, q, r \rangle \in \mathcal{PB}_i(p)\}$
$\mathcal{C}_i(p)$	Candidate partitions for partition p	$\{q \mid \langle p, q, r \rangle \in \mathcal{PB}_i(p)\}$
$\mathcal{M}_i(u)$	Partitions in vertex bucket for vertex u	$\{p \mid \langle p, q, r \rangle \in \mathcal{VB}_i(u)\}$
$\mathcal{N}_i(p)$	Neighborhood partitions of partition p	$\cup_{u \in \mathcal{V}_i(p)} \mathcal{M}_i(u)$

Table 5.1: Summary of the notations used in Section 5.4

5.4 Algorithm

5.4.1 Parallel SV Algorithm

5.4.1 Notations

Given an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, our algorithm identifies its connected components, and labels each vertex $v \in V$ with its corresponding component. Our algorithm works on an array of 3-tuples $\langle p, q, r \rangle$, where p, q, and r are integers. The first two elements of these tuples will be updated in each iteration of the algorithm. The third element r corresponds to a vertex $r \in V$ of the graph and is not changed throughout the algorithm. This element will also be used to identify the vertices of G with their final connected components after termination.

Let $\mathcal{A}_i$ denote the array of tuples in iteration i. We initialize $\mathcal{A}_0$ as follows: for each vertex $x \in V$, we add the tuple $\langle x, \_, x \rangle$, and for each undirected edge $\{x, y\} \in E$, we add tuples $\langle x, \_, y\rangle$ and $\langle y, \_, x\rangle$. The middle elements will be initialized later during the algorithm.

We denote the set of unique values in the first entry of all the tuples in $\mathcal{A}_i$ by $\mathcal{P}_i$, therefore $\mathcal{P}_i = \{p \mid \langle p, q, r\rangle \in \mathcal{A}_i\}$. We refer to the unique values in $\mathcal{P}_i$ as *partitions*, which represent intermediate groupings of tuples that eventually coalesce into connected components. We say that a tuple $\langle p, q, r\rangle$ is a member of the partition p. Once the algorithm converges, all tuples for a vertex r will have a single unique partition p, which is also the unique connected component label for this vertex.

In order to refer to the tuples of a partition p, we define the *partition bucket* $\mathcal{PB}_i(p)$ of p as those tuples which contain p in their first entry: $\mathcal{PB}_i(p) = \{\langle \hat{p}, q, r\rangle \in \mathcal{A}_i \mid \hat{p} = p\}$. Further, we define the *candidates* or the next potential partitions $\mathcal{C}_i(p)$ of p as the values contained in the second tuple position of the partition bucket for p: $\mathcal{C}_i(p) = \{q \mid \langle p, q, r\rangle \in \mathcal{PB}_i(p)\}$. We denote the minimum of the candidates of p as $p_{min} = \min \mathcal{C}_i(p)$. A partition p for which $p_{min} = p$ is called a *stable partition*. Further, to identify all the vertices in a partition, we define the *vertex members* of a partition p as $\mathcal{V}_i(p) = \{r \mid \langle p, q, r\rangle \in \mathcal{PB}_i(p)\}$.

Each vertex $u \in V$ is associated with multiple tuples in $\mathcal{A}_i$, possibly in different partitions p. We define *vertex bucket* $\mathcal{VB}_i(u)$ as those tuples which contain u in their third entry: $\mathcal{VB}_i(u) = \{\langle p, q, r\rangle \in \mathcal{A}_i \mid r = u\}$. We define the partitions $\mathcal{M}_i(u)$ as the set of partitions in the vertex bucket for u: $\mathcal{M}_i(u) = \{p \mid \langle p, q, r\rangle \in \mathcal{VB}_i(u)\}$. The minimum partition in $\mathcal{M}_i(u)$, i.e., $\min \mathcal{M}_i(u)$ is called *nominated partition* by u.

For a small example graph with vertices u, v_1, v_2, (Fig. 5.1), we show the array of tuples $\mathcal{A}$. At the initialization stage, the vertex bucket $\mathcal{VB}_0(u)$ of u is the set of tuples $\{\langle u, \_, u\rangle, \langle v_1, \_, u\rangle, \langle v_2, \_, u\rangle\}$. The set of unique partitions $\mathcal{P}_0$ equals $\{u, v_1, v_2\}$. The partition bucket $\mathcal{PB}_0(u)$ for partition u is given by the set $\{\langle u, \_, u\rangle, \langle u, \_, v_1\rangle, \langle u, \_, v_2\rangle\}$. At termination of our algorithm, all tuples will have the same common partition id, which for this example will be $\min(u, v_1, v_2)$.

Each partition is associated with a set of vertices, and the tuples for a vertex can be part of multiple partitions. We define the neighborhood for a partition p as those partitions which share at least one vertex with p, i.e., those which share tuples with a common identical value in the third tuple element. More formally, we define the *neighborhood* partitions of p as $\mathcal{N}_i(p) = \cup_{u \in \mathcal{V}_i(p)} \mathcal{M}_i(u)$. In the above example, the neighborhood partitions $\mathcal{N}_0(v_1)$ for the partition v_1 are u, v_1 and v_2. All the notations introduced in this section are summarized in Table 5.1 for quick reference.

5.4.1 Algorithm

We first describe the sequential version of our algorithm, outlined in Algorithm 13. Our algorithm is structured similar to the classic *Shiloach-Vishkin* algorithm. However, our algorithm is implemented differently, using an edge-centric representation of the graph.

At a high level, every vertex begins in its own partition, and partitions are connected via the edges of the graph. In each iteration, we join each partition to its numerically minimal neighbor, until the partitions converge into the connected components of the graph. In order to resolve large diameter components quickly, we utilize the pointer doubling technique during shortcutting. To implement pointer doubling, we will require the *parent* partition id of the newly joined partition in each iteration. We use *temporary* tuples $\langle p, q, r \rangle_{tmp}$ to fetch this information. These tuples will be created and erased within the same iteration.

As laid out in Section 5.4.1.1, we first create an array of tuples $\mathcal{A}$, containing one tuple per vertex and two tuples per edge (Algorithm 13). In each iteration i, we perform four sorting operations over $\mathcal{A}_i$. During the first two sorting operations, we compute and join each partition p to its minimum neighborhood, i.e. $\min \mathcal{N}_i(p)$. Sorting $\mathcal{A}_i$ by the third entry, namely the vertex ids enables easy and cache efficient processing of each vertex bucket $\mathcal{VB}_i(u), u \in V$, since the tuples of a bucket are positioned contiguously in $\mathcal{A}_i$ due to the sorted order (line 9-15). For each vertex bucket $\mathcal{VB}_i(u)$, we scan all the partition ids containing u, i.e., $\mathcal{M}_i(u)$ and compute the nominated partition u_{min} which becomes the candidate (potential next partition). We save the candidate partition id in the second

```
ALGORITHM 13: Connected components labeling
Input: Undirected graph $G = (V, E)$
Output: Labeling of Connected Components
1  $\mathcal{A}_0 = []$
2  for $x \in V$ do $\mathcal{A}_0$.append($\langle x, \_, x \rangle$);
3  for $\{x, y\} \in E$ do $\mathcal{A}_0$.append($\langle x, \_, y \rangle$, $\langle y, \_, x \rangle$);
4  $i \leftarrow 1$
5  $converged \leftarrow$ false
6  while $converged \neq true$ do
7      $converged \leftarrow$ true
8      $\mathcal{A}_i \leftarrow \mathcal{A}_{i-1}$
9      $\mathcal{M}_i(u) \leftarrow \texttt{sort}(\mathcal{A}_i$ by third element)
10     for $u \in V$ do
11         $u_{min} \leftarrow \min \mathcal{M}_i(u)$
12         for each $\langle p, q, r \rangle \in \mathcal{VB}_i(u)$ do
13             $\langle p, \boldsymbol{q}, r \rangle \leftarrow \langle p, \boldsymbol{u_{min}}, r \rangle$
14         end
15     end
16     $\mathcal{C}_i(p) \leftarrow \texttt{sort}(\mathcal{A}_i$ by first element)
17     for $p \in \mathcal{P}_i$ do
18         $p_{min} \leftarrow \min \mathcal{C}_i(p)$
19         if $p \neq p_{min}$ then
20             $converged \leftarrow$ false
21         end
22         for each $\langle p, q, r \rangle \in \mathcal{PB}_i(p)$ do
23             $\langle \boldsymbol{p}, q, r \rangle \leftarrow \langle \boldsymbol{p_{min}}, q, r \rangle$
24         end
25         $\mathcal{A}_i$.append($\langle p_{min}, \_, p_{min} \rangle_{tmp}$)
26     end
27     redo steps 9 - 15
28     redo steps 16 - 24
29     for each $\langle p, q, r \rangle_{tmp} \in \mathcal{A}_i$ do
30         $\mathcal{A}_i$.erase($\langle p, q, r \rangle_{tmp}$)
31     end
32     $i \leftarrow i + 1$
33 end
```

Figure 5.2: Our parallel SV algorithm, presented using sequential semantics.

element of the tuples.

