Location-Sensitive String Problems in MPC

Jacob Gilbert*
jgilber8@umd.edu
University of Maryland
College Park, Maryland, USA

MohammadTaghi Hajiaghayi *
hajiaghayi@gmail.com
University of Maryland
College Park, Maryland, USA

Hamed Saleh*
hameelas@gmail.com
University of Maryland
College Park, Maryland, USA

Saeed Seddighin
saeedreza.seddighin@gmail.com
Toyota Technological Institute at Chicago
Chicago, Illinois, USA

ABSTRACT

A suffix tree is a trie-like data structure that stores every suffix of an input string of length n. Finding the *Suffix Tree* of a given string is a well-studied and classic problem. A compressed suffix tree is constructible in $O(n)$ time using the well-known algorithm of McCreight (JACM, 1976) [24]. Suffix trees alongside with hashing are two powerful tools in solving *location-sensitive* string problems. Many well-studied fundamental string problems such as *String Matching*, *Longest Palindrome Substring* (LPS), *Longest Common Substring* (LCS), and *Longest Common Prefix* (LCP) queries are location-sensitive and have linear time solutions via reductions to suffix tree.

Inspired by suffix trees, we study location-sensitive string problems LCP, LPS, and LCS in the *Massively Parallel Computation* (MPC) model. Our algorithms extensively utilize a novel data structure for answering $O(n^{1+\epsilon})$ arbitrary LCP queries, called an LCPQ oracle, in $O(1)$ rounds and with $\widetilde{O}(n^{1+\epsilon})$ total memory. Using our LCPQ oracle we are able to give the first $O(1)$-round, $\widetilde{O}(n)$ total memory algorithm in MPC for LPS and an $O(1)$-round LCS solution using $\widetilde{O}(n^{1+\epsilon})$ total memory, beating previous state-of-the-art techniques for both problems. Furthermore, we give an $O(1/\epsilon)$-round MPC algorithm for constructing suffix arrays and suffix trees utilizing our LCPQ oracle, and demonstrate reductions for LPS and LCS to suffix tree in $O(1)$ rounds of MPC. Finally, we show that in the *Adaptive Massively Parallel Computation* (AMPC) model, we can build a fully-functional suffix tree for a given input string in $O(1)$ rounds and with $\widetilde{O}(n)$ total memory for any constant $\epsilon > 0$.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; **Data structures design and analysis**.

KEYWORDS

Data Structures, Massively Parallel Algorithms. Suffix Trees

*Partially supported by DARPA QuICC, NSF AF:Small 2218678, and NSF AF:Small 2114269 grants.

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591090

ACM Reference Format:
Jacob Gilbert, MohammadTaghi Hajiaghayi, Hamed Saleh, and Saeed Seddighin. 2023. Location-Sensitive String Problems in MPC. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3558481.3591090

1 INTRODUCTION

One of the most well-studied topics in recent computer science history is the algorithmic aspects of string problems. The *longest palindrome substring* (LPS), the *longest common substring* (LCS), *string matching*, and *edit distance* are some of the more famous problems in this category. Some of these problems such as string matching, LCS, LPS, and *longest common prefix* (LCP) are *location-sensitive*, i.e. solutions to these problems require compiling information from continuous sequences of characters of the input string(s) and are dependent on the order of said characters and sequences. Efforts to solve these problems led to the discovery of several fundamental techniques such as *hashing algorithms* and *suffix trees* which are useful for solving location-sensitive problems. Both of these tools have numerous applications in several fields including DNA-sequencing, compiler design, anti-virus softwares, etc.

Suffix trees are tries made of the sorted suffixes of a given string, and computing a suffix tree is a classic problem in computer science literature. The concept was first introduced by Weiner [27], and McCreight (JACM, 1976) [24] showed that it is possible to create a *compressed* suffix tree of total size $O(n)$. Later, Ukkonen [26] introduced an online $O(n)$-time algorithm that dynamically maintains an updated suffix tree as new characters are appended to the input string. There are numerous applications for the suffix tree data structure such as *exact and approximate string matching*, *matching statistics*, *Lempel-Ziv decomposition*, and *longest repeated substring* [17]. Inspired by suffix trees, we discuss two particular applications of interest [13] at length in this paper:

1 Longest Palindrome Substring (LPS), and
2 Longest Common Substrings (LCS).

In addition to LPS and LCS, we consider the problem of Longest Common Prefix (LCP) queries, which is often a crucial step in suffix tree construction and can also be answered in constant time using an already-constructed suffix tree [19].

Designing algorithms which are sublinear in space is crucial since the input data usually does not fit in the memory of available systems. In recent years, with the abundance of data and the increasing demand for large scale data processing, the size of datasets

easily surpasses that of the typical machine's memory. Examples of the problem domains vary from simple batch applications such as Diff, Awk, and Sed to large-scale applications like database queries, search engines, and even genomics [5, 25], and efficiently analyzing such datasets has great economic and social impact. However, to address any problem subject to this restriction efficiently, we need alternative models of computation as the traditional RAM model is no longer feasible in practice. One approach to overcome this obstacle is to utilize a large pool of typical machines that have a substantially smaller amount of memory comparing to the input size. This simple idea is the key ingredient behind the modern distributed processing frameworks such as MapReduce, Hadoop, and Spark that are widely deployed across the industry at very large scales. The *Massively Parallel computation* (MPC) model is proposed by Karloff, Suri, and Vassilvitskii [20], and enhanced in subsequent works [1, 6, 16], to provide a theoretical foundation for algorithms implemented on such frameworks.

In the MPC model, problem data of size $O(n)$ is distributed among a set of machines each with sublinear memory, and algorithms proceed in several rounds. In each round, machines perform an arbitrary amount of computation on their local data independently. At the end of each round, machines can communicate with each other. Since the communication phase is often a major performance bottleneck in real world scenarios, the main goal in designing MPC algorithms is to optimize the round complexity while keeping total memory across all machines linear. Note that for some $0 < \epsilon < 1$, each machine has $O(n^{1-\epsilon})$ memory; during a single communication round, each machine may send and receive any number of messages such that the total size of the messages fits within its local memory constraints. In some of our algorithms below, we require that each machine may have a large enough local memory to store a message from every other machine in a single round of communication. To this end, we must make sure that the local memory of a machine $O(n^{1-\epsilon})$ is larger than the number of machines $O(n^{\epsilon})$, and so, we restrict ϵ to $0 < \epsilon \leq .5$. This restriction is well-motivated for the MPC model since with modern computing resources, the number of machines does not exceed the size of the local memory of each machine.

In this paper, we introduce a massively parallel framework for solving a wide range of *location-sensitive* string problems such as *string matching*, LCS, and LPS, which usually are easily solvable using either a *suffix tree* (or *hashing*) in the sequential setting. We simulate powerful tools such as the *suffix tree* of the given string and a novel MPC data structure for handling batches of LCP queries. We call this data structure an LCPQ oracle, and it is the first MPC data structure able to efficiently answer a large (even super-linear) number of LCP queries in a small number of communication rounds. In addition to demonstrating constant-round MPC reductions from the aforementioned problems of LPS and LCS to suffix trees, our framework provides the means to give algorithms using $O(1)$ communication rounds with tight $\widetilde{O}(n)$ total memory for LPS and $\widetilde{O}(n^{1+\epsilon})$ memory for LCS by applying a technical analysis of string periodicity in each problem. Our novel LCPQ oracle is critical to the improved algorithms for LPS and LCS as well as the MPC suffix tree simulation.

Beyond discussing LCP, LPS and LCS, we also discuss improved algorithms for *suffix tree* construction. In the stronger, recently-proposed *Adaptive* MPC (AMPC) model [9], each machine is subject to the same local memory and communication limitations as in MPC. However, the model is stronger because in each round, all communications can be stored in a large $O(n)$ shared memory that allows adaptive reads by all machines. We will utilize the shared memory to store *hashes* of suffixes, which is an interesting strategy that will allow us to avoid sending large messages between machines when comparing suffixes and instead just compare hash values of constant size in the shared memory. With this simpler communication, we show how to create an improved LCPQ oracle of a given string in AMPC and then subsequently the *suffix tree* in $\tilde{O}(n)$ memory and constant rounds [1].

1.1 Further Related Works

Among others, a specific family of graph problems known as *Locally Checkable Labeling* (LCL) problems – which includes vertex coloring, edge coloring, maximal independent set, and maximal matching to name a few – admit highly efficient MPC algorithms, and have been heavily studied during recent years [3, 4, 7, 8, 10, 11, 14, 15]. Another interesting family of problems studied in the MPC model are string problems. The *String Matching* problem, that is to find all occurrences of a pattern string in a text, has a wide range of applications on massive datasets from analyzing genomic data to Search Engine Indexing.