After computing all the candidate partitions, we perform a second global sort of $\mathcal{A}_i$ by the first tuple element in order to process the partition buckets $\mathcal{PB}_i$ (line 16-24). Each partition $p \in \mathcal{P}_i$ then computes and joins the minimum candidate partition, i.e., $p_{min} =$

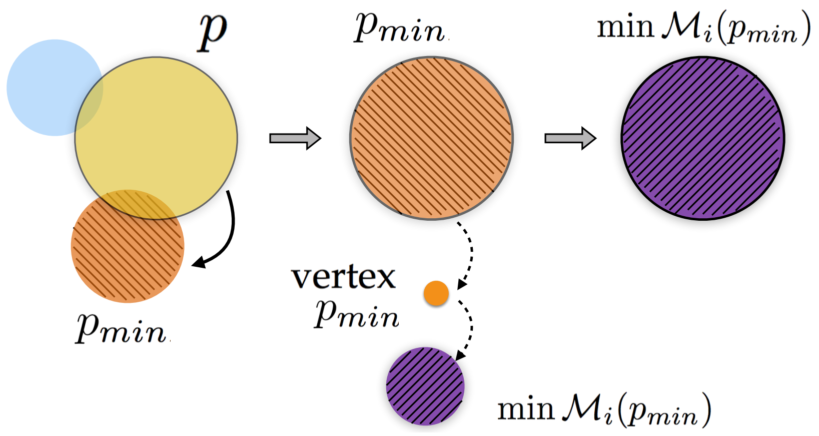


Figure 5.3: Role of the four sorting phases used in each iteration of the algorithm. Using the first two sorts, partition p joins p_{min}. The next two sorts enable pointer-jumping as p_{min} joins $\min \mathcal{M}_i(p_{min})$. The temporary tuple $\langle p_{min}, \_, p_{min}\rangle_{tmp}$ used in the algorithm simulates a link between the partition p_{min} and the vertex p_{min} to allow jumping.

$\min \mathcal{C}_i(p)$. In other words, partition p joins its minimum neighbor p_{min}. We loop over these two sort-and-update steps until partitions converge into the connected components of the graph. Convergence for a partition p is reached when its *neighborhood* $\mathcal{N}_i(p)$ contains p as its only member. Consequently, we can determine when to terminate the algorithm by checking whether all the partitions have fully converged, i.e., if they do not have any further neighboring partitions. For any partition p, $p \neq p_{min}$ implies the existence of at least one neighbor partition around p (line 19).

Iteratively invoking lines 7-24 until convergence produces connected components of the graph within $O(n)$ iterations in the worst-case. By following the pointer doubling technique described in the SV algorithm [56], we achieve logarithmic convergence. We summarize the role of all the four sorting operations in Figure 5.3. After joining partition p to p_{min}, we revise p_{min} to $\min \mathcal{M}_i(p_{min})$. The revision is effected by introducing temporary tuples $\langle p_{min}, \_, p_{min}\rangle_{tmp}$ in $\mathcal{A}_i$ (line 25), then repeating the two sorts by the third and first element respectively (line 27, 28). In a way similar to the first two sorts of this iteration, the third sort forces the vertex p_{min} to nominate $\min \mathcal{M}_i(p_{min})$ as the candidate partition id in the second element of the temporary tuples. Partition p_{min}, then, joins the partition id $\min \mathcal{M}_i(p_{min})$ after the final sort. The temporary tuples are removed from $\mathcal{A}_i$ after the pointer doubling phase is completed (line 30).

Note that the global count of the temporary tuples equals $|\mathcal{P}_i|$ in each iteration, and we know $|\mathcal{P}_i| \leq |V|$ (by the definition of $\mathcal{P}_i$). Therefore, the $O(m+n)$ bound holds for $|\mathcal{A}_i|$ throughout the execution. After the algorithm converges, the unique connected component label c of a vertex $u \in V$ can be projected from the first element of any tuple $\langle c, \_, u\rangle$ in $\mathcal{A}$.

5.4.1 Parallel Algorithm

We now describe our parallel implementation of the above algorithm for connected components labeling in a distributed memory environment. In this setting, each processor in the environment has its own locally addressable memory space. Remote data is accessible only through well defined communication primitives over the interconnection network. The algorithm consists of three components: data distribution, parallel sorts, and bucket updates. We designed our algorithm and its components using MPI primitives.

Data Distribution: All data, including the input, intermediate results, and final output, are equally distributed across all available processors. As specified in section 5.4.1.2, the pipeline begins by generating tuples of the form $\langle p, q, r\rangle$ from the block distributed input $G(V, E)$ as edge list. By the end of this operation, each of the ρ processes contains its equal share of $|\mathcal{A}|/\rho$ tuples.

Parallel Sorts: The bulk step of the algorithm is the sorting of tuples by either their third or first element in order to form the buckets $\mathcal{VB}_i$ or $\mathcal{PB}_i$, respectively. Parallel distributed memory sorting has been studied extensively. Blelloch *et al.* [32] give a good review of different methods. With sufficiently large count of elements per process, which is often true while processing large datasets, the study concluded that samplesort is the fastest. Accordingly, we implement a variant of samplesort with regular sampling, where each processor first sorts its local array independently, and then picks equally spaced samples. The samples are then again sorted and $\rho - 1$ of these samples are used as splitters for distributing data among processors. In a final step, the sorted sequences are merged locally.

Bucket Updates: After each sort, we need to determine the minimum element for each bucket, either u_{min} for $\mathcal{VB}_i(u)$ or p_{min} for $\mathcal{PB}_i(p)$. As a result of the parallel sorting, all the tuples $\langle p, q, r \rangle$ belonging to the same bucket are stored consecutively. However, a bucket might span multiple processors. Therefore, the first and last bucket of each processor require global communication during processing, while the internal buckets are processed in the same way as in the sequential case. Note, the first and last bucket on a processor may be the same if a bucket spans an entire processor. Communicating the minimum of buckets with the previous and next processor would require $O(\rho)$ communication steps in the worst case, since large $O(|\mathcal{A}|)$ size partitions can span across $O(\rho)$ processes. We thus use two parallel prefix (scan) operations with custom operators to achieve independence from the size of partitions, requiring at most $O(\log \rho)$ communication steps in addition to the local linear time processing time.

We describe the custom reduction operation to compute the p_{min} within the partition buckets $\mathcal{PB}_i(p)$. Note that when computing p_{min} in the algorithm, $\mathcal{A}_i$ is already sorted by the first element of the tuples and p_{min} is the minimum second element for tuples in each bucket. We first perform an exclusive scan, where each processor participates with the minimum tuple from its last bucket. This operation communicates the minimum of buckets from lower processor rank to higher rank. The binary reduction operator chooses from 2 tuples the tuple $\langle p, q, r \rangle$ with the maximum p, and between those with equal p, the minimum q. Next we perform a reverse exclusive prefix scan to communicate the minimum from high rank to low rank. Here, each processor participates with its minimum tuple of its first bucket. Given the two results of the scan operations, we can compute for each processor the overall minimum p_{min} for both the first and the last buckets. Computing u_{min} follows a similar procedure.

Runtime Complexity: The runtime complexity of each iteration is dominated by sorting $\mathcal{A}$, and the number of iterations is bounded by $O(\log n)$. If $T(k, \rho)$ is the runtime to sort k elements using ρ processes, the runtime of our algorithm for computing connectivity of graph $G(V, E)$ equals $O(log(n) \cdot T(m + n, \rho))$.

5.4.1 *Excluding Completed Partitions*

As the algorithm progresses through iterations, certain partitions become *completed*. A partition p is *completed* if p has no neighbor partition except itself, i.e., $\mathcal{N}_i(p) = \{p\}$. Even though we have described how to detect the global convergence of the algorithm, detecting as well as excluding the *completed* partitions reduces the active working set throughout successive iterations.

By the definition of $\mathcal{N}_i(p)$ in Section 5.4.1.1, $\mathcal{N}_i(p) = \{p\}$ implies that $\cup_{u \in \mathcal{V}_i(p)} \mathcal{M}_i(u) = \{p\}$. Since the third elements of the tuples are never altered, each vertex is associated with at least one partition throughout the algorithm, therefore $|\mathcal{M}_i(u)| > 0 \, \forall u \in V$. Using these arguments, we claim the following: p is *completed* $\Leftrightarrow \mathcal{M}_i(u) = \{p\} \, \forall u \in \mathcal{V}_i(p)$. Once the partition is *completed*, it takes us one more iteration to detect its completion. While processing the vertex buckets after the first sort of the algorithm, we label all the tuples in $\mathcal{VB}_i(u), u \in V$ as *potentially completed* if $|\mathcal{M}_i(u)| = 1$. While processing the partition buckets subsequently, partition p is marked as *completed* if all the tuples in $\mathcal{PB}_i(p)$ are *potentially completed*.

Completed partitions are marked as such and swapped to the end of the local array. All following iterations treat only the first, non-completed part of its local array as the local working set. As a result, the size of the active working set shrinks throughout successive iterations. This optimization yields significant reduction in the volume of active data, particularly for graphs with a large number of small components, since many small connected components are quickly identified and excluded from future processing.

5.4.1 *Load Balancing*

Although we initially start with a block decomposition of the array $\mathcal{A}$, exclusion of *completed partitions* introduces an increasing imbalance of the active elements with each iteration. Since we join partitions from larger *ids* to smaller *ids*, a large partition will have smaller final partition ids than small partitions probabilistically. As the sort operation maps large id partitions to higher rank processes, the higher rank processes retain fewer and fewer active tuples over time, while lower rank processes contain growing partitions with small ids. Our experiments in Section 5.6 study this imbalance of data distribution and its effect on the overall run time. We resolve this problem and further optimize our algorithm by evenly redistributing the active tuples after each iteration. Our results show that this optimization yields significant improvement in the total run time.

5.4.2 Hybrid Implementation using BFS

Connected components can be found using a series of BFS traversals, one for each component. The known parallel BFS algorithms are asymptotically work-optimal, i.e., they maintain $O(m + n)$ parallel work for small-world networks [54]. Parallel BFS software can be adapted to achieve the same objective as our parallel SV algorithm, namely to compute all the connected components in a graph. To do so, parallel BFS can be executed iteratively, each time selecting a new seed vertex from among the vertices that were not visited during any of the prior BFS iterations. However, we note the following strengths and weaknesses associated with using BFS methods for the connectivity problem:

- **Pro:** For a massive connected component with a small diameter, the large number of vertices at each level of the traversal yields enough data parallelism for parallel BFS methods to become bandwidth bound, and thus efficient.
- **Con:** When the diameter of a component is large and vertex degrees are small, for instance in mesh graphs, the number of vertices at each level of BFS traversal is small. The application becomes latency-bound due to the lack of data parallelism. This leads to under-utilization of the compute resources and the loss of efficiency in practice [54].

- **Con:** For graphs with a large number of small components, parallel BFS needs to be executed repeatedly. The application becomes latency-bound as the synchronization and remote communication latency costs predominate the effective work done during the execution. In this case, BFS method's scalability is greatly diminished. Slota *et al.* [74] draw a similar conclusion while parallelizing the strongly connected components problem using shared memory systems.

A small world scale-free network contains a single large connected component [76]. To compute the connectivity of these graphs, we note that identifying the first connected component using a BFS traversal is more efficient than using the SV algorithm over the complete graph. For parallel BFS, we use Buluç *et al.*'s [54] state-of-the-art implementation available as part of the CombBLAS library [77] and integrate this software as an alternative pre-processing step to our parallel SV algorithm.

Scale-free networks are characterized by a power-law vertex degree distribution [78]. Therefore, we classify the graph structure as scale-free by checking if the degree distribution follows a power-law distribution. We use the statistical framework described by Clauset *et al.* [79] to fit a power-law curve to the discrete graph degree distribution, and estimate the goodness of fit with one-sample Kolmogorov-Smirnov (K-S) test. The closer the K-S statistic value is to 0, the better is the fit. If this value is below a user specified threshold τ, then we execute a BFS iteration before invoking our parallel SV algorithm. Algorithm 14 gives the outline of our hybrid approach.

In our implementation, we choose to store each undirected edge (u, v) as two directed edges (v, u) and (u, v) in our edge list. This simplifies the computation of the degree distribution of the graph (line 2). We compute the degree distribution $\mathcal{D}$ of the graph by doing a global sort of edge list by the source vertex. Through a linear scan over the sorted edge list, we compute the degree of each vertex $u \in V$. In practice, it is safe to assume that the maximum vertex degree c is much smaller than number of edges $|E|$ ($c \ll |E|$). Thus each process can compute the local degree distribution in an array of size c, and a

ALGORITHM 14: Connected components labeling

```
Input: Undirected graph $G = (V, E)$
Output: Labeling of Connected Components
1  // Graph structure prediction
2  $\mathcal{D} \leftarrow$ Degree_Distribution$(G)$
3  if K-S statistic ($\mathcal{D}$) $< \tau$ then
4      // Relabel vertices
5      $G(V, E) \leftarrow G(V, E)$ s.t. $u \in [0, |V| - 1] \, \forall u \in V$
6      //Execute BFS
7      choose a seed $s \in V$
8      $\mathcal{VI} \leftarrow$Parallel-BFS$(s)$
9      //Filter out the traversed component
10     $V \leftarrow V \setminus \mathcal{VI}$
11     $E \leftarrow E \setminus \{(u, v) | u \in \mathcal{VI}\}$
12 end
13 Parallel-SV$(G(V, E))$
```

Figure 5.4: Hybrid approach using parallel BFS and SV algorithms to compute connected components

parallel reduction operation is used to solve for $\mathcal{D}$. Once $\mathcal{D}$ is known, evaluating the degree distribution statistics takes insignificant time as size of $\mathcal{D}$ equals c. Therefore, we compute the K-S statistics as described before, sequentially on each process.

If the K-S statistic is below the set threshold, we choose to run the parallel BFS on $G(V, E)$ (line 3). Buluç's BFS implementation works with the graph in an adjacency matrix format. Accordingly, we relabel the vertices in $G(V, E)$ such that vertex ids are between 0 to $|V| - 1$ (line 5). This process requires sorting the edge list twice, once by the source vertices and second by the destination vertices. After the first sort, we perform a parallel prefix (scan) operation to label the source vertices with a unique id $\in [0, V - 1]$. Similarly, we update the destination vertices using the second sort.

Next, we execute the parallel BFS from a randomly selected vertex in $G(V, E)$ and get a distributed list of visited vertices $\mathcal{VI}$ as the result. Note that the visited graph component is expected to be the largest one as it spans the majority of $G(V, E)$ in the case of scale-free graphs. To continue solving for other components, we filter out the visited component $\mathcal{VI}$ from $G(V, E)$ (line 10,11). $\mathcal{VI}$ is distributed identically as V, therefore vertex filtering is done locally on each process. We already have the edge list E in the sorted order by destination vertices due to the previous operations, therefore we execute an all-to-all collective operation to distribute $\mathcal{VI}$ based on the sorted order and delete the visited edges locally on each processor. Finally, irrespective of whether we use BFS or not, we run the parallel SV algorithm on $G(V, E)$ (line 13). In our experiments, we show the overall gain in performance using the hybrid approach as well as the additional overhead incurred by the prediction phase. We also report the proportion of time spent in each of the prediction, relabeling, parallel-BFS, filtering, and parallel-SV stages.

5.5 Experimental Setup

5.5.1 Hardware

For the experiments, we use Edison, a Cray XC30 supercomputer located at Lawrence Berkeley National Laboratory. In this system, each of the 5,576 compute nodes has two 12-core Intel Ivy Bridge processors with 2.4 GHz clock speed and 64 GB DDR3 memory. To perform parallel I/O, we use the scratch storage supported through the Lustre file system. We assign one MPI process per physical core for the execution of our algorithm. Further, we only use square process grids as CombBLAS [77] requires the process count to be a perfect square.