In a previous work, Hajiaghayi, Saleh, Seddighin, and Sun [18] show that the *string matching* problem and also some of its variants can be solved efficiently in MPC. In particular, they show an $O(1)$ rounds algorithms for String Matching and Integer Convolution (using FFT) in MPC, and as a result they give efficient algorithms for String Matching with *wildcard* characters. Some building blocks in the string matching solution such as fast batch substring hash queries, which is a special case of block-based data structures, are also used in this paper (defined in later Section 4.1). Furthermore, our methods in this paper can be applied to the string matching problem (without wildcards) as an application of suffix trees [24].

PRAM solutions to these problems also are heavily related to our work. Apostolico, Breslauer and Galil [2] give a PRAM solution for LPS with $O(n^2)$ total memory needed. While this solution requires more memory and rounds than we want if we were to convert it directly to MPC, our own optimal LPS algorithm utilizes their main idea that strings with multiple palindromes must be periodic. Iliopoulos and Rytter [19] build a suffix tree from a simpler data structure, the suffix array, in PRAM in tight $\tilde{O}(n)$ total memory with $O(\log n)$ rounds. We follow this construction technique as well for our own suffix tree algorithm, with significant changes so that the transformation works in $O(1)$ rounds. Several other papers discuss related problems in PRAM such as the *longest common subsequence* problem [22] and suffix tree construction for binary strings [12].

1.2 Problem Definition

A *string* $s \in \Sigma^n$ of length n is a sequence of n characters $s_0 s_1 \ldots s_{n-1}$ from alphabet Σ. For any pair of integers $0 \leq l < r \leq n$, we use

[1] $\tilde{O}(n)$ is used to denote a linear size with a multiplicative polylogarithmic factor, i.e. $O(n\text{polylog}(n)) = \tilde{O}(n)$ and is used for convenience in the rest of the paper.

brackets and parentheses[2], i.e., $s[l : r)$, to denote a *substring* of string s which is a consecutive interval of characters $s_l s_{l+1} \dots s_{r-1}$ from index l to index $r-1$. The length of a substring $s[l : r)$ is equal to $r - l$. Every substring $s[l : n)$ which ends with s_{n-1} is called a *suffix* of s. Similarly, every substring $s[0 : r)$ which starts with s_0 is called a *prefix* of s.

One of the primitive tools in solving classical string problems is the **longest common prefix** (LCP) query. For two input strings of length n denoted by s and s', we define $\mathsf{LCP}_{s,s'}(i, j)$ as the length of the longest prefix shared between two suffixes $s[i : n]$ and $s'[j : n]$. For example, [23] uses an array containing $\mathsf{LCP}_{s,s}(\pi_i, \pi_{i+1})$ for every $0 \le i < n-1$ as an auxiliary data structure alongside suffix arrays to mimic the capabilities of suffix trees. In the sequential setting, we can find the solution of LCP queries in $O(\log n)$ time by performing a *binary search* on the solution size and comparing the hashes of substrings of each suffix. Alternatively, if we already have a suffix tree built for string $s\#s'$, we may find the *lowest common ancestor* (LCA) in the suffix tree of the two suffixes $s[i : n]\#s'$ and $s'[j : n]$ (see following definition of suffix tree for more details).

Definition 1.1. Given strings s and s' each of length n, the LCP of indices i and j, denoted by $\mathsf{LCP}_{s,s'}(i, j)$, is equal to the length of the **longest common prefix** of suffixes $s[i : n)$ and $s'[j : n)$.

In the **longest common substring** problem (LCS), the input consists of two strings s and s' and our goal is to find the longest substring which is shared between the two strings. We assume that s and s' have the same length, which we denote by n. We use Σ to denote the alphabet of the strings.

In the **longest palindrome substring** problem (LPS), the goal is to find the longest substring of a given string s which reads the same both forward and backward. Let $\bar{s}$ denote the reverse of string s. Formally, a string s is a *palindrome* if $\bar{s} = s$. The length of s is also denoted by n and its alphabet is denoted by Σ.

The **suffix tree** of a given s is a rooted *trie*-like data structure in which every edge is labeled with a string, and it stores every *suffix* of s, i.e., $s[0 : n), s[1 : n), \dots, s[n-1 : n)$. Every node in the suffix tree represents a substring of s which is equal to the concatenation of the labels in the path from the root to this node. There is a leaf in the suffix tree for every *suffix* $s[i : n)$ $(0 \le i < n)$, and the concatenation of the labels in the path from the root to this leaf results in suffix $s[i : n)$. See Figure 1 for an example for string $s =$ "bobocel". Often we will need a suffix tree for two strings, and we follow the traditional approach of building a suffix tree on the concatenation of the two strings with a special character such as '#' between each string, e.g. $s\#s'$ represents the concatenation of strings s and s'.

Since the implementation of *compressed* suffix trees is complicated, Manber and Meyers [23] introduced a simpler and consequently less flexible data structure called **suffix array** as an alternative representation of suffix trees. Let $[n)$ denote the set of numbers $\{0, 1, 2, \dots, n-1\}$. The suffix array of s is a permutation $\pi : [n) \to [n)$ which represents an order of the suffixes of s that is sorted in the lexicographical order, i.e., $s[\pi_i, n)$ is lexicographically smaller than $s[\pi_{i+1}, n)$ for every $i \in [n-1)$. When it is clear from context, we may simply refer to the ith suffix according to the order of the suffix array, i.e. $s[\pi_i, n)$, as suffix i. In the sequential setting, the suffix array of a given string s can be found in $O(n \log n)$ running time without using suffix tree by performing a *radix sort* for sorting the prefixes of length 2^i of every suffix at step i for a total of $O(\log n)$ steps.

[2]This notation is often used for intervals $[l, r)$ that include l and exclude r.

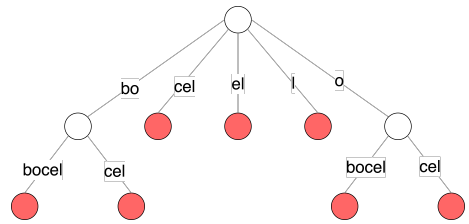

Figure 1: An example of *suffix tree* for example string "bobocel".

2 KMR-BASED SOLUTIONS

We start with a solution for location-sensitive problems using an existing, well-known KMR-based technique of [21]. While technically simple, we show these results for the sake of completeness, and the KMR algorithm gives us a point of comparison for our own results. We give specialized algorithms for LPS and LCS, respectively, that have a lower number of communication rounds (the LPS algorithm also improves the total memory size) (see Section 6 for improved LPS algorithm). Additionally the suffix tree construction we give utilizing our LCPQ oracle ties the KMR approach and allows us to improve the communication rounds and total memory in AMPC (see Section 7 for details).

In the KMR-based algorithm, we recursively define a hash function that we use in an $O(\frac{1}{\epsilon})$-round, $\widetilde{O}(n^{1+\epsilon})$-memory MPC algorithm for computing suffix array. First, we define this recursive hash function which maps substrings of an input string s to integers. When sorting strings in the construction of the KMR hash, we assume that we always sort in ascending order lexicographically. Additionally, if we refer to a substring $s[i, j]$ where $j \ge |s|$, we instead consider the substring $s[i, |s|)$.

Definition 2.1. Given a strings s and $\epsilon \in (0, 1)$, we build a KMR-based hash function $\mathcal{K}_{s,\epsilon}$. For all $1 \le i \le \frac{1}{\epsilon}$, $0 \le j < n$, $\mathcal{K}_{s,\epsilon}(i, j)$ denotes a hash of $s[j, j + n^{i\epsilon}]$ defined as follows:

1. Sort all n^{ϵ}-length substrings of s and remove duplicates from the list. For all $0 \le j < n$, let $\mathcal{K}_{s,\epsilon}(1, j)$ be the position of substring $s[j, j + n^{\epsilon})$ in this list.
2. Assume the hash function $\mathcal{K}_{s,\epsilon}(i-1, j)$ has been defined for all $0 \le j < n$ where $1 \le i \le \frac{1}{\epsilon}$. For all $0 \le j < n$, sort tuples $(\mathcal{K}_{s,\epsilon}(i-1, j), \mathcal{K}_{s,\epsilon}(i-1, j + n^{(i-1)\epsilon}), \mathcal{K}_{s,\epsilon}(i-1, j + 2n^{(i-1)\epsilon}), \dots, \mathcal{K}_{s,\epsilon}(i-1, j + (n^{(i-1)\epsilon} - 1)n^{(i-1)\epsilon}))$ by comparing elements of the tuples from left to right and removing duplicate tuples from the list. For all $0 \le j < n$, let $\mathcal{K}_{s,\epsilon}(i, j)$ be the position of substring tuple $(\mathcal{K}_{s,\epsilon}(i-1, j), \mathcal{K}_{s,\epsilon}(i-1, j + n^{(i-1)\epsilon}), \mathcal{K}_{s,\epsilon}(i-1, j + 2n^{(i-1)\epsilon}), \dots, \mathcal{K}_{s,\epsilon}(i-1, j + (n^{(i-1)\epsilon} - 1)n^{(i-1)\epsilon}))$ in the sorted list.