5.5.2 Datasets

Table 5.2 lists the 9 graphs used in our experiments. These include 4 de Bruijn graphs constructed from different metagenomic sequence datasets, one social graph from Twitter, one web crawl, one road network and two synthetic Kronecker graphs from the Graph500 benchmark. The sizes of these graphs range from 83 million edges to 54 billion edges.

For each graph, we report the relevant statistics in Table 5.2 to correlate them with our performance results. Computing the exact diameter is computationally expensive and often infeasible for large graphs [82]. As such, we compute their approximate diameters by

Table 5.2: List of the nine graphs and their sizes used for conducting experiments. Edge between two vertices is counted once while reporting the graph sizes. Largest component's size is computed in terms of percentage of count of edges in the largest component relative to complete graph.

Id	Dataset	Type	Vertices	Undirected Edges	Components	Approx. diameter	Largest component	Source
M1	Lake Lanier	Metagenomic	1.1 B	1.1 B	2.6 M	3,763	53%	NCBI (SRR947737)
M2	Human Metagenome	Metagenomic	2.0 B	2.0 B	1.0 M	3,989	91.1%	NCBI (SRR1804155)
M3	Soil (Peru)	Metagenomic	531.2 M	523.6 M	7.6 M	2,463	0.3%	MG-RAST (4477807.3)
M4	Soil (Iowa)	Metagenomic	53.7 B	53.6 B	319.2 M	-	44.2%	JGI (402461)
G1	Twitter	Social	52.6 M	2.0 B	29,533	16	99.99%	[80]
G2	sk-2005	Web Crawl	50.6 M	1.9 B	45	27	99.99%	[81]
G3	eu-usa-osm	Road Networks	74.9 M	82.9 M	2	25,105	65.2%	[81]
K1	Kronecker (scale = 27)	Kronecker	63.7 M	2.1 B	19,753	9	99.99%	Synthetic [58]
K2	Kronecker (scale = 29)	Kronecker	235.4 M	8.6 B	73,182	9	99.99%	Synthetic [58]

executing a total of 100 BFS runs from a set of random seed vertices. For all the graphs but M4, this approach was able to give us an approximation. However, the size of M4 required a substantial amount of time for completing this task and as such it did not complete. We estimate that only 4 of the 9 tested graphs are small world networks.

5.5.2 *Metagenomic de Bruijn Graphs*

M1-M4 are built using publicly available metagenomics samples from different environments. We obtained the sequences in FASTQ format. We discarded the sequences with unknown nucleotides using the fastx_clipper utility supported in the FASTX toolkit [83]. The size of the sequence dataset depends upon the amount of sampling done for each environment. We build de Bruijn graphs from these samples using the routines from the parallel distributed memory k-mer indexing library Kmerind [84]. It is worth noting that in de Bruijn graphs, vertex degrees are bounded by 8 [57]. One motivation for picking samples from different environments is the difference in graph properties associated with them such as the number of components and relative sizes. These are dependent on the degree of microbial diversity in the environments. Among the environments we picked, it has been estimated that the soil environments are the most diverse while the human microbiome samples are the least diverse of these environments [85]. This translates to large number of connected components in the soil graphs M3 and M4.

5.5.2 Other Graphs

Graphs K1-K2 and G1-G3 are derived from widely used graph databases and benchmarks. We use the synthetic Kronecker graph generator from the Graph500 benchmark specifications [58] to build Kronecker graphs with scale 27 (K1) and 29 (K2). Graphs G1-G3 are downloaded directly from online databases in the edge list format. G1 and G2 are small world scale-free networks from twitter and online web crawl respectively. G3 consists of two road networks from Europe and USA, downloaded from the Florida Sparse Matrix Collection [81]. Among all our graphs, G3 has the highest estimated diameter of 25K. To read these data files in our program, a file is partitioned into equal-sized blocks, one per MPI process. The MPI processes concurrently read the blocks from the file system and generate distributed arrays of graph edges in a streaming fashion.

5.6 Performance Analysis

In all our experiments, we exclude file I/O and de Bruijn graph construction time from our benchmarks, and begin profiling after the block-distributed list of edges are loaded into memory. Profiling terminates after computing the connected component labels for all the vertices in the graph. Each vertex id in the input edge list is assumed to be a 64 bit integer. The algorithm avoids any runtime bias on vertex naming of the graph by permuting the vertex ids using Robert Jenkin's 64 bit mix invertible hash function [86].

5.6.1 Load Balancing

We first show the impact of the two optimizations performed by our parallel SV algorithm (Sections 5.4.1.4 and 5.4.1.5) for reducing and balancing the work among the processes. Our algorithm used 10 iterations to compute the connectivity of M1. Figure 5.5 shows the minimum (min), maximum (max), and mean size of the distributed tuple array per process as iterations progress in three variants of our algorithm, using 256 cores. The max load is important as it determines the parallel runtime. A smaller separation between the min and max values indicates better load balance. The first implementation, referred to as Naive (Section 5.4.1.3), does not remove the completed components along the iterations and therefore the work load remains constant. Removing the stable components reduces the size of the working set per each iteration as illustrated by the desirable decrease in mean tuple count. The difference between min and max grows significantly after 4 iterations. With our load balanced implementation, we see an even distribution of tuples across processors, as the minimum and maximum count are the same for each iteration. We see that the mean drops to about 50% of the initial value because the largest component in M1 contains 53% of the total edges (Table 5.2).

Consequently, we see improvement in the execution time for M1 and M3 in Figure 5.6 as a result of these optimizations. Of the three implementations, the load balanced implementation consistently achieves better performance against the other two approaches. For the M2 graph, we get negligible gains using our load-balanced approach against the Naive approach because the largest component in M2 covers 91% of the graph. Therefore, the total work load stays roughly the same across the iterations.

5.6.2 Hybrid Implementation Analysis

As discussed in Section 5.4.2, BFS is more efficient for computing the first component in the small world scale-free graphs. We use an open-source C++ library [87] which fits the power-law distributions to discrete empirical data based on the procedure described

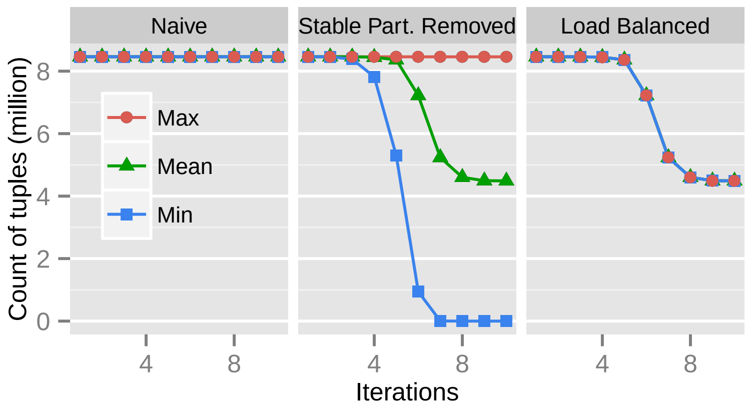


Figure 5.5: Work load balance in terms of tuples per processes during each iteration of the three algorithm variants for parallel SV algorithm. Illustrated are the maximum, average, and minimum count of tuples on all the processes. The experiments were conducted using the M1 graph and 256 processor cores. Each edge is represented as 2 tuples internally in the algorithm.

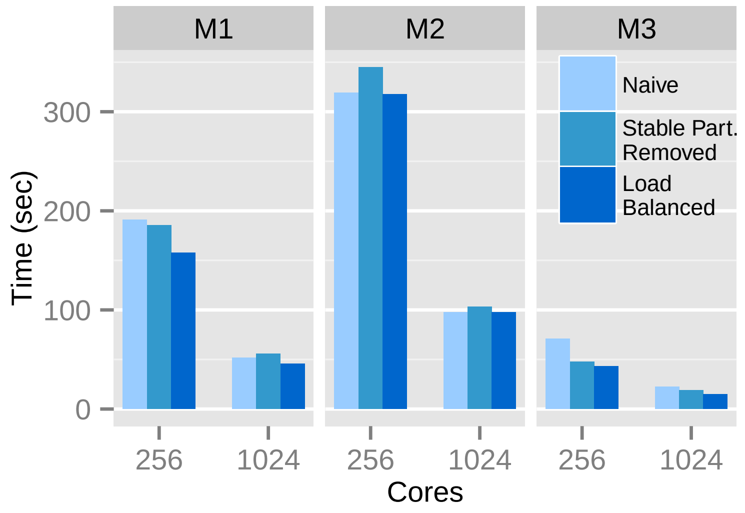


Figure 5.6: Performance gains due to load balancing for graphs M1-M3 using 256, 1024 processor cores.