Sorting in MPC can be done in constant rounds [16], so constructing the KMR-based hash function can be completed straightforwardly. Additionally, we prove the fact that by definition $\mathcal{K}_{s,\epsilon}(i,j)$ gives the position of the jth suffix of s in the suffix array when $i = 1/\epsilon$, so computing all such KMR-based hash values suffices for constructing the suffix array of s. Proof of this fact and the following theorems can be found in the full version of the paper.

Theorem 2.2. *For $\epsilon \in (0,1]$, there is an $O(\frac{1}{\epsilon})$-round* MPC *algorithm for a given string s using $\mathcal{K}_{s,\epsilon}$ for computing the* **suffix array** *of s in $\tilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor.*

With this suffix array construction, we may then build the suffix tree of any input string in constant rounds of MPC using the ideas of a PRAM algorithm from [19] (proof in full version).

Theorem 2.3. *For $\epsilon \in (0,1]$, there is an $O(\frac{1}{\epsilon})$-round* MPC *algorithm for suffix tree using $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

In the sequential setting, LPS and LCS each have folklore solutions in linear time using suffix tree. We now describe these folklore solutions: for LPS, one can build a suffix tree on the input string $s = s_0 s_1 \dots s_{n-1}$ concatenated with the reverse string $\bar{s} = s_{n-1} \dots s_1 s_0$. The longest palindrome will then be the two suffixes in the tree with the lowest common ancestor where one suffix is from s and one suffix is from $\bar{s}$. Similarly for LCS, we build a suffix tree on the two input strings s and s' concatenated together and find the two suffixes with the lowest common ancestor in the suffix tree where one suffix is from s and the other is from s'. These reductions are not completely trivial to implement in MPC, and we give a version of them formally in the full paper after giving our own novel suffix tree construction using our Longest Common Prefix Query oracle.

Theorem 2.4. *For $\epsilon \in (0,1]$, there are $O(\frac{1}{\epsilon})$-round* MPC *algorithms for* LCS *and* LPS *using $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

Now that we've shown a simple approach to some location-sensitive string problems and suffix tree, we next discuss our techniques involving the LCP problem which matches the state-of-the-art results for suffix tree construction in MPC and improves the number of rounds and total space for LPS and LCS. Additionally, these techniques will allow us to give a linear space algorithm for suffix tree construction in the AMPC model, which is not possible with the KMR approach.

3 OUR RESULTS AND TECHNIQUES

We present a general framework for solving *location-sensitive* string problems in the MPC model. At the core of most of our algorithms, we rely on a novel data structure we call an LCPQ oracle which handles a large number of *Longest Common Prefix* (LCP) queries simultaneously in MPC. For example, when searching for the longest palindrome in a string it is necessary to do some sort of lexicographic comparisons to find the lengths of palindromes within the string. The LCPQ oracle allows us to efficiently evaluate such comparisons, and throughout this work, we will often reduce *location-sensitive* string problems to a polynomial number of LCP queries.

Problem	Model	Source	Rounds	Total Memory
LCPQ Oracle (with k queries)	MPC	Lemma 3.1	$O(1)$	$\widetilde{O}(n^{1+\epsilon}+k)$
		Theorem 3.2	$O(1)$	$\widetilde{O}(n+k+\min(n,k)\cdot n^{\epsilon})$
		Theorem 5.1	$O(\log\log n)$	$\widetilde{O}(n+k)$
LPS	MPC	Theorem 3.3	$O(1)$	$\widetilde{O}(n)$
LCS	MPC	Theorem 3.4	$O(1)$	$\widetilde{O}(n^{1+\epsilon})$
Suffix Array/Tree	MPC	Theorem 3.5	$O(1/\epsilon)$	$\tilde{O}(n^{1+\epsilon})$
Suffix Tree	AMPC	Theorem 3.6	$O(1)$	$\widetilde{O}(n)$

Table 1: The above table shows a summary of our results for LCPQ oracle, LPS, and LCS problems in MPC, as well as suffix tree constructions in MPC and AMPC using the LCPQ oracle. The local memory of each machine is $\widetilde{O}(n^{1-\epsilon})$ for a small constant ϵ.

3.1 LCPQ Oracle

Given two strings s and s' of length n, an LCP query $q = (i,j)$ returns the length of the longest common prefix of the suffixes of s and s' which start at indices i and j respectively, i.e., the LCP of $s[i:n)$ and $s'[j:n)$. The main challenge of finding the longest common prefix between two suffixes is that a suffix may be very long, up to $O(n)$ if it includes most of the string. A single machine cannot compare two such suffixes of length $O(n)$ in its $O(n^{1-\epsilon})$-size local memory, and furthermore, a naive approach to computing the longest common prefix by comparing substrings of length $O(n^{1-\epsilon})$ sequentially could require $O(n^{\epsilon})$ rounds for long suffixes.

Problem	Model	Source	Rounds	Notes
LPS	PRAM	[2]	$O(\log n)$	$O(n^2)$ memory/work
String Matching	MPC	[18]	$O(1)$	$O(n)$ total memory
Suffix Array/Tree	PRAM	[19]	$O(\log(n))$	$\tilde{O}(n)$ memory/work

Table 2: The existing results for LPS and suffix tree in the PRAM model, and string matching in MPC.

By hashing every substring of size n^{ϵ}, which we call *blocks*, and storing the hashes of subsequent blocks in a single machine, we can compare hashes of entire suffixes even of length $O(n)$ on a single machine. For example, for a suffix $s[i,n]$ we will store on a single machine the hashes of blocks $s[i, i+n^{\epsilon}]$, $s[i+n^{\epsilon}, i+2n^{\epsilon}]$, etc. Notice that the starting index of each of the blocks considered in the previous example has a remainder of i modulo n^{ϵ}. Grouping blocks by their remainders after a modulo operation is a classic algorithmic technique, and we call this idea *Modular Partitioning*. Modular partitioning is the main way we can build and compare suffixes from their composite blocks efficiently on a single machine. Given query $q = (i,j)$, we then just have to make sure that the hashes of all blocks of suffixes $s[i:n)$ and $s'[j,n)$ are stored on the same machine together in order to find a rough estimate of the longest common prefix between them.

Once we know the specific blocks where the prefixes no longer match, we can then easily compare these blocks of size $O(n^{\epsilon})$ by sending a query to a machine that contains the (not hashed) substrings of each block using a single round of communication. Since

each string can be broken into n^ϵ blocks of size $n^{1-\epsilon}$, in order to compare blocks directly we only need $O(n^{2\epsilon})$ machines which will each contain a unique pair of blocks from the input strings. This is not a novel technique, but we formalize this idea of *block-based data structures* in Section 4.1 since it is common to all of our MPC algorithms. The resulting data structure can answer a collection of $k = O(n^{1+\epsilon})$ arbitrary LCP queries $Q = \{q_1, q_2, \ldots, q_k\}$ in $O(1)$ rounds.

Since our LCPQ oracle construction relies on hashing, our algorithms are necessarily randomized and succeed *with high probability*[3]. Additionally, our algorithms may need to try $O(\log n)$ different large prime numbers as the modulo base in our hash functions. To avoid $\log n$ communication rounds, we may try these prime numbers in parallel at the cost of an additional $O(\log n)$ factor in our space constraint. Furthermore, to perform modular partitioning, clearly local memory $O(n^{1-\epsilon})$ must be larger than $O(n^\epsilon)$ in order to find the hashes of blocks of length n^ϵ. Therefore, as discussed in the introduction, we bound ϵ by $0 < \epsilon \leq .5$ to make sure local memory $O(n^{1-\epsilon})$ is always large enough for our algorithms.