Table 5.3: Kolmogorov Smirnov test values used to estimate the goodness of power law curve fit to the degree distribution of each graph. BFS is executed if K-S statistic value is less than 0.05.

Dataset	**K-S statistic**	**Run BFS iteration?**	**Correct Decision?**
M1	0.41	×	✓
M2	0.24	×	×
M3	0.39	×	✓
M4	0.31	×	✓
G1	**0.01**	✓	✓
G2	**0.03**	✓	✓
G3	0.21	×	✓
K1	**0.01**	✓	✓
K2	**0.01**	✓	✓

by Clauset *et al.* [79]. Table 5.3 shows the K-S statistic value computed using the degree distribution for all our graphs. For each of the graphs with scale-free topology (G1, G2, K1, K2), there is a clear distinction of these values against rest of the graphs. Based on these observations, we set a threshold of 0.05 to predict the scale-free structure of the underlying graph topology and execute a BFS iteration for such cases.

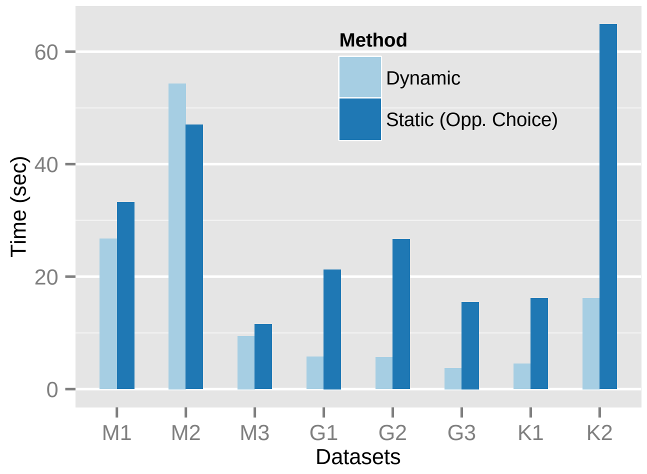
To measure the relative improvement obtained by running BFS iteration based on the prediction, we compare the runtime of this dynamic approach against our implementation that does not compute K-S statistics and is hard-coded to make the opposite choice, i.e., executing BFS iteration only for the graphs M1-M4, G3. This experiment, using 2025 processor cores, measures whether the prediction is correct and if correct, how much performance benefit do we gain against the opposite choice. As illustrated in figure 5.7a, we see positive speedups for all the graphs except M2. We see more than 3x performance gains for all the small world graphs as well as G3. For M1 and M3, we gained approximately 25% improvement in the runtime. This experiment confirms that using BFS to identify and exclude the largest component is much more effective for small world graphs while running BFS on large diameter graph such as G3 is not optimal. Moreover, using the degree distribution statistics, we can choose an optimal strategy for most of the graphs.

Note that computing the degree distribution of a graph and measuring K-S statistics adds an extra overhead to the overall runtime of the algorithm. We evaluate the additional overhead incurred by comparing the dynamic approach against the implementation which is hard-coded to make the same choice, i.e., execute BFS iteration only for G1-G2, K1-K2 (Figure 5.7b) using 2025 processor cores. The overhead varies from 60% for G1 to only 2% for M1. In general, we find this overhead to be relatively high for small-world graphs. Fitting the degree distribution curve against a power-law model is a sequential routine in our implementation, and it takes us about a second for scale-free graphs because they tend to have long-tailed degree distributions. We leave parallelizing and optimizing this routine as future work. Overall, we observe that the performance gains significantly outweigh the cost of computing the degree distribution and K-S test.

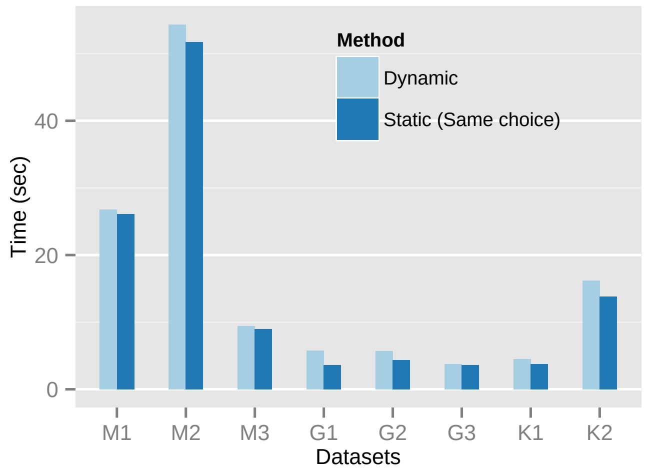
5.6.3 Strong Scaling

With the optimizations in place, we conducted strong scaling experiments on our algorithm. In this experiment, we use 256-4096 cores for G1-G3, K1, and M1-M3. Results for M4, the largest graph are discussed separately as we could not process it with fewer than 4096 cores. Graph K2 is ignored for this experiment because it has same topology as K1. In Fig. 5.8, we show the runtimes as well as speedups achieved by our algorithm. Most of these graphs cannot fit in the memory of a single node, therefore speedups are measured relative to the runtime on 256 cores. Ideal relative speedup on 4096 cores is 16. We achieve maximum speedup of more than 8x for the metagenomic graphs M1 and M2 and close to 6x speedup for small world graphs G1, G2 and K1. G3 shows limited scalability due to its much smaller size relative to other graphs. We are able to compute connectivity for our largest graph M4 in 215 seconds using 32761 cores (Table 5.4).

In section 5.4.1.3 we discussed how each iteration of our parallel SV algorithm uses



(a) Comparison of runtime of our algorithm making decision to run BFS dynamically versus the implementation which is hard-coded to make the opposite decision for all the graphs, using 2025 processor cores. For all the graphs but M2, our decision is correct (Table 5.3).



(b) Comparison of runtime of our algorithm making decision to run BFS dynamically versus the implementation hard-coded to make the same decision for all the graphs, using 2025 processor cores. The difference in the timings is the overhead of our prediction strategy.

Figure 5.7: Evaluation of prediction heuristics in our algorithm

parallel sorting to update the partition ids of the edges. As a majority of time of this algorithm is spent in performing sorting, we also execute a micro benchmark that sorts 2 billion randomly generated 64 bit integers using 256 and 4096 cores. Interestingly, we

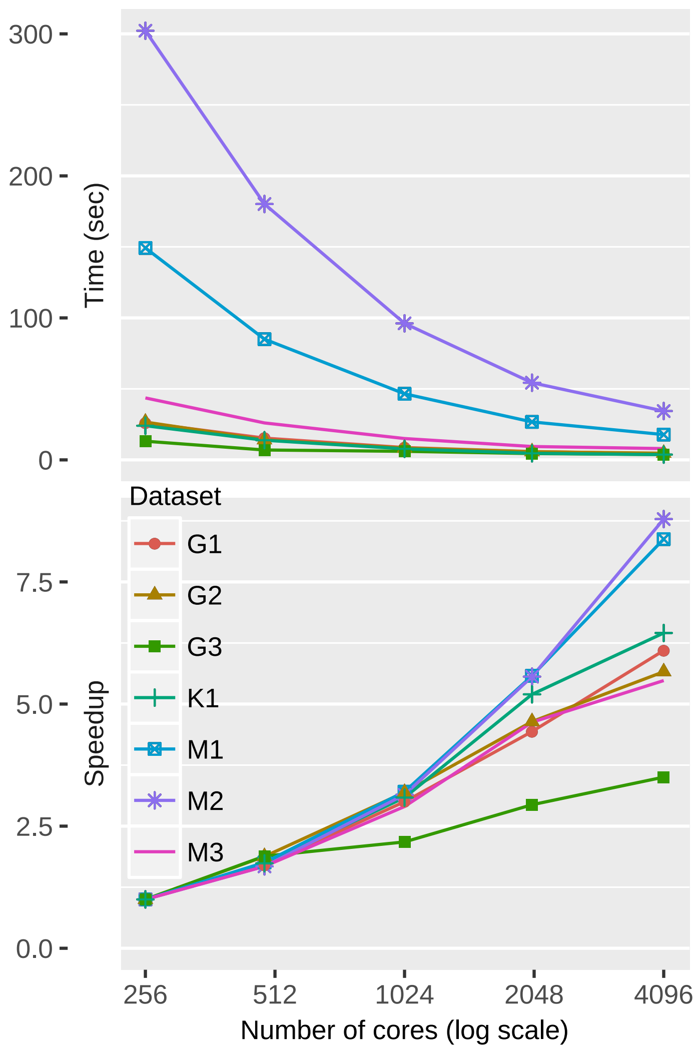


Figure 5.8: Strong scalability results of our algorithm on different graphs using 4096 cores. Speedups are computed relative to the runtime on 256 cores.

achieve speedup of 8.06 using our sample sorting method which is close to our scalability for M1 and M2. We anticipate that implementing more advanced sorting algorithms [88] may further improve the efficiency of our parallel SV algorithm.

5.6.4 Performance Anatomy

We also report the percentage of total execution time on 2025 cores that are attributable to each stage of our algorithm (Fig. 5.9). This figure is noteworthy especially for the graphs for which our algorithm chooses to execute BFS. For G1, G2, K1 and K2, more than 50%

Table 5.4: Timings for the largest graph M4 with increasing processor cores

Cores	8281	16384	32761
Time for M4 (sec)	429.89	291.19	214.56

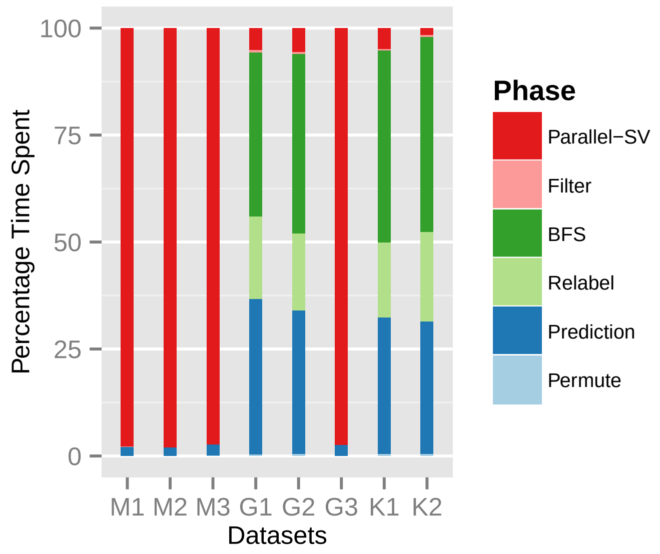


Figure 5.9: Percentage time spend in different stages by the algorithm for different graphs using 2025 cores. BFS is executed only for graphs G1, G2, K1 and K2.

of the total percentage of time is devoted to predicting the graph structure and relabeling the vertices before running the parallel BFS and SV algorithm. This figure is not meant to convey the true overhead due to the relabel and prediction operations individually, as the time for relabeling is reduced after we sort the edges during the prediction stage (Section 5.4.2). Further, we measure the percentage time spent in the sorting operations in our parallel-SV algorithm for the graphs M1, M2, M3 and G3. As we expected, this measure is high and ranges from 91% - 94% for all the four graphs.

5.6.5 Comparison with Previous Work

We achieve notable speedups when the performance of our algorithm is compared against the state-of-the-art Multistep algorithm [75] using 2025 cores. As before, we begin count-ing the time once the graph edge list is read into the memory in both cases. We ran the Multistep algorithm with one process per physical core as we observed better performance doing so than using hybrid MPI-OpenMP mode. Also, because the Multistep method expects the vertex ids to be in the range 0 to $|V| - 1$, we inserted our vertex relabeling routine in their implementation in order to run the software. Figure 5.10 shows the comparison of our approach against the MultiStep method. We see > 1 speedups for our method in all the graphs except G1. The speedup achieved ranges from 1.1x for K2 to 24.5x for G3. The speedup roughly correlates with the diameter of the graphs. The improvements achieved for the graphs M1, M2, M3 and G1 can be attributed to two shortcomings in the Multistep approach: 1) It executes BFS for computing the first component in all the graphs. BFS attains limited parallelism for large diameter graphs due to small frontier sizes. 2) It uses the label propagation technique to compute other components which in the worst case can take as many iterations as the diameter of the graph to reach the solution.

We could not compare our approach against the distributed-memory graph contraction algorithm [73] proposed to solve the connectivity problem, as the implementation is not open-source. Based on their experiment description, the graph contraction algorithm showed strong scalability only till 32 cores. Other distributed graph frameworks such as GraphX [69], and FlashGraph [89] based on in-memory Apache Spark and external-memory framework, respectively, can compute the connectivity of large-scale graphs as well. Slota *el al.* [75] show that their Multistep algorithm achieves superior performance against both of these methods. Because our algorithm performs better than Multistep, we skip a direct comparison against GraphX and FlashGraph.

5.6.6 Comparison with Sequential Implementation

We examine the performance of our algorithm against the best known sequential implementation for computing connectivity, for graph instances which can fit in the single node memory (64 GB) - these are relatively small. Previous works [65, 90] have shown that

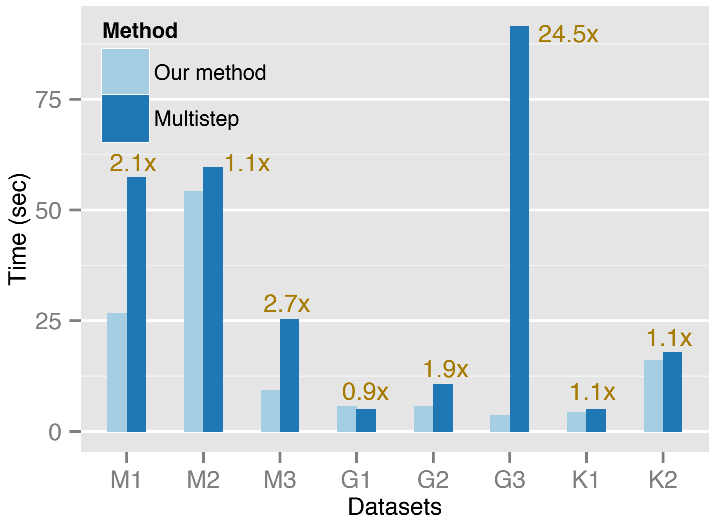


Figure 5.10: Performance comparison of our algorithm against the Multistep method [75] using multiple graphs with 2025 cores.

the Rem's method [91] based on the union-find approach achieves the best sequential performance. The sequential implementation we use in this algorithm was obtained from the authors of [65]. Again, because the disjoint-set structure used in the algorithm requires the vertices to be numbered from 0 to $n-1$, we placed our relabeling routine in the implementation. This experiment uses graphs M3 and G3, as all the other graphs require more than 64 GB memory. We also add a Kronecker graph of scale 25 ($m = 537$M, $n = 17$M) to include a short diameter graph instance. Results of this experiment are shown in Table 5.5. For these three graphs, our algorithm selects BFS iteration for Kronecker graph only. For the Kronecker graph, our algorithm achieves a 100x speedup using 1024 cores. For the other graphs, M3 and G3, where the SV algorithm is selected, the speedup decreases with respect to the sequential algorithm - which is partially due to the fact that the algorithm is not work optimal.

5.6.7 Comparison with Shared-memory Implementations

The objective of the following comparative discussion between the distributed and shared-memory algorithms is not only to discuss the performance difference where shared-memory implementations tend to get good scaling per core, rather it is to highlight some of the constraints that shared-memory implementations have in contrast to their distributed counterparts.

Shared-memory parallel methods [65, 66] exhibit good speedups over the best sequential implementation. It is therefore of no surprise to us that these algorithms can outperform our algorithm, especially for small to mid-range graphs. However, there are numerous problem scales that these shared memory algorithms cannot cope with due to the size of the graph; whereas our algorithm can easily deal with such networks. Our parallel algorithm utilizes bulk synchronous communication instead of the fast asynchronous communication found in shared-memory frameworks. While such communications are inherently slower, they do enable processing larger networks. Consider the largest network analyzed (Table 5.2): metagenomic graph M4 which has 53.6 billion edges and an equal number of vertices. Processing this graph in memory requires at least the following amount of memory: $2 \cdot (|V| + |E|) \times 8$ bytes. This assumes that the graph requires $|V|$ elements for the vertices and $2 \cdot |E|$ elements for the edges [3]. Also, $|V|$ integers are required for tracking the connected component labels. Given the size of the graph, 4 byte integers are not large enough to store all the unique keys and as such this requires using 8 byte integers. For the M4 network, a total of 1.7 TB DRAM is needed. As the sequencing cost continues to decline much faster than Moore's law [92], we envision the need to analyze even larger metagenomic graphs that require even more memory, in the near future. The problems of optimizing distributed-memory parallel algorithms while trying to attain peak performance continues to be an important challenge and one that deserves additional attention, especially the ability to reduce the overhead of communication.