Lemma 3.1. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm, with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per machine, which initializes the* **Longest Common Prefix Query (LCPQ) Oracle** *in $O(1)$ rounds w.h.p., and then computes a collection of $k = O(n^{1+\epsilon})$ queries, $Q = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds.*

With the previous techniques as a starting point, we give a more efficient LCPQ oracle, one of our main contributions, in Section 5. Specifically, we show that for any value of k which is either $O(n^{1-\epsilon})$ or $\Omega(n^{1+\epsilon})$, we can solve a batch of k queries in $O(1)$ rounds of MPC using tight total memory $O(n+k)$. For the values of $k = \omega(n^{1-\epsilon}) \cap o(n^{1+\epsilon})$, we present a compressed LCPQ oracle with $O(n + k + \min(n, k) \cdot n^\epsilon)$ total memory. Intuitively, since the bottleneck is $\min(n, k) \cdot n^\epsilon$ for the values of k in this range, we end up having a sub-optimal total memory size if n and k are too close (within an $O(n^\epsilon)$ multiplicative factor of each other); we introduce a different approach utilizing *modular partitioning* to show that solving the LCP queries with a total of $O(n+k)$ memory is achievable if we allow $O(\log \log n)$ rounds of communication in MPC (Lemma 5.1). In the uncompressed LCPQ oracle algorithm discussed before, recall that we hash blocks of size n^ϵ until we find the specific blocks where the prefix of two suffixes no longer match. We can simply repeat this process for blocks of size $n^{\epsilon/2}$, $n^{\epsilon/4}$, etc. to find smaller and smaller non-matching blocks until we find the exact longest common prefix length in at most $O(\log \log n)$ rounds. See Figure 2 or Table 3 to find more precise details about the performance of a *compressed* LCPQ oracle.

Theorem 3.2. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm with $\widetilde{O}(n^{1-\epsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $Q = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n + k + \min(n, k) \cdot n^\epsilon)$.*

[3]w.h.p. (with high probability) should have an exponentially small probability of failure $\frac{1}{n^c}$ for some constant c we can choose.

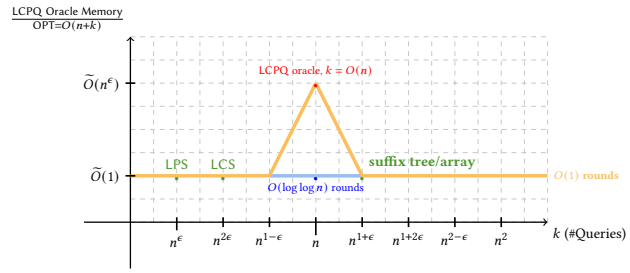

Figure 2: This plot illustrates the ratio of the memory blowup in a *compressed* LCPQ oracle for different ranges of k where k denotes the number of LCP queries in a batch. The vertical axis shows the ratio of memory blowup of the LCPQ oracle given query set Q of size k compared to the optimal memory $O(n + k)$. The horizontal axis shows the number of queries k. Both axes are logarithmic. The yellow path shows the memory performance of our best algorithm for a compressed LCPQ oracle in $O(1)$ rounds. The green points – corresponding to problems LPS (Section 6), LCS, and suffix tree (details in full version of the paper) – show the problems in which we use memory-tight variants of the LCPQ oracle. The red point shows the problem of LCPQ oracle with $k = O(n)$ queries, in which we use sub-optimal memory for the LCPQ oracle, and the blue point demonstrate a memory-tight algorithm for the same problem if we allow $O(\log \log n)$ communication rounds in MPC.

3.2 LPS, LCS, and the Periodicity Properties

In Section 6, we present another main result, an algorithm for LPS using linear total memory and $O(1)$ communication rounds. A naive approach to solving LPS in the MPC model would be to distribute blocks of the input string to processors and have each processor find palindrome candidates. For any palindrome candidate, to find the actual full length of any palindrome would require comparing any two pairs of blocks since the palindrome may span the entire string, and to find the full length of many palindromes naively would require either super-linear memory or non-constant rounds. It might be tempting to suggest an MPC algorithm which queries the LCPQ oracle for all indices, and finds the longest palindrome centered at each location subsequently. However according to Figure 2, we end up with either a super-linear total memory or $O(\log \log n)$ rounds since we need to query every index and $k = O(n)$.

To avoid having to check each palindrome, we use an idea from the PRAM setting [2] to analyze the periodicity of a string and limit the number of LCP queries we need to compute to be sub-linear. Complemented by the novel compressed LCPQ oracle (see Section 5), our algorithm finds the maximum length palindrome per block using tight memory and constant rounds, as described in Theorem 3.3.

Theorem 3.3. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Palindrome Substring (LPS)** *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

Similarly, in the full version of the work we present an algorithm for LCS that improves the communication rounds of prior techniques from $O(1/\epsilon)$ rounds to $O(1)$ rounds. The algorithm uses $\widetilde{O}(n^{1+\epsilon})$ total memory and is based on the periodicity properties of a pair of blocks with more than one LCS candidate pair. A *candidate pair* $p = (i, j)$ is a pair of suffixes i and j in two given strings s and s' respectively, i.e. $s[i : n)$ and $s'[j : n)$, such that $\mathsf{LCP}(q) \geq 3n^{1-\epsilon}$ where $q = (i, j)$ is an LCP query for those pair of suffixes. As it is clear from the definition of the candidate pairs, they represent maximal common substrings with length at least $3n^{1-\epsilon}$. We use *pairwise block machines* to compute the longest common substring with length smaller than $3n^{1-\epsilon}$ by an exhaustive search. Therefore, it only remains to find the result of LCP queries for each candidate pair and take the maximum value.

The number of maximal candidate pairs excluding the pair of blocks with the same period string is bounded by $O(n^{2\epsilon})$. However, if we include the candidate pairs for a common periodic range in both strings the number of LCP queries increases to $O(n)$. We can further reduce the total number of candidate pairs to $O(n^{2\epsilon})$ by finding the common periodic range imposed by a collection of more than one candidate pairs in a pair of blocks. The subsequent pair of indices in s and s' after the periodic range determines the candidate pair with the maximum LCP in this pair of blocks. Thus, there are a collection of $O(n^{2\epsilon})$ candidate pairs, and the maximum LCP among these candidate pairs is equal to the LCS of the input strings s and s'.

Theorem 3.4. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Common Substring (**LCS**)** *with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

3.3 Massively Parallel Suffix Tree using LCPQ

In our final section, we use the stronger AMPC model to improve upon the suffix array and suffix tree constructions. To this end, we give our own LCPQ oracle-based algorithm for building the *suffix array* of a given string with $O(n^{1+\epsilon})$ total memory and $O(1/\epsilon)$ communication rounds in the full version of the paper. Using our suffix array to suffix tree conversion, this allows us to build the suffix tree with our own LCPQ approach that ties the state-of-the-art KMR approach from Section 2. Moreover, we show that unlike the KMR approach, this suffix array approach can be used in AMPC to improve suffix tree construction from $O(1/\epsilon)$ communication rounds and $\widetilde{O}(n^{1+\epsilon})$ total memory to $O(1)$ rounds and $\widetilde{O}(n)$ total memory.

For our LCPQ-based suffix array construction, we start by sorting the suffixes of a given input string by utilizing an LCPQ oracle which we initialize for string s and itself, to support a comparison-based parallel quick sort in total space $\tilde{O}(n^{1+\epsilon})$ similar to the one discussed in [16]. At each round, we choose n^{ϵ} random pivots for every remaining unsorted interval. In this process, the maximum interval size at round i is bounded by $\widetilde{O}(n^{1-\frac{i\epsilon}{2}})$ with high probability. Thus, after $O(1/\epsilon)$ rounds, we end up with $\widetilde{O}(n^{\frac{\epsilon}{2}})$-sized intervals which we sort locally using the LCPQ oracle. For comparing a pair of suffixes (i, j), we need to compare characters $s_{i'}$ and $s_{j'}$, the first unequal pair of characters we encounter if we start moving forward from indices i and j at the same rate. It is easy to observe that by definition $i' = i + \mathsf{LCP}_{s,s}(i, j)$ and $j' = j + \mathsf{LCP}_{s,s}(i, j)$, and so the LCPQ oracle provides all the information we need to find these indices.

Theorem 3.5. *For $\epsilon \in (0, .5]$, there is an $O(1/\epsilon)$-round* MPC *algorithm for computing the* **suffix array** *of a given string s with $\widetilde{O}(n^{1+\epsilon})$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor w.h.p..*

3.4 Suffix Tree in AMPC

Last but not least, we observe that in the *Adaptive Massively Parallel Computation* (AMPC) model, it is possible to build a fully-functional suffix tree[4] using $\widetilde{O}(n)$ total memory and in $O(1/(1-\epsilon))$ rounds. Note that LCP queries are handled by $O(\log n)$ sequential adaptive queries as we perform a binary search to find the length of an LCP query (i, j), i.e., the maximum l where $s[i : i+l)$ and $s'[j : j+l)$ have the same hash value. We retrieve the hash of a substring $s[l : r)$ using a *block-based* hashing data structure by spending only $O(1)$ adaptive queries. Since we use the $O(n)$ shared memory to store hashes, we avoid the requirement that $\epsilon \leq .5$, which stemmed from modular partitioning and the MPC versions of the block-based data structures.

Theorem 3.6. (Suffix Tree in AMPC) . *For $\epsilon \in (0, 1]$, there is an $O(1)$-round* AMPC *algorithm for computing the* **suffix tree** *of a given string s with $\tilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

We do not give the statements and their proofs, but utilizing the fully-functional suffix tree to solve LPS and LCS in $O(1)$ rounds and linear memory in AMPC easily follows using the reductions to suffix tree shown for MPC.

4 MPC IMPLEMENTATION DETAILS

In this section, we summarize a set of tools we frequently use in our MPC algorithms. First we introduce a technique called *Modular Partitioning* that is used to construct LCPQ oracles in Section 5, which are an important building blocks of most of our algorithms. Next, we formalize two folklore MPC techniques called *Block-based Data Structures* and *Weighted Load Balancing*. We extensively use block-based data structures as auxiliary tools in the reduction of LPS and LCS to suffix tree in the full version of the paper, as well as in the memory-optimal algorithms for LPS in Section 6. Moreover, weighted load balancing is used in the construction of LCPQ oracles and suffix trees as well as in the reductions from LPS and LCS to suffix tree.

At their cores, our algorithms use *hashing*, and thus all of our results are randomized and succeed *with high probability*. There are additional sources of randomness in our algorithms such as the randomness induced by quick sort for finding the suffix array, but hashing is used in all algorithms described in this paper. We define the hash of a substring $s[l : r)$ as the following for a large enough

[4] By a *fully-functional* suffix tree we mean one including backward and forward pointers as opposed to suffix tree we build in the MPC model which only captures the structure of the tree. A fully-functional suffix tree rarely relies on the suffix array and auxiliary block-based data structures for solving location-sensitive problems.

prime number p.[5]

$$\mathsf{hash}\big(s[l:r]\big) = \Big(\sum_{i=l}^{r-1} s_i \cdot |\Sigma|^{r-1-i}\Big) \bmod p \tag{1}$$

4.1 Block-based Data Structures

The input string s is initially partitioned into n^ϵ consecutive *blocks* of size $O(n^{1-\epsilon})$, denoted by $b_0, b_1, \ldots, b_{n^\epsilon-1}$, so that each block fits into the memory of a single machine. We denote the machine which stores b_α by μ_α. Furthermore, we assume that μ_α has local access to the next block $b_{\alpha+1}$ and previous block $b_{\alpha-1}$ if either exists. Similarly, in problems that have two input strings s and s', we denote the blocks of s' by b'_β, and the dedicated machine that stores this block by μ'_β. In this paper, we use α and β variables when referring to the index of a block in s and s' respectively. We also always assume w.l.o.g. that both strings have length n. Otherwise, we pad the smaller one with enough number of '#' character, where '#' $\notin \Sigma$, to make their sizes equal.

Definition 4.1 (Blocks b_α and Block Machines μ_α). There is a collection of n^ϵ machines $\{\mu_0, \mu_1, \ldots, \mu_{n^\epsilon-1}\}$ referred to as ***block machines*** for string s (and $\{\mu'_0, \mu'_1, \ldots, \mu'_{n^\epsilon-1}\}$ for string s'), where μ_α (and μ'_β) contains the substring of the α-th ***block*** of s (and the β-th block of s') denoted by $b_\alpha = s[\alpha n^{1-\epsilon} : (\alpha+1)n^{1-\epsilon})$.

We often use *block-based* data structures in our algorithms which is a standard technique in the MPC model. Roughly speaking, in a block-based data structure we gather a constant amount of information about every block b_α, say $f(b_\alpha)$[6], so that $\{f(b_0), f(b_1), \ldots, f(b_{n^\epsilon-1})\}$ can be copied and fit inside the memory of a single machine as long as $\epsilon \le 0.5$. Furthermore, we use a *segment tree* inside each machine μ_α which computes $f(b_\alpha[l:r])$ for a total of $O(n^{1-\epsilon})$ substrings $b_\alpha[l:r]$ inside block b_α. Such data structures can answer a batch of $O(n)$ queries $f(s[l:r])$ about arbitrary substrings of s in 2 rounds of MPC. For example, we can define $f(s) = \mathsf{hash}(s)$, and thus query the hash of an arbitrary batch of $O(n)$ substrings efficiently in MPC. Function f needs to have the property that there exists a merge function g so that $f(s) = g\big(f(s[0:i]), f(s[i:n]), i, n\big)$ for every $0 < i < n$. For the example function hash as defined in (1), the corresponding function merge function g equals:

$$g\big(\mathsf{hash}(s[0:i]), \mathsf{hash}(s[i:n]), i, n\big) = \big(\mathsf{hash}(s[0:i]) \cdot |\Sigma|^{n-i} + \mathsf{hash}(s[i:n])\big) \bmod p$$

4.2 Modular Partitioning

The *Modular Partitioning* of an array of length n is a partition of its elements based on their remainder when dividing by n^ϵ. Recall the definition of block machines μ_α from Definition 4.1. Within each block b_α, μ_α computes the hash of every substring of length n^ϵ starting in b_α, i.e., $\mathsf{hash}(s[i : i+n^\epsilon])$ for every $\alpha n^\epsilon \le i < \min\{(\alpha+1)n^\epsilon, n\}$[7]. We allocate a machine λ_x for every $0 \le x < n^\epsilon$ and gather into λ_x the hash values of every substring $s[i : i+n^\epsilon)$ such that $i \bmod n^\epsilon = x$. In other words, we are assigning each index i to one of n^ϵ machines $\lambda_0, \lambda_1, \ldots, \lambda_{(n^\epsilon-1)}$ so that indices assigned to each machine have the same remainder in division by n^ϵ. This simple trick allows us to compute the hash of any substring whose length is a multiple of n^ϵ inside the local memory of some λ_x. A substring $s[i : i+un^\epsilon)$ is a concatenation of several substrings of length n^ϵ, specifically $s[i : i+n^\epsilon) + s[i+n^\epsilon : i+2n^\epsilon) + \ldots + s[i+(u-1)n^\epsilon : i+un^\epsilon)$, all of which are assigned to the same machine $\lambda_{(i \bmod n^\epsilon)}$. For an example, see Figure 3 in which block machines μ_α and mod machines λ_x are specified for an example string s.

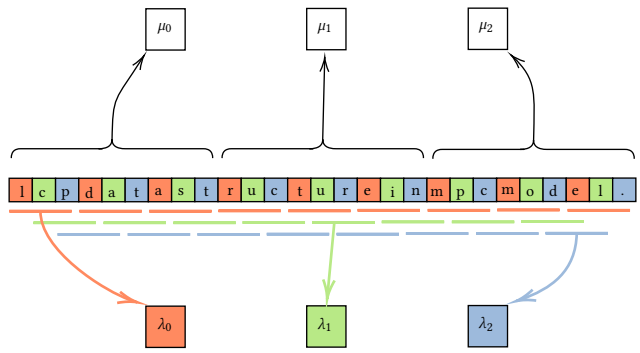

Figure 3: The modular partitioning is illustrated in this figure. In this example, $n = 27$ and $\epsilon = 1/3$, which means $n^\epsilon = 3$ and $n^{1-\epsilon} = 9$. There are 3 *block machines* μ_0, μ_1, and μ_2 which store the substrings of s corresponding to blocks b_0, b_1, and b_2 respectively. There are also 3 *mod machines* λ_0, λ_1, and λ_2. Each λ_x for $0 \le x \le 2$, stores the hash of every substring of length $n^\epsilon = 3$ which start at a location i so that $i \bmod 3 = x$.

Definition 4.2. We denote a collection of n^ϵ ***mod machines*** $\{\lambda_0, \lambda_1, \ldots, \lambda_{n^\epsilon-1}\}$ for string s (and $\{\lambda'_0, \lambda'_1, \ldots, \lambda'_{n^\epsilon-1}\}$ for string s'), where λ_x (and λ'_x) contains the hash of substrings of length $O(n^\epsilon)$ starting at indices i so that $i \bmod n^\epsilon = x$.