Overall, we see that our proposed algorithm and the optimizations help us improve the state-of-the-art for distributed-memory parallel solution to the graph connectivity problem. Simple and fast heuristics to detect the graph structure enables our algorithm to choose the

[3]Recall that these are un-directed edges and it is customary in CSR, Compressed Sparse Row, format to store both directions of the edge

Table 5.5: Performance comparison against Rem's sequential connectivity algorithm [90, 91] using 1024 cores.

Dataset	**Fastest Seq. Time (s)**	**Speedup**		
		p = 64	256	1024
Kronecker (25)	228.8	10.1	34.3	100.6
M3	406.2	2.5	9.3	27.0
G3	45.9	0.9	3.5	7.6

appropriate method dynamically for computing connectivity. This approach enabled us to compute connectivity for a graph with more than 50 billion edges and 300 million components in less than 4 minutes. The speedup we achieve over the state-of-the-art algorithm ranges from 1.1x to 24.5x.

5.7 Conclusion

In this work, we presented an efficient distributed memory algorithm for parallel connectivity, based on the Shiloach-Vishkin PRAM algorithm. We proposed an edge-based adaptation of this classic algorithm and optimizations to improve its practical efficiency in distributed systems. Our algorithm is capable of finding connected components in large undirected graphs. We show that a dynamic approach that analyzes the graph and selectively uses the parallel BFS and SV algorithms achieves better performance than a static approach using one or both of these two methods. The dynamic approach prefers BFS execution only for a large short-diameter graph component. Our method is efficient as well as generic, as demonstrated by the strong scalability of the algorithm on a variety of graph types. We also observed better performance when compared to a recent state-of-the-art algorithm. The measured speedup is significant, particularly in the case of large diameter graphs.

CHAPTER 6
CONCLUSIONS

In this dissertation, we presented distributed memory parallel algorithms for solving string and graph problems stemming from applications in computational biology. Our algorithms are designed to be able to scale to much larger problems compared to prior approaches, a necessity motivated by the rapid increase in the sizes of genomic data sets.

Suffix arrays and trees are fundamental string data structures which lie at the foundation of many string algorithms, with important applications in text processing, information retrieval, and computational biology. Conversely, the parallel construction of these indices is an actively studied problem. However, prior approaches lacked good worst-case run-time guarantees and exhibit poor scaling and overall performance.

In order to be able to scale to very large inputs, our algorithms are designed so that all data and data structures are fully distributed, requiring no more than $O(n/p)$ memory per processor - a key constraint that allows our algorithms to scale to arbitrarily large inputs given enough compute nodes. Surprisingly, most prior approaches do not follow this constraint and require up to $O(n)$ memory per node - a drastic limitation for scalability.

Despite this constraint on the distribution, our distributed-memory parallel algorithms for the construction of suffix arrays, LCP arrays, and suffix trees clearly advance the state-of-the-art. Our construction algorithms improve the overall theoretical runtime complexity, and our implementation exhibit far superior practical performance: outperforming competing approaches by as much as an order of magnitude.

In Chapter 2, we introduced new 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. We presented several algorithm engineering techniques that improve performance in practice and demonstrated the the construction of suffix and LCP arrays of the human genome in less than 7.5 seconds on 1,024 Intel Xeon cores. Our implementation reaches speedups of above $110\times$ over *divsufsort*, one of the fastest known sequential implementations. 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.

In Chapter 3, we presented a work-optimal distributed memory parallel algorithm for the construction of suffix trees. In contrast to the linear work performed by the algorithm, all previous distributed memory algorithms exhibit quadratic worst-case complexity. Our algorithm also improves prior state-of-the-art for distributed memory in terms of practical performance. We illustrate performance of the algorithm on the human genome, for which we construct the suffix and LCP arrays in 7.5 seconds from the genome, followed by construction of the suffix tree in less than 2 additional seconds, on 64-node dual 8-core Xeon CPU cluster. Furthermore, we demonstrate that our MPI based implementation performs better in shared memory than state-of-the-art shared memory algorithms, and can scale to a large number of cores in distributed memory.

In Chapters 2 and 3 we discussed in depth how to efficiently construct suffix arrays and trees. In distributed memory, and for large problems, these classical data structures taken as-is exhibit drawbacks for their use for downstream applications. Suffix trees require a lot of memory to fully store, which limits their scalability to very large problems. The classic sequential query algorithms for suffix arrays and LCP arrays do not generalize well to their distributed representation - as they would incur massive communication latency costs at almost every step. This lead us to develop a novel distributed string index: the *Distributed Enhanced Suffix Array (DESA)*. In Chapter 4, we introduced this new distributed data structure, which is based on the suffix and LCP arrays, and adds additional data structures. The DESA allows efficient construction and querying, all while requiring at most $O(n/p)$ memory per process. We presented efficient distributed-memory parallel algorithms for querying, as well as for the efficient construction of this distributed index.

We demonstrated the performance of our algorithms by comparing against other sequential approaches, and additionally demonstrated strong scalability to over 1500 cores.

Finally, in Chapter 5, we presented a distributed memory algorithm for finding the connected components in large edge-list graphs. This work was motivated by a *grand challenge* metagenomics problem: by finding the connected components in the de Bruijn graph, we can enable downstream analysis on the independent components. Our solution was able to find the connected components in the grand challenge 1.8 billion reads metagenomics data set - corresponding to a graph with 135 billion edges - in just 22 minutes on 1280 Xeon cores. Furthermore, we showed that our algorithm can be extended to and performs well also on general graphs. For general graphs, a dynamic hybrid approach between our connectivity algorithm and a distributed BFS achieves performance better than a static approach using either one of these two methods. The dynamic approach prefers BFS execution only for a large short-diameter graph component. The hybrid method is efficient as well as generic, as demonstrated by the strong scalability of the algorithm on a variety of graph types. We also observed better performance when compared to a recent state-of-the-art algorithm. The measured speedup is significant, particularly in the case of large diameter graphs.

REFERENCES

[1] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[2] D. E. Knuth, J. H. Morris Jr, and V. R. Pratt, “Fast pattern matching in strings,” *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[3] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[4] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[5] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[6] U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[7] J. Kärkkäinen and P. Sanders, “Simple linear work suffix array construction,” in *Automata, Languages and Programming*, Springer, 2003, pp. 943–955.

[8] P. Ko and S. Aluru, “Space efficient linear time construction of suffix arrays,” in *Combinatorial Pattern Matching*, Springer, 2003, pp. 200–210.

[9] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, “A taxonomy of suffix array construction algorithms,” *ACM Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.

[10] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[11] 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*, Springer, 2001, pp. 181–192.

[12] P. Flick and S. Aluru, “Parallel distributed memory construction of suffix and longest common prefix arrays,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 16.

[13] ——, “Parallel construction of suffix trees and the all-nearest-smaller-values problem,” in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, IEEE, 2017, pp. 12–21.

[14] ——, “Distributed enhanced suffix arrays: Efficient algorithms for construction and querying,” in *SPAA’19 (under review)*.

[15] P. Flick, C. Jain, T. Pan, and S. Aluru, “A Parallel Connectivity Algorithm for de Bruijn Graphs in Metagenomic Applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 15.

[16] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, “An adaptive parallel algorithm for computing connected components,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, 2017.

[17] P. Flick, C. Jain, T. Pan, and S. Aluru, “Reprint of a parallel connectivity algorithm for de bruijn graphs in metagenomic applicationsi,” *Parallel Computing*, vol. 70, pp. 54–65, 2017.

[18] Y. Mori, *Libdivsufsort*, `https://github.com/y-256/libdivsufsort`.

[19] V. Osipov, “Parallel suffix array construction for shared memory architectures,” in *String Processing and Information Retrieval*, Springer, 2012, pp. 379–384.

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

[21] N. Futamura, S. Aluru, and S. Kurtz, “Parallel suffix sorting,” pp. 76–81, 2001.

[22] J. L. Bentley and R. Sedgewick, “Fast algorithms for sorting and searching strings,” in *SODA*, vol. 97, 1997, pp. 360–369.

[23] 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*, IEEE, 2014, pp. 334–337.

[24] F. Kulla and P. Sanders, “Scalable parallel suffix array construction,” *Parallel Computing*, vol. 33, no. 9, pp. 605–612, 2007.

[25] R. Homann, D. Fleer, R. Giegerich, and M. Rehmsmeier, “Mkesa: Enhanced suffix array construction tool,” *Bioinformatics*, vol. 25, no. 8, pp. 1084–1085, 2009.

[26] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, no. 1, pp. 33–50, 2004.

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

[28] K.-B. Schürmann and J. Stoye, "An incomplex algorithm for fast suffix array construction," *Software: Practice and Experience*, vol. 37, no. 3, pp. 309–329, 2007.

[29] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the gpu," in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 197–206.

[30] N. J. Larsson and K. Sadakane, "Faster suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 258–272, 2007.

[31] 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*, ACM, 1972, pp. 125–136.

[32] 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*, ACM, 1991, pp. 3–16.

[33] 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*, Springer, 2007, pp. 459–470.

[34] . G. P. Consortium *et al.*, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, pp. 56–65, 2012.

[35] 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*, vol. 497, no. 7451, pp. 579–584, 2013.

[36] 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*, ACM, 2009, pp. 827–840.

[37] 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*, vol. 5, no. 1, pp. 49–60, 2011.

[38] 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*, ACM, 2013, pp. 211–216.

[39] O. Berkman, B. Schieber, and U. Vishkin, "Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values," *Journal of Algorithms*, vol. 14, no. 3, pp. 344–370, 1993.

[40] X. He and C.-H. Huang, "Communication efficient bsp algorithm for all nearest smaller values problem," *Journal of Parallel and Distributed Computing*, vol. 61, no. 10, pp. 1425–1438, 2001.

[41] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, "Parallel construction of a suffix tree with applications," *Algorithmica*, vol. 3, no. 1-4, pp. 347–365, 1988.

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

[43] C. Iliopoulos and W. Rytter, "On parallel transformations of suffix arrays into suffix trees," in *15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, Citeseer, 2004.

[44] J. Shun and G. E. Blelloch, "A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, p. 8, 2014.

[45] T. Hagerup and C. Rüb, "Optimal merging and sorting on the erew pram," *Information Processing Letters*, vol. 33, no. 4, pp. 181–185, 1989.

[46] J. JaJa and K. W. Ryu, "Optimal algorithms on the pipelined hypercube and related networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 5, pp. 582–591, 1993.

[47] D. Kravets and C. G. Plaxton, "All nearest smaller values on the hypercube," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, no. 5, pp. 456–462, 1996.

[48] J. Labeit, J. Shun, and G. E. Blelloch, "Parallel lightweight wavelet tree, suffix array and FM-index construction," *Journal of Discrete Algorithms*, vol. 43, pp. 2–17, 2017.

[49] 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*, ACM, 2009, p. 61.

[50] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software: Practice and Experience*, vol. 44, no. 11, pp. 1287–1314, 2014.

[51] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed text indexes: From theory to practice," *Journal of Experimental Algorithmics (JEA)*, vol. 13, p. 12, 2009.

[52] H. Nordberg, M. Cantor, S. Dusheyko, S. Hua, A. Poliakov, I. Shabalov, T. Smirnova, I. V. Grigoriev, and I. Dubchak, "The Genome Portal of the Department of Energy Joint Genome Institute: 2014 updates," *Nucleic Acids Research*, vol. 42, no. D1, pp. D26–D31, 2014.

[53] A. C. Howe, J. K. Jansson, S. A. Malfatti, S. G. Tringe, J. M. Tiedje, and C. T. Brown, "Tackling Soil Diversity with the Assembly of Large, Complex Metagenomes," *Proceedings of the National Academy of Sciences*, vol. 111, no. 13, pp. 4904–4909, 2014.

[54] A. Buluç and K. Madduri, "Parallel Breadth-First Search on Distributed Memory Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011, p. 65.

[55] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, IEEE, 2013, pp. 1618–1627.

[56] Y. Shiloach and U. Vishkin, "An O(logn) Parallel Connectivity Algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[57] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de Bruijn Graphs to Genome Assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[58] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User's Group (CUG)*, 2010.

[59] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing Connected Components on Parallel Computers," *Communications of the ACM*, vol. 22, no. 8, pp. 461–464, 1979.

[60] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick, "Connected Components on Distributed Memory Machines," in *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994, volume 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1994, pp. 1–21.

[61] L. Buš and P. Tvrdík, "A Parallel Algorithm for Connected Components on Distributed Memory Machines," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, 2001, pp. 280–287.

[62] S. Goddard, S. Kumar, and J. F. Prins, "Connected Components Algorithms for Mesh-connected Parallel Computers," *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, vol. 30, pp. 43–58, 1994.

[63] D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[64] D. A. Bader and G. Cong, "A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, IEEE, 2004, p. 38.

[65] M. M. A. Patwary, P. Refsnes, and F. Manne, "Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, IEEE, 2012, pp. 827–835.

[66] J. Shun, L. Dhulipala, and G. Blelloch, "A Simple and Practical Linear-Work Parallel Algorithm for Connectivity," in *Proceedings of the 26th ACM symposium on Parallelism in Algorithms and Architectures*, ACM, 2014, pp. 143–153.

[67] G. Cong and P. Muzio, "Fast Parallel Connected Components Algorithms on GPUs," in *Euro-Par 2014: Parallel Processing Workshops*, Springer, 2014, pp. 153–164.

[68] K. Ueno and T. Suzumura, "Highly Scalable Graph Search for the Graph500 Benchmark," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2012, pp. 149–160.

[69] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph Processing in a Distributed Dataflow Framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[70] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ACM, 2015, p. 1.

[71] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed Graph-parallel Computation on Natural Graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[72] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed Graphlab: A Framework for Machine Learning and Data Mining in the Cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[73] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of Connected-component Labeling algorithms for Distributed-memory Systems," *Parallel Computing*, vol. 44, pp. 53–68, 2015.

[74] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, IEEE, 2014, pp. 550–559.

[75] G. M. Slota, S. Rajamanickam, and K. Madduri, "A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization," in *Parallel and Distributed Processing Symposium, 2016 IEEE 30th International*, IEEE, 2016.

[76] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[77] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *International Journal of High Performance Computing Applications*, p. 1 094 342 011 403 516, 2011.

[78] A.-L. Barabási, R. Albert, and H. Jeong, "Scale-free characteristics of random networks: The topology of the world-wide web," *Physica A: Statistical Mechanics and its Applications*, vol. 281, no. 1, pp. 69–77, 2000.

[79] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law Distributions in Empirical Data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[80] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy.," *ICWSM*, vol. 10, no. 10-17, p. 30, 2010.

[81] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[82] J. Shun, "An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2015, pp. 1095–1104.

[83] A Gordon and G. Hannon, "Fastx Toolkit," *Computer program distributed by the author, website http://hannonlab.cshl.edu/fastx_toolkit/index. html [accessed 2015–2016]*, 2010.

[84] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, “Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems,” in *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, ACM, 2016, pp. 422–433.

[85] L. M. Rodriguez-R and K. T. Konstantinidis, “Estimating Coverage in Metagenomic Data Sets and Why it Matters.,” *The ISME journal*, vol. 8, no. 11, p. 2349, 2014.

[86] B. Jenkins, “Hash functions,” *Dr Dobbs Journal*, vol. 22, no. 9, pp. 107–+, 1997.

[87] T. Nepusz, *Fitting Power-law Distributions to Empirical Data*, `https://github.com/ntamas/plfit`, 2016.

[88] E. Solomonik and L. Kale, “Highly Scalable Parallel Sorting,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April, pp. 1–12.

[89] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, “Flashgraph: Processing Billion-node Graphs on an Array of Commodity SSDs,” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 45–58.

[90] M. M. A. Patwary, J. Blair, and F. Manne, “Experiments on Union-find Algorithms for the Disjoint-set Data Structure,” in *Experimental Algorithms*, Springer, 2010, pp. 411–423.

[91] E. W. Dijkstra, *A discipline of programming*. prentice-hall Englewood Cliffs, 1976, vol. 1.

[92] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky, *et al.*, “The Real Cost of Sequencing: Scaling Computation to Keep Pace with Data Generation,” *Genome biology*, vol. 17, no. 1, p. 1, 2016.