4.3 Weighted Load Balancing

Our machine and data structures are often required to answer large amounts of queries in parallel, i.e. $O(n)$ or $O(n^{1+\epsilon})$ queries. We first partition a query set into groups so that all queries that must be routed to the same machine are together. Unfortunately, more queries than the size of the local memory of a machine may be routed to a single machine. We use a 2-round algorithm *Weighted Load Balancing* to replicate the data stored in each machine proportionally to the number of queries assigned to that machine so that the local memory constraint is never violated.

We take our LCP query set Q as an example where each query $q \in Q$ is a pair of indices (i, j) such that we want to find the LCP between the suffixes starting at index i in one string and j in another. We partition Q into $n^{2\epsilon}$ groups $Q_{x,y}$ based on the remainders of dividing both coordinates of each query by n^ϵ. Each group $Q_{x,y}$ contains all the queries $q = (i, j)$ such that $i \bmod n^\epsilon = x$ and $j \bmod n^\epsilon = y$. We may assign small groups, ones with size $O(n^{1-\epsilon})$, to one of $\lambda_{x,y}$ machines without violating the memory constraints. However, sometimes a large number of queries are concentrated in

[5] We can pick a set of $O(\log(n))$ random prime numbers, and process them in parallel, so that our algorithm succeeds *with high probability*.
[6] Function f can also be defined on any arbitrary array with length n instead of the string itself.
[7] Note that we have a local access to the next block, i.e., $b_{\alpha+1}$, inside μ_α.

specific groups. The input queries might even consist of a single group of size $O(n^{1+\epsilon})$. To fix this issue, we replicate the data stored in each machine based on the number of queries assigned to it. See Algorithm 1 for more details. In Algorithm 1, the data with total size $O(n^{1+\epsilon})$ is partitioned into $n^{2\epsilon}$ groups $P_{x,y}$, and we aim to distribute the queries among a collection of $O(n^{2\epsilon})$ machines $\xi_{x,y}$ with possible repetitions[8].

Algorithm 1: WeightedLoadBalancing($\mathcal{S} = \{\xi_1, \xi_2, ...\xi_m\}, \{P_1, P_2, ...P_m\}$)

for $\xi_x \in \mathcal{S}$ **do**
  $c_x \leftarrow \lceil \frac{|P_x|}{n^{1-\epsilon}} \rceil$;
  Replicate the data stored in machine ξ_x into c_x copies $\xi_x^{(1)}, \xi_x^{(2)}, \ldots, \xi_x^{(c_x)}$;
  Partition the elements of P_x into c_x groups $P_x^{(1)}, P_x^{(2)}, \ldots, P_x^{(c_x)}$ of almost equal size;
  Send $P_x^{(z)}$ to $\xi_x^{(z)}$ for every $1 \le z \le c_x$;
end
return $\{c_1, c_2, ..., c_m\}$;

5 COMPRESSED LCPQ ORACLE

In this section, we study instances of the LCPQ oracle in which there are a smaller collection of queries $Q = \{q_1, q_2, \ldots, q_k\}$, for example $k = O(n)$ as opposed to $O(n^{1+\epsilon})$ inspired by the instances of LCP that we later consider for LPS in Section 6. The ultimate question is whether it is possible to handle k queries for any value of k using $\widetilde{O}(n+k)$ total memory in a constant number of rounds since we need at least $\Omega(n+k)$ total memory to store the strings and queries. However, it is not possible to always get $\widetilde{O}(n+k)$ memory using our techniques. So the natural question is for which range of values for k can we handle k queries in $O(1)$ rounds of MPC with $\widetilde{O}(n+k)$ total memory. In this section, we study this problem thoroughly and optimize it for different ranges of k, which is summarized in Table 3 as well as in Figure 2.

We present an alternative approach for solving a batch of k LCP queries. The total memory used by this approach equals $\widetilde{O}(n+kn^{\epsilon})$ instead of the $\widetilde{O}(n^{1+\epsilon}+k)$ total memory used in Lemma 3.1. This implies a constant round MPC algorithm with total memory $\widetilde{O}(n+k+\min(n,k)\cdot n^{\epsilon})$ if we use Lemma 3.1 when $k = \Omega(n)$ and our alternative approach when $k = O(n)$. Thus, the algorithm uses $\widetilde{O}(n+k+\min(n,k)\cdot n^{\epsilon})$ total memory since it needs to store the strings and queries.

Theorem 3.2. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round MPC algorithm with $\widetilde{O}(n^{1-\epsilon})$ memory per machine which initializes an* LCPQ **oracle** *in $O(1)$ rounds w.h.p., and then processes a collection of k queries, $Q = \{q_1, q_2, \ldots, q_k\}$, in $O(1)$ rounds. The total memory used by this algorithm is $\widetilde{O}(n+k+\min(n,k)\cdot n^{\epsilon})$.*

[8]In the definition of Algorithm 1, we label the servers and query groups with subscript indices in the range $[1, m]$. However, in our actual applications, servers may have any labelling such as a pair of subscripts, e.g. $\lambda_{x,y}$. The only requirement is that a query in P_i is expected to be sent to a copy of ξ_i for any ξ_i of the server set.

However, for the values of k in the range from $\omega(n^{1-\epsilon})$ to $o(n^{1+\epsilon})$, we still get a sub-optimal total memory of $O(n+k+\min(n,k)\cdot n^{\epsilon}) = \omega(n+k)$. In this case, we show an $O(\log\log n)$-round MPC algorithm with optimal total memory $O(n+k)$ which solves a batch of LCP queries of size $O(k)$ (See Theorem 5.1 for more details). Our memory-optimal algorithms for different ranges of k are demonstrated in Table 3.

#Queries	Source	Rounds	Total Memory
$k = O(n^{1-\epsilon})$		$O(1)$	$O(n+k\cdot n^{\epsilon}) = O(n)$
$k = \omega(n^{1-\epsilon})$, and $k = o(n^{1+\epsilon})$	Theorem 5.1	$O(1)$ $O(\log\log n)$	$O(n+k+\min(n,k)\cdot n^{\epsilon})$ $O(n+k)$
$k = \Omega(n^{1+\epsilon})$	Lemma 3.1	$O(1)$	$O(n^{1+\epsilon}+k) = O(k)$

Table 3: The above table shows the different memory-optimal results we show in this section for different ranges of k, the number of LCP **queries.**

Theorem 5.1. *For $0 < \epsilon < 1/2$, there is an $O(\log\log n)$-round* MPC *algorithm, with $\widetilde{O}(n+k)$ total memory and $\widetilde{O}(n^{\epsilon})$ memory per machine, which uses the* LCPQ Oracle *to compute a collection of k queries, $Q = \{q_1, q_2, \ldots, q_k\}$, w.h.p.*

6 LONGEST PALINDROME IN MPC

In this section, we give an improved solution to *Longest Palindrome Search* (LPS). Missing algorithms and full proofs may be found in the full version of the paper. Given a string $s = s_0 s_1 ... s_{n-1}$, let $\bar{s} = s_{n-1} ... s_1 s_0$ be the reverse of s. Note we also use $s[j : i)$ to represent the reverse of a substring $s[i+1 : j+1)$ for $i < j$. A *palindrome* contained in a string s is a substring c of s such that $c = \bar{c}$. If c_0 is a palindrome in s such that for all other palindromes c_1 in s, $|c_0| > |c_1|$, then $\mathsf{LPS}(s) = c_0$. Our algorithm, Algorithm 6 below, will use $O(n^{\epsilon})$ processors each with $\widetilde{O}(n^{1-\epsilon})$ memory for $\epsilon < .5$ and finish in $O(1)$ rounds. In LPS, finding short palindromes locally on an individual processor is very easy. However, like many of the other problems in the paper, finding long palindromes that have length larger than the memory of a processor becomes difficult since it requires coordination between processors to calculate the exact length. Moreover, it is possible that there are many such long palindromes, even as much as $\Omega(n)$ palindrome candidates. The existence of multiple palindromes that overlap in a string implies that the section of the string containing these palindromes is periodic. For example, if there are two palindromes, a first palindrome to the left of a second palindrome, then the left side of the first palindrome must match its right side, which overlaps the left side of the second palindrome. Thus, each left side must match and therefore, this left side is exactly the repeated substring of this section of the string. We call this repeated substring the *period* of a substring if it spans the entire substring. By finding periods of each block of length $O(n^{1-\epsilon})$ of the original string s, we can keep track of palindrome candidates and find the largest palindrome per period without needing to check all $\Omega(n)$ candidates across the entire string, which would otherwise require either a non-constant number of rounds or super-linear memory.

Our algorithm for LPS, based on the properties of periodic substrings proven in [2], will be split into several steps. At the start of our main algorithm (Algorithm 6), we break the input string s into blocks b_α of size $n^{1-\epsilon}$ where $n = |s|, 0 \leq \alpha < n^\epsilon$. We then utilize the following algorithm, Algorithm 3, for finding palindrome candidates and a period string on each block. After finding each period w_α for block b_α, we find the exact range of the period and use the center of this range as our palindrome candidate for period w_α. Once we have our final set of palindrome candidates per period, we can use the LCPQ oracle to find the length of each palindrome by finding the longest common prefix of a palindrome center in string s with the corresponding center in $\bar{s}$ and return the longest palindrome.

The first algorithm we formally write, Algorithm 3, takes three adjacent blocks $b_{\alpha-1}, b_\alpha, b_{\alpha+1}$ as input and returns the period of block b_α if it is a periodic string as well as any potential palindrome candidates in case block b_α is not periodic. Algorithm 3 begins by storing the indices of palindrome centers from block b_α in an array C by checking if the block to the left of the center equals the block to the right. Without loss of generality we can assume all palindromes have a single character as their center since we can simply add an additional character not found in the string between every two characters of the string to preserve all palindromes while converting any palindromes with a two character center to a one character center. Therefore, each index stored in C will be the center of a palindrome of length at least $2n^{1-\epsilon} - 1$ characters, i.e. the palindrome spans the entire block b_α. Then, we exploit the fact that any string will have periodic characters if there are at least two different palindromes that span the length of the string. If there are several palindrome centers within a block b_α with palindrome length $2^\epsilon - 1$, then to find the minimum period length we can take the *Greatest Common Divisor* (GCD) of the pairwise differences of the indices of these centers multiplied by 2 to account for the mirrored left and right side of each palindrome. We use this length to pull a period string w from the beginning of the block and return w alongside C.

In the following two lemmas, we show that Algorithm 3 correctly finds minimum length period strings in space $O(n^{1-\epsilon})$ given blocks of size $n^{1-\epsilon}$. Again note that missing proofs and algorithms are in the full version of the work.

Lemma 6.1. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\epsilon}$, Algorithm 3 will return w, C where C contains all indices of b_α that are centers of palindromes of length at least $2n^{1-\epsilon}+1$ and w is the minimum length period string if $|C| \geq 2$.*

Lemma 6.2. *On input $t = b_{\alpha-1}b_\alpha b_{\alpha+1}$ with $|b_{\alpha-1}| = |b_\alpha| = |b_{\alpha+1}| = n^{1-\epsilon}$, Algorithm 3 uses $O(n^{1-\epsilon})$ memory.*

After finding the period and palindrome centers of a block b_α, we then find the range of blocks surrounding b_α with matching period. We build a generic "period block" W, the concatenation of the period string with itself $\lceil \frac{n^{1-\epsilon}}{|w|} \rceil$ times. By carefully rotating W, we can find a range of block indices denoted $[\ell_1, r_1]$ such that the blocks $b_\alpha, j \in [\ell_1, r_1]$, including b_α, form a large periodic substring of the original string s. Figure 4 shows an example of a period block W_1 for a block b_1 and its subsequent rotation. After finding this block range, we then also find the exact indices in blocks b_{ℓ_1} and b_{r_1} where the period ends. The following subroutines, performed by each processor μ_α in parallel, lists the above steps, formalized.

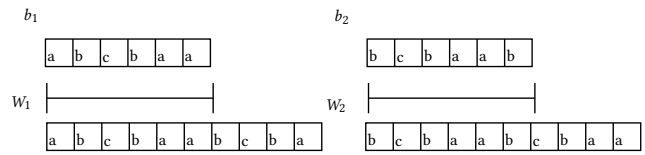

Figure 4: An example of two consecutive string blocks b_1 and b_2. Block b_1 has period w_1 = "*abcba*", which is used to build W_1 as well as W_2, a rotation of $W1$, used to check that b_2 follows the same period.

Lemma 6.3. *Given the period string w_α of periodic block b_α and an array of block string hashes, Algorithm 4 finds the period range (ℓ, r) of consecutive blocks surrounding block b_α that fully contains a periodic substring with period w_α in space $\widetilde{O}(n^{1-\epsilon})$ w.h.p..*

Lemma 6.4. *Given period w of block b_α and period range ℓ, r on processor μ_α, Algorithm 5 returns indices o_ℓ, o_r which are the indices of the leftmost and rightmost characters, respectively, denoting the last consecutive full period w in b_ℓ and b_r in space $\widetilde{O}(n^{1-\epsilon})$ and $O(1)$ rounds w.h.p.*

To wrap up the section, we present the full LPS algorithm for a string s on n^ϵ processors in the MPC model with $O(n)$ total memory and $O(1)$ rounds. First, each processor μ_α finds palindrome candidates and a period string w_α on block b_α using Algorithm 3. Each block then finds its surrounding period range using Algorithms 4 and 5. The center of the period range will be the longest palindrome center for all blocks in the period range and so we use the LCPQ oracle to check the length of the palindrome by finding how many characters to the left of the palindrome center match the characters to the right. After finding the longest palindrome on each block, we use a separate processor to compare each block's longest palindrome to find the maximum length palindrome across the entire string

Theorem 3.3. *For $\epsilon \in (0, .5]$, there is an $O(1)$-round* MPC *algorithm that solves* **Longest Palindrome Substring** (LPS) *with $\widetilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor, w.h.p.*

7 SUFFIX TREE IN AMPC

We will now show that in the AMPC model, we are able to improve our LCPQ oracle-based suffix array and suffix tree construction to $\tilde{O}(n)$ total memory and $O(1)$ communication rounds. The main flexibility that AMPC allows us from MPC is providing an $O(n)$ shared memory in which we can preprocess the input string s and answer $O(n)$ LCP queries within $\tilde{O}(n)$ space. In the preprocessing step, for all indices $i, 0 \leq i < n$, we store hash$(s[i : n))$. Utilizing these partial hashes, we can then find the hash of any interval in string s, and by performing a binary search on interval length, we can find exactly the interval where two prefixes match in $\log(n)$ memory. The following algorithm performs such a recursive search.

Lemma 7.1. *For a preprocessed string s, given any indices $i < j$ such that $0 \leq i < j < |s|$ and $s[i] = s[j]$, $k = 1$, and $o = n - j$*

Algorithm 2: LCP2Queries(i, j, o, k)

1 **if** $\text{hash}(s[i : i + o)) = \text{hash}(s[j : j + o))$ **then**
2 **if** $k = 1$ *or* $\text{hash}(s[i : i + o + 1)) \neq \text{hash}(s[j : j + o + 1))$ **then**
3 return o;
4 $o \leftarrow o + \lceil \frac{n-j}{2^k} \rceil$;
5 LCP2Queries$(i, j, o, k + 1)$;
6 **else**
7 $o \leftarrow o - \lceil \frac{n-j}{2^k} \rceil$;
8 LCP2Queries$(i, j, o, k + 1)$;

as input, Algorithm 2 computes the LCP of two indices in s in $O(1)$ rounds and $O(\log^2 n)$ total space with high probability.

Proof. Algorithm 2 is essentially performing a binary search on potential LCP lengths until it finds the exact length of the longest common palindrome between two prefixes of s starting at i and j. Therefore, we will prove the above claim by induction.

The algorithm ends in line 3 if the hash of the prefixes of s starting at i and starting at j are equal and $k = 1$, which means the entire suffix starting at i is equal to the suffix at j. If $k = 1$, then Algorithm 2 has just began and therefore $o = n - j$. Thus, if the hash of the prefixes of s starting at i and j are equal up to o characters, then the prefixes are completely equal up until the end of the string s and o is returned as the LCP length. Alternatively, if the hash of the prefixes are equal up to o characters but not $o + 1$, we also stop since we've found the LCP. If $\text{LCP}(i, j) = 1$, we will eventually stop due to this second condition since $s_i = s_j$ but $s_{i+1} \neq s_{j+1}$ when $k = \log n$. We will assume without loss of generality that $\text{LCP}(i, j) > 0$, which is easily verifiable without the need for hashing.

Assume that for the kth recursive call to Algorithm 2, the algorithm is correct. Consider the $(k+1)$th call. If $\text{LCP}(i, j) \geq o$, then we find in step 1 that $\text{hash}(s[i : i+o])=\text{hash}(s[j : j+o])$ and call the algorithm again with $o+\lceil \frac{n-j}{2^{k+1}} \rceil < o+\lceil \frac{n-j}{2^k} \rceil$. Similarly, if $\text{LCP}(i, j) < o$, we call the algorithm again with $o - \lceil \frac{n-j}{2^{k+1}} \rceil > o - \lceil \frac{n-j}{2^k} \rceil$. Since the algorithm was correct up to the kth recursive call, and we continue the search for the next smallest interval in both cases, the algorithm continues correctly. In the case that $\text{LCP}(i, j) = o$, then $\text{hash}(s[i : i + o + 1)) \neq \text{hash}(s[j : j + o + 1))$, and we still stop at step 3 correctly and output $\text{LCP}(i, j) = 0$.

This algorithm has $\log(n)$ recursive calls since k increases from 1 to at most $\log(n)$ before terminating. Since this algorithm runs locally on a single processor and for sufficiently large n, $n^\epsilon > \log(n)$, then Algorithm 2 uses only $O(1)$ rounds and requires $O(\log n)$ space per LCP query. We gain an additional $O(\log n)$ space factor to guarantee a good hash function with high probability. □

With our updated LCP query algorithm and Lemma 7.1, we have reduced the memory requirements of the LCPQ oracle construction from $O(n^{1+\epsilon})$ in MPC to $\tilde{O}(n)$ in AMPC with $O(1)$ rounds. We are also able to update the PivotFiltering parameters in AMPC to select $n^{1-\epsilon}$ pivots at once rather than n^ϵ. With high probability, we will then only need two rounds of pivot filtering to sort the suffixes. This change is seen in the third parameter at step 12.

With the updated LCPQ oracle we therefore can construct the suffix array π for a string s in $O(1)$ rounds and $\tilde{O}(n)$ memory since constructing the suffix array requires $O(n)$ queries per pivot sort and the LCPQ oracle requires $O(\log(n))$ memory per query. Furthermore, we can construct the suffix tree in the same space and rounds (see full version for details). Thus, we have our main AMPC result:

Theorem 3.6. (Suffix Tree in AMPC) . *For $\epsilon \in (0, 1]$, there is an $O(1)$-round* AMPC *algorithm for computing the* **suffix tree** *of a given string s with $\tilde{O}(n)$ total memory and $\widetilde{O}(n^{1-\epsilon})$ memory per processor with high probability.*

8 CONCLUSION

We have shown how to build a suffix tree in MPC in $O(n^{1+\epsilon})$ memory and constant rounds and in AMPC in $\tilde{O}(n)$ memory and constant rounds. For further improvement in MPC, one may look to improve the LCPQ oracle. Currently that is a main bottleneck that requires $O(n^{1+\epsilon})$ memory in MPC. Specifically, for an LCP query set Q of size k where $n^{1-\epsilon} < k < n^{1+\epsilon}$, a new solution to answer all queries in $O(n)$ total memory may be possible. Currently, we do not have this solution, and future work may be able to find such an LCPQ oracle. The following problem encapsulates this idea:

Problem 8.1. *Given a string s of length n, initialize an* LCPQ **oracle** *with linear memory in $O(1)$ rounds of* MPC *that is capable of handling $O(n)$ arbitrary* LCP *queries in $O(1)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory.*

Unfortunately, our current algorithm for suffix array construction (and therefore, suffix tree construction) requires $O(n^{1+\epsilon})$ comparisons when performing the pivot filtering. Therefore, even if the LCPQ structure is improved, our current suffix array algorithm would still require $O(n^{1+\epsilon})$ memory. If we can find a way to reduce the number of LCP queries needed, we may be able to have constant rounds and $\tilde{O}(n)$ total memory in MPC for suffix tree construction.

Problem 8.2. *Given a string s of length n, find the* **suffix tree** *in $O(1/\epsilon)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory.*

Finally, the memory for LCS may still be improved. Specifically, when the LCS of two strings is small, our current solution simply checks every pair of blocks for the small solution, of which there are $n^{2\epsilon}$ pairs. If all pairs of blocks would not be needed, and instead only $O(n^\epsilon)$ pairs, it would be possible to improve the LCS algorithm in MPC further as large solutions to LCS that span multiple blocks should be easier to detect than small solutions.

Problem 8.3. *Given two strings s and s' of length n, find* $\text{LCS}_{s,s'}$ *in $O(1)$ rounds of* MPC *with $\widetilde{O}(n)$ total memory. It is guaranteed that* $\text{LCS}_{s,s'} = o(n^{2\epsilon})$.

REFERENCES

[1] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel algorithms for geometric graph problems. In *STOC*. 574–583.
[2] Alberto Apostolico, Dany Breslauer, and Zvi Galil. 1994. Parallel Detection of all Palindromes in a String. In *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*. Springer, 497–506.
[3] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. 2019. Coresets Meet EDCS: Algorithms for Matching and Vertex Cover on Massive Graphs. In *SODA*. 1616–1635.

[4] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. 2019. Sublinear Algorithms for $\Delta + 1$ Vertex Coloring. In *SODA*. 767–786.

[5] Md Momin Al Aziz, Parimala Thulasiraman, and Noman Mohammed. 2020. Parallel Generalized Suffix Tree Construction for Genomic Data. In *Algorithms for Computational Biology - 7th International Conference, AlCoB 2020, Missoula, MT, USA, April 13-15, 2020, Proceedings*. 3–15. https://doi.org/10.1007/978-3-030-42266-0_1

[6] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication steps for parallel query processing. *Journal of the ACM (JACM)* 64, 6 (2017), 1–58.

[7] Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M Karp, and Jara Uitto. 2019. Massively parallel computation of matching and MIS in sparse graphs. In *ACM SIGACT*. 481–490.

[8] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Marina Knittel, and Hamed Saleh. 2019. Streaming and Massively Parallel Algorithms for Edge Coloring. In *ESA*. 15:1–15:14.

[9] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab S. Mirrokni, and Warren Schudy. 2019. Massively Parallel Computation via Remote Memory Access. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*. ACM, 59–68.

[10] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. 2019. Exponentially Faster Massively Parallel Maximal Matching. In *FOCS*. 1637–1649.

[11] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. 2018. Round compression for parallel matching algorithms. In *ACM SIGACT*. 471–484.

[12] Martin Farach and S. Muthukrishnan. 1996. Optimal Logarithmic Time Randomized Suffix Tree Construction. In *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1099)*, Friedhelm Meyer auf der Heide and Burkhard Monien (Eds.). Springer, 550–561.

[13] François Le Gall and Saeed Seddighin. 2020. Quantum Meets Fine-grained Complexity: Sublinear Time Quantum Algorithms for String Problems. *CoRR* abs/2010.12122 (2020).

[14] Mohsen Ghaffari, Christoph Grunau, and Ce Jin. 2020. Improved MPC algorithms for MIS, matching, and coloring on trees and beyond. *arXiv preprint arXiv:2002.09610* (2020).

[15] Mohsen Ghaffari and Jara Uitto. 2019. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *SODA*. SIAM, 1636–1653.

[16] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*. Springer, 374–383.

[17] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60.

[18] MohammadTaghi Hajiaghayi, Hamed Saleh, Saeed Seddighin, and Xiaorui Sun. 2021. String Matching with Wildcards in the Massively Parallel Computation Model. In *SPAA*. 275–284.

[19] Costas Iliopoulos and Wojciech Rytter. 2004. On parallel transformations of suffix arrays into suffix trees. In *Australasian Workshop on Combinatorial Algorithms (AWOCA)*, Vol. 11. 11–4.

[20] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA*. 938–948.

[21] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. 1972. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing* (Denver, Colorado, USA) *(STOC '72)*. 125–136.

[22] Mi Lu and Hua Lin. 1994. Parallel Algorithms for the Longest Common Subsequence Problem. *IEEE Trans. Parallel Distributed Syst.* 5, 8 (1994), 835–848.

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

[24] Edward M McCreight. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)* 23, 2 (1976), 262–272.

[25] Md Safiur Rahman Mahdi, Md Momin Al Aziz, Noman Mohammed, and Xiaoqian Jiang. 2021. Privacy-preserving string search on encrypted genomic data using a generalized suffix tree. *Informatics in Medicine Unlocked* 23 (2021), 100525. https://doi.org/10.1016/j.imu.2021.100525

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

[27] Peter Weiner. 1973. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, 1–11